WaterSlide User Guide
Version 1.0

Updated March 2016

1 Introduction

WaterSlide is an event-at-a-time architecture for processing metadata. It is designed to take in a set of streaming events from multiple sources and process them through a set of modules and return meaningful outputs. Processing modules in WaterSlide are called "kids". The user of WaterSlide specifies a directed processing graph (or "pipeline") of kids that are used to process data. WaterSlide can process grouped events-at-a-time by using state-tracking data structures to turn events into event summaries.

WaterSlide is a dataflow processing environment [2]. Thus, data flows between user-specified operations to accomplish a variety of analysis tasks. Processing elements (i.e., kids) can be filters, aggregators, annotators, decoders, translators, and collectors. Code is only executed when data is made available to a processing operation.

You can use also WaterSlide as a streaming MapReduce framework [1, 4] for complex event processing. WaterSlide is designed to efficiently process both raw content (e.g. files, binary structures) and metadata about that content by minimizing data copies, grouping data, and reusing memory. It contains specially designed data structures intended to explore event correlation on a massive scale with data that is fragmented across processes and systems. These data structures go by funny names like "D-Left Digest Hashtable with Approximate Least Recently Used Expiration." As with most stream processing frameworks, many WaterSlide processing functions favor efficient approximate computation over less-efficient exact computations.

1.1 Events

WaterSlide operates on discrete events that represent information about what is happening at a particular time, a detectable condition, or a state change. Primarily, a WaterSlide event is a collection of metadata that includes timestamps, condition attributes, and other related information. Typically, data about an event is grouped into a metadata container called a *tuple* that contains datatypes and labels on each element of the event record.

1.2 Data

All event data in WaterSlide consists of three elements: a datatype (and its associated functions), a buffer to hold the data, and a reference counter to track "copies" of the data. Optionally, each data element may have a set of labels to describe the data and a set of references to dependent data relationships. Note that multiple labels can be assigned to a single data item.

1.3 Tuples

A common method for grouping metadata about an event is to use a data structure called a tuple. A WaterSlide tuple is a collection of discrete data elements (or *members*) that can be of different datatypes and have different attributes and labels. Additionally, the tuple (or *container*) itself may have its own attributes and labels. In many event processing architectures, data collections are statically defined such that the structure and members of a datatype are defined once using a strict data specification. Although WaterSlide allows for static collections, the preferred method of tracking event data is using a dynamic tuple mechanism that allows additional data to be easily appended. In WaterSlide, a single tuple can contain an unlimited number data element members each with up to 20 different labels.

1.4 Data Labels

WaterSlide takes an "Everything is Miscellaneous" approach to data labeling where each data element can be assigned up to 20 different labels. This is useful in annotating information based on context (e.g., direction, role, type), provenance, and prior knowledge of data properties.

1.5 Key Features

A number of features distinguish WaterSlide from other stream processing frameworks, including the following:

- dynamic datatypes
- unified output
- stateful analysis of events
- use of any data element as a key for hashing
- distributed processing
- rapid prototyping environment

2 Getting Started with WaterSlide

WaterSlide is run by launching the main executable called `waterslide` (or `waterslide-parallel`). The user specifies a processing graph or configuration file as input to `waterslide` to launch WaterSlide processing.

2.1 Basic WaterSlide Build

The WaterSlide package is distributed as either a compressed tarball (typically named `waterslide\_<datetag>.tar.bzip2`) or via a git repository. Set up your build environment based on your distribution environment.

```
# extract the tarball
tar xvfj waterslide_<datetag>.tar.bzip2

# or clone the git repository
git clone ssh://<server>/<path-to-repository>
```

A directory named `waterslide` (or `waterslide-core`, depending on the version) will be created in your current directory.

WaterSlide relies on the following libraries that must be installed in the build environment:

- `bison`
- `flex`

After determining the appropriate variables for your build environment, compile the `waterslide` executable. Below is the default compilation:

```
# build the package
cd waterslide
make -j
```

Note: In most situations, the `-j` option should be used to take advantage of multi-threaded compilation in order to complete the building process more efficiently. If there are resource limitations, the number of threads can be limited via a parameter (e.g., `-j 8`).

Note: In cases where your build environment already has protobuf libraries, you will need to add the `waterslide/bin` directory to the `$PATH` environment variable before compiling (see the next step). This will ensure that the waterslide protobuf libraries are properly linked during compilation.

When compilation is complete, the following executables will be placed in the `waterslide/bin` directory, with symbolic links in the `waterslide` directory:

- `waterslide`
- `waterslide-parallel`
- `wsman`
- `wsalias`

The WaterSlide tools can be directly invoked within the `waterslide` directory. For more flexibility, add the `waterslide/bin` directory to the execution `$PATH` shell variable. Running the script below will add the `waterslide/bin` directory to your path:

```
source mpsetup.sh
```

2.2 Installation

Once the WaterSlide tools have been built, they can be installed into a central location for general use:

```
cd waterslide
sudo make install      # must be privileged user
```

Note: If `make install` is invoked by an unprivileged user, the environment will be "installed" into the user's `$(HOME)/local/waterslide` directory.

The WaterSlide files and libraries will be copied into the central location `/usr/local/waterslide`. If desired, the WaterSlide tools can be linked to enable general access to WaterSlide functionality:

```
ln -s /usr/local/waterslide/bin/* /usr/local/bin
```

For use in a different environment, WaterSlide can be installed into a location of your choosing by setting the environment variable `WS_INSTALL=<destination_path>` before invoking `make install`. Each user would need to add `<destination_path>/bin` to their executable `$PATH` in order to easily invoke the tools.

2.3 Behind the Scenes

The build process for the WaterSlide code base consists of a series of Makefiles in various subdirectories to create the WaterSlide components. To provide a consistent set of parameters, there is a make directive file `src/Makefile.common` that is included by each Makefile. The default build process starts at the top level directory and invokes the build process twice: first, to create the `waterslide` executable and support files for use in serial processing, and second, to create the `waterslide-parallel` environment.

By default, the build output is succinct. To display the specific compilation commands:

```
export WS_VERBOSEBUILD=1
```

A number of kids that depend on non-standard components are only built by setting specific environment variables (e.g., `HASMAGIC=1`) during compilation.

```
make -j HASMAGIC=1
```

View `src/procs/Makefile` or the output of the make process for details of the various build options.

The directory structure for the installed `waterslide` environment is:

- `waterslide/bin`: compiled executables (e.g., waterslide, waterslide-parallel, wsman)
- `waterslide/lib`: compiled libraries and special datatypes
- `waterslide/procs`: compiled kids

This formulaic structure allows the executables to find the support files by searching relative to the executable's location at runtime. If some components are stored in a non-standard location, that can be specified by setting environment variables including `WATERSLIDE_PROC_PATH`, `WATERSLIDE_ALIAS_PATH`, and/or `WATERSLIDE_CONFIG_PATH`.

2.4 Cleaning or Uninstalling WaterSlide

If changes are made to the core WaterSlide source code, you may need to "clean" your `waterslide` build before recompiling the code base:

```
make clean
```

The compiled WaterSlide libraries (not including the `waterslide/protobuflib/lib` libraries) and binaries will be deleted. In some extreme cases, you may also need to rebuild the `protobuf` libraries:

```
make scour
```

Finally, the WaterSlide tools can also be uninstalled (see Section 2.2):

```
sudo make uninstall
```

In this case, the WaterSlide files and libraries will be deleted from the central location `/usr/local/waterslide` (or `$(HOME)/local/waterslide`, if run by an unprivileged user).

Uninstalling WaterSlide does not delete the WaterSlide binaries or libraries that were built in the `$(HOME)/local/waterslide` directory. Use `make clean` or `make scour` as described above.

2.5 Documentation

WaterSlide documentation can be found in the following locations:

- Build/installation process: `waterslide/README`
- Guide for WaterSlide users: this document (`waterslide/doc/users_guide/`)
- Guide for WaterSlide kid developers: `waterslide/doc/developer/`

Additionally, each kid has command-line help that you can access with the `wsman` tool. This program is similar to man pages for Unix commands.

```
wsman -h        # documentation for wsman
wsman           # list of all kids
wsman print     # documentation for print kid
wsman -v print  # verbose documentation
wsman -t input  # tag search
wsman -s smtp   # string search
```

3 The WaterSlide Processing Graph Language

To run WaterSlide, users have to specify a processing graph to acquire data and apply operations to streaming metadata. The processing graph is a way of describing the stages of processing data from source to output. It is like a script or recipe. The processing graph is either specified on the command line or using a configuration file. This processing graph typically consists of pipelines of processing function chains. You can think of this processing graph as a query on the data stream that selects interesting features for computation. In WaterSlide specific processing functions or operations are called "kids". In the language there are ways to combine streaming data inputs and outputs into a complete graph including feedback loops. The processing graph must first consist of at least one input source (csv_in for example)

Here is an example of the syntax:

```
#the following is a config file for waterslide (use with the -F
   option)
csv_in FIELD1 FIELD2 -> $cin ; $cin | match -R FOO -> $match
$match | tuplehash -L PAIR FIELD1 FIELD2 -> $pair
$pair | uniq PAIR -> $upair

#print out metadata
$upair | subtuple FIELD2 FIELD1 | print -TV
```

3.1 Syntax

The WaterSlide processing graph syntax adheres to the following set of rules and is similar in style to Unix shell pipelining..

processor - a processor "kid" function with command line arguments

| - a data pipe. Data output from the left side of the pipe is used as input for the right side function

-> - a stream variable assignment

$foo - a stream variable. The stream variable is used in assignment of output from a command pipeline or as input to a processor. When used in an output assignment, the stream variable must be the last element in the command pipeline. When used to specify an input to a function, the variable or list of variables must be the first element in the command pipeline. It is possible to specify multiple stream variables to be used as input to a processing function. It is also possible to assign the output of multiple command pipelines into one stream variable.

; - used to separate pipeline commands

 # and // - a comment in a configuration file. Everything on the same line after the symbol is a comment.

$foo:PORT - a stream variable assigned to a specific input port of a processing function

$foo.LABEL - a stream variable that only contains events that have the specified data label. In the case of a tuple, the label must be assigned to the tuple container and does not check tuple member labels.

$foo.LABEL:PORT - a stream variable that contains the specified label is assigned to a specific input port of a processing function

PORT:processor - specifies a non-default port of a processor "kid" function

#include *filename* when specified in a config file, this include line will load additional graph information from another file

%thread(num) { ... } - Specify pipelines to run inside a particular thread

**%func fname($foo, $bar −> $baz) { ... }** - Define a function named "fname", which takes in two stream variables ($foo and $bar), and produces one output stream variable

%fname($alpha, $beta −> $gama) - Call a function "fname" and use $alpha and $beta stream variables as its input. Send its output to $gamma

3.1.1 Functions

In an effort to reduce code duplication (especially in the case of multi-threaded environments), simplistic function capabilities are available in WaterSlide.

To define a function in a processing graph, use the following syntax:

```
%func FunctionName($in1, $in2 -> $out1, $out2) {
   $in1 | label ALPHA -> $out1
   $in2 | label BETA -> $out2
}
```

This defines a new function, named "FunctionName", which expects to take in two stream variables, and produce two stream variables. When this function is called, the variables will be replaced with whatever the user specifies in the calling line. The input variables are listed before the '—' specifier, and the output variables are listed after the '—'. The function should send the output of some pipeline to each of the output variables, and read from each of the input variables.

To call a function, use the following syntax:

```
source_keygen -> $alpha
source_keygen -> $beta

%FunctionName($alpha, $beta -> $delta, $gamma)

$delta | bw
$gamma | noop
```

This will call "FunctionName", sending $alpha and $beta as inputs to the function, and provide $delta and $gamma as new stream variables which can be used as a source for future pipelines.

3.2 Additional `waterslide` Options for Debugging

The waterslide executable has other options that are useful in debugging processing elements and running waterslide in quiet environments. These include:

-F file load a processing graph from file (can be called multiple times)
-V turn on verbose debugging
-VV turn on very verbose debugging
-D path specify a path for datatype shared libraries
-P path specify a path for proc kid shared libraries
-G filename save processing graph as a graphviz file
-L filename redirect stderr messages to file (/dev/null is common for this option)

4 Using WaterSlide

4.1 Basic Input & Output

In order to get data into WaterSlide, at least one data source processor or "kid" must be specified in the processing graph. WaterSlide has several useful input "kid" processors including:

csv_in - a source for event tuples based on character-delimited data
file_in - a source for reading in files as memmap-ed binary buffers
mpproto_in - a source for reading in metadata in Protocol Buffer formats

4.1.1 csv_in input kid

The csv_in kid provides a mechanism to read in data from comma separated files (other delimiters can be specified). The data from each row of the CSV file is then bundled into a tuple. The labels on each field are specified in the command line arguments for csv_in.

Here is an example that reads in a list of csv files and prints out only the unique 2nd field:

```
find /data/*.csv | sort | waterslide "csv_in FIELD1 FIELD2 FIELD3 |
 subtuple FIELD2 | uniq FIELD2 | print -VV"
```

4.1.2 csv_in UDP input

The *csv_in* kid can listen on a UDP port for input when the -U option is specified. It performs the same parsing operations as the CSV module. Here is an example of running WaterSlide while listening to to sources, a pcap source and a udp source:

```
#read in csv while listening for other data from a udp port
find /data/*.csv | sort | waterslide "csv_in | bandwidth ;
 csv_in -U 3333 | print -VV"
```

```
#now in another terminal
# you can run netcat to connect to the udp server by issuing
nc -u localhost 3333

#if text is typed in the netcat console, it will printed by
   waterslide
```

4.2 print processor

One of the most useful processing "kids" is the print processor. The print processor takes any data as input and generates a human readable output for display on a terminal or to print to a file. By default, the print kid prints each event and separates the data fields by a semicolon. There is a verbose option that allows for both the data and its associated labels.

The print processor also can print out only the ascii portion of strings or print data to a file.

4.3 subtuple processor

There are times when only specific data found in a tuple needs to be selected. The subtuple kid can be used to select only data with specified labels. It also orders the data using the user-specified ordering. Here is an example of using subtuple

This operation actually generates a new tuple that is a subset of the original tuple. If no tuple members are specified, the tuple is duplicated and a copy of every member is placed in the new tuple. This is useful in very complex processing graphs where some processing needs to append to a tuple and other processing does not, so a separate tuple structure is needed. ======= This operation actually generates a new tuple that is a subset of the original tuple. If no tuple members are specified, the tuple is duplicated and a copy of every member is placed in the new tuple. This is useful in very complex processing graphs where some processing needs to append to a tuple and other processing does not, so a separate tuple structure is needed.

```
#the following graph duplicates a tuple and appends to it
csv_in FIELD1 FIELD2 FIELD3 | subtuple FIELD2 FIELD3 -L TUP1 ->
   $tuple1
$tuple1 | subtuple -L TUP2 -> $t2

$t2 | match -R FOO -> $tuple2
$tuple1, $tuple2 | print -V
```

4.3.1 Flattening Nested Tuples

The subtuple kid can be used to flatten nested tuples (tuples within tuples) to produce a new tuple with the selected elements from nested structures. Elements of a nested tuples can be accessed by specifying a parent label and then a child label. For instance if a tuple has members $< FOO, BAR >$

and BAR is a tuple with members $< A, B >$, then you access member A or B by specifying BAR.A and BAR.B respectively.

4.4 Manipulating Tuples

In addition to subtuple, other kids for manipulating tuples exist. These include

removefromtuple - specifies items by label in a tuple to be removed from a tuple. The emitted tuple will not contain the specified items

tuplehash - generates a hash based on specific items in a tuple. This is used to combine data from different data fields into a single key for state tracking algorithms.

splittuple - splits tuple into multiple tuples based on labels. If multiple items exist in a tuple, it will create a tuple for each item. It is also possible to specify data to keep/replicate for each new tuple.

mergetuple - merges data from multiple tuples based on a shared, specified key

appendfirstitem - appends item to tuple based on key, useful for inferencing and merging

appendlast - appends last item to tuple based on key, useful for inferencing and merging

4.5 Counting things

A number of kids have been created to count and analyze data. Most of these tools expect the data to exist in a dynamic tuple data structure where the data to be counted can be identified by a unique label. These tools include:

bandwidth - tracks items per second, counts all items. Reports output at end of stream

keycount - counts number of items with a specific key value. For instance the number of events with name ABCD

keyadd - adds values of a given labeled data field and accumulates these sums based on key

keyaverage - computes the average value of a data field based on sums of the same key

heavyhitters - adds values of a given labeled data field and key, keeps track of the top N keys

labelstat - counts the occurrence of each label within tuples

4.5.1 Heavy Hitters

The heavyhitters kid implements an Space-Saving approximate counting algorithm [3] that keeps the top N elements with the greatest counts. Counts can either be incremental or additive.

```
#select the top 10 FIELD1 based on event count
waterslide "csv_in FIELD1 | heavyhitters FIELD1 | print -V"

#selects the top 10 FIELD1 based on length field FIELD2
waterslide "csv_in FIELD1 FIELD2 |
heavyhitters FIELD1 -V FIELD2 | print -V"
```

4.6 Numeric Calculations

If members of a tuple are numeric values (that can be cast as a double), the *calc* kid can be used to perform calculations on these features and assign results to a new labeled data element in the tuple. *calc* can also be used to assign labels to tuples and tuple members.

4.6.1 Options

There are several command line options you can use to change calc"s behavior.

-N - no pass through. This prevents the processor from passing tuples down the pipeline. This is mostly useful when you are using local keyed variables and you only really care about the final values of those counts.

-M x - hashtable size. Each local keyed variable uses its own hashtable. This option sets the size of the hashtable. When the number of entries exceeds that size, the hashtable will expire entries, flushing them out to the pipeline.

-R - keep only the key. This option will cause local keyed variables to only keep the tuple member containing the key of interest rather than the whole tuple as a representative.

4.6.2 Variables in Calc

Calc currently support three kinds of variables: tuple members, local variables, and local keyed variables.

Tuple members simply refer to numerical members of tuples indicated by labels. These variables are referred to by tuple member label. On the left hand side of an assignment, they create a new labeled tuple member. On the right hand side of an assignment or in an expression, they retrieve the value from the tuple. If the label does not exist or if it cannot be cast into a double, the value returned is zero. If more than one member has the same label, the first one encountered is used. So, for example, the following script:

```
calc 'RESULTLEN=LEN1-LEN2;'
```

retrieves the tuple members labeled LEN1 and LEN2 and subtracts one from the other. The difference is assigned to a new member appended to the tuple and is labeled RESULTLEN.

Local variables are variables that are kept local to the processor and persist between tuples. Local variables are indicated by a hash "#" at the beginning of the variable name. So, for example, the following script would simply count the number of tuples that have been sent through the processor:

```
calc '#TUPLECOUNT++;'
```

Local variables are not added to tuples. To add the count so far to each tuple, you'd have to do the following.

```
calc '#TUPLECOUNT++; TUPLESSOFAR=#TUPLECOUNT;'
```

Local keyed variables are also variables that are kept local to the processor. However, they keep a unique value for each key provided. The key's label is indicated by square brackets after the variables. The variable also has to be preceded by a hash mark. So, for instance, the following script keeps count of the number of times a certain NAME has been seen.

```
calc '#NAMECOUNT[NAME]++;'
```

This will keep a separate count for each value of NAME that it encounters. When flushed, all these counts are pushed into the pipeline, one tuple for each unique NAME encountered. Each tuple will contain a member with the variable name (NAMECOUNT) and a representative tuple or key from what it counted.

Note that these values are kept in expiring hashtables. By default, each table gets approximately 10,000 entries. This value can be modified with command line options.

Local windowed variables (both keyed and unkeyed) are variables that are kept local to the processor. Unlike any of the other variables, however, these variables implements rolling windows over the data. Using the enqueue function, you specify what data is to be added to the window and the maximum size of the window. The oldest values are dropped off and only the top n most recent values are kept. Windowed variables can be both keyed and unkeyed.

There are several functions that are used with windowed variables:

enqueue(varname, window_size, expression value to add) - this function adds a value to the windowed variable and makes sure that any old values are dropped if necessary. The variable name can optionally specify a key in which case different windows are kept for each unique key.

qavg(varname), qcount(varname), qsum(varname) - these return the average, the number of entries, or the sum of the values in the specified window.

Note that if a windowed variables is referred to outside of the context of one of the above functions, it returns the most recently added value.

```
calc 'enqueue(#window[NAME], 15, LEN);
   AVGWINLEN=qavg(#window[NAME]);'
```

This will keep a rolling window for each unique NAME that contains the 15 most recently observed values for LEN. It will also append the average LEN for the last 15 seen values to the tuple.

4.6.3 Operations in Calc

The following functions are available to compute values: "+ - / *", sin, cos, atan, ln, exp sqrt, abs, trunc, floor, ceil, and round.

Also, for variable assignment, "++" and "–" can be used to increment and decrement by one. "+=" and "-=" can also be used to increment and decrement by other values. Note that when

tuple member labels are on the left hand side of these operators, they might not work as you expect. See the variables subsection for more information.

4.6.4 Conditionals in Calc

Calc also supports if-then-else style conditionals. The format is:

`if` *expr* `then` *statement_list* `else` *statement_list* `endif;`

The *else statement_list* is optional. The `endif` is not.

expr can be any expression that evaluates to a numerical value. If the absolute value of the resulting numerical value is less than 0.0001, then the expression is considered false. Otherwise it is considered true. The following operators can be used to build an *expr*:

- logical AND (&&), logical OR (||), and logical NOT (!) connectives. Logical operators evaluate to 1.0 (true) or 0.0 (false).
- numeric comparators (>, <, ≥, ≤, and ==). Note that these are operators like + and -, returning numerical values. Those values are 0 for false, and non zero for true.
- the built-in function *exists* or *EXISTS* can be used to test for the existence of a label in the tuple. These functions return 1.0 if the label exists, 0.0 otherwise.

Examples of Conditional Expressions:

- The following example will add a tuple member labeled A to the tuple. A will be 5 or 6, depending on the value of CSTAT_LENGTH:

```
calc 'if CSTAT_LENGTH>50 then A=5; else A=6; endif;'
```

- The following compares two label values and adds the tuple member with label INTERESTING to the tuple:

```
calc 'if (#COUNT[NAME] > 100 || MYLABEL <= 20) then
   INTERESTING=1; endif;'
```

- The following tests for the existence (or non-existence) of two labels (see below for details on #PASS):

```
calc ' if (exists(MY_LABEL) && !exists(MY_OTHER_LABEL))
       then #PASS=1; else #PASS=0; endif;'
```

4.6.5 Conditional Passthrough

The -N option for calc can be used to specify whether or not a tuple will pass through calc. However, it is also possible to use the local variable #PASS to conditionally pass through tuples. #PASS will cause a tuple to pass through if it is set to something other than 0 and prevent the tuple from passing through with any other value. The following two examples will both do the same thing, passing the tuple through if CSTAT_LENGTH is greater than 50.

```
calc 'if CSTAT_LENGTH>50 then #PASS=1; else #PASS=0; endif;'
```

```
calc '#PASS=CSTAT_LENGTH>50;'
```

Note - Assigning #PASS anywhere in a script causes calc to ignore the presence or absence of the -N flag and depend solely on the value of #PASS to determine whether or not to pass a tuple. #PASS is a local variable that maintains its value between tuples. So once you set #PASS, that will be the settings used for any tuple until you change the value again. The initial value of #PASS will be zero.

4.6.6 Assigning Labels

Calc can also be used to assign labels to tuples and tuple members. The function is "label" and takes two forms. The simples form simply adds a label to the tuple. For example:

```
calc 'label(MY_NEW_LABEL);'
```

will add the container label "MY_NEW_LABEL" to every tuple. To assign the label to a tuple member rather than the tuple itself, just specify an existing label on the tuple member. For example:

```
calc 'label(MY_NEW_LABEL, MY_OLD_LABEL);'
```

will add the label "MY_NEW_LABEL" to every tuple member containing the label "MY_OLD_LABEL."

Neither of these alone is terribly useful. However, combined with conditionals, you can conditionally assign labels based on specified criteria. For example:

```
calc 'if AVAL>3 then label(AVAL, BIG_NUMBER); endif;'
```

will add the label "BIG_NUMBER" to a tuple member named AVAL if the value of that tuple member is greater than three.

4.6.7 Examples

The following script counts the number of times each unique SERVERIP is encountered.

```
calc -N '#COUNT{SERVERIP}++;'
```

The following script computes the average CONTENT_LENGTH per SERVERIP. This will flush COUNT, SUM, and AVG to the pipeline rather than just AVG. That's probably not what is the most useful and will be fixed in a future version of calc.

```
calc -N '#COUNT[SERVERIP]++; #SUM[SERVERIP]+=CONTENT_LENGTH;
#AVG[SERVERIP]=#SUM[SERVERIP]/#COUNT[SERVERIP];
```

4.7 Filtering items

One of the powerful features of WaterSlide is the ability to filter and select items of interest. In terms of filters, here are some useful kid processors:

uniq - determines if labeled items are new. Multiple ports allows for removal and queries against the state of each labeled key.
uniqexpire - determines if items are new, expires out old items based on timestamp
firstn - selects the first N tuples from each key
bloom - determines if items are new, keep track of items using a bloom filter
sample - samples items based on probability
cntquery - determine if the value of a keys is mostly positive. This module has multiple ports, one for INCREMENTing, one for DECREMENTing, one for querying. This is for finding items sequence conditions that are biased.

4.8 Matching

When dealing with datatypes, one might want to do data specific searches such as string matching and integer comparisons. The following kid processors provide datatype specific ways to match data:

match - finds strings in character buffers that match a dictionary of strings at arbitrary offset locations
fixedmatch - finds strings in character buffer that match a dictionary of strings at fixed, explicit locations in the character buffer
match_uint - matches integers with specified properties and/or numeric ranges
equal - checks to see if two elements in a tuple are equal
haslabel - checks to see if a label or set of labels exists in an event or members of a tuple
re2 - when WS is compiled with Google's re2 library, this allows you to specify perl compatible regular expressions including extracting content

4.9 State Tracking

State tracking is a core element of WaterSlide. With a state tracking system, one needs a key for which to track. Fortunately any labeled data can serve as a key in WaterSlide. In addition, a combination of data can be hashed together to generate a key for use in tracking complex state. Here are some descriptions of state tracking kids:

uniq - only passes one uniq items. Items are seleted by label.
appendfirstitem - stores first item (value) at a key, subsequent queries of the key results in the stored value appended to the query tuple.
appendlast - stores last item (value) at a key, subsequent queries of the key results in the stored value appended to the query tuple
keyflow - creates a temporal hash based on keys that occur close in time.
keeplast - stores the last data item at the key. Only outputs data when table is full, or the table is flushed

cntquery - determines of state of key has reached a counting threshold. Uses ports to increment or decrement state
firstn - passes the first n events for each key
keepn - keeps the last n items at the key
stateclimb - user specifies labels and values (positive or negative). when label is found the value is added to accumulator for each key. If accumulator reaches a target value, then output is triggered.

4.9.1 Flushing State

In most cases, the hash tables in WaterSlide do not emit any state until they receive a flush event. There are two types of flush events: (1) exit flush and (2) inline flush. When a processing graph completes, or is interrupted (e.g., by pressing Ctrl+c), an exit flush is issued to each of the kids (that have a registered flusher) in the processing graph; each kid then processes the flush event based on its internal functions. Some issue processing statistics, some flush internal tables, and others exit silently. In contrast, certain kids (e.g., `flush`) issue inline flushes to the processor immediately downstream, causing that kid to flush internal state tracking structures. In most cases, inline flushes are not forwarded beyond the kid immediately downstream from the flush generator.

For example, the `flush` kid issues an inline flush event to the downstream processor to clear any stored state from internal tables controlled by that kid. Flush events can be issued based on the number of event seen or according to a time delta.

The inline flush events generated by the `flush` kid are received and processed by the `heavyhitters`; they are not forwarded any further downstream.

4.10 Decoding

A number of kids have been written for the purpose of taking a buffer and extracting out protocol subelements or by taking a buffer and transforming it into another buffer (such as decompression). Here are some descriptions of such kids:

base64 - takes an identified base64 encoded buffer and decodes it into another buffer
re2 - regular expression searching in buffers, can extract out data using expressions
asciihex - takes an ascii string with hex values and converts them to binary

5 Nested Tuples within WaterSlide

WaterSlide supports the creation of nested tuples. Not all the WaterSlide kids know how to search for elements in a nested tuple. To specify an element within a nested tuple, you can use a *dot* notation when specifying a member (example: LABEL1.LABEL2) .

5.1 HTML output

Information from tuples can be displayed in an html page using the *print2html* kid. This kid can format output string such that any javascript content or active content is not executed by the web browser. Not executing content is important when examining malicious content strings. For each tuple you can have the tool allow for a drill-down link to another web page. This can be useful when integrating WaterSlide output with other web-based analysis applications.

5.2 Multi-Threading

In the latest version of the WATERSLIDE source code, the multithreaded version of WaterSlide is built side-by-side with the serial version. The WaterSlide executable for use with multi-threaded functionality is *waterslide-parallel*.

WaterSlide is designed to take advantage of systems with many cores and can be optimized to place work on specific cores in order to gain performance on asymmetric access to memory or other resources.

To take advantage of threading, users need to specify threads in their WaterSlide graphs.

5.2.1 Explicit Threading

In production scripts it is better to specify explicit core IDs when threads are specified so that threads can be pinned to a given core. Pinning threads provides a performance advantage when processing and allows the user to place threads in ideal locations on many-core processors.

Specifying a thread separation (configuration file only):

```
%thread(1) {
source_keygen -> $keys
}
%thread(2) {
$keys | match -R 123 KEY | print -V
}
```

In this example the first pipeline is pinned to thread 1 and the second pipeline is pinned to thread 2. The waterslide-parallel executable allows the user to specify an offset (-T option) to be added to their configuration thread ids so that threads can be pinned to offset cores.

Currently there are not any performance measuring tools that can assist users in thread separation. If a thread is not keeping up it will result in the prior thread blocking. Thus if a source is not able to keep up with their workload, it is an indication that some thread is not keeping up.

5.2.2 Performance

On certain system architectures, executing on consecutive CPUs (e.g., 0, 1, 2, 3) results in significantly worse performance when compared to executing on every other CPU (e.g., 1, 3, 5, 7). This

impact to performance is caused by data exchanges performed between the shared work queues crossing CPU socket boundaries. Unless extremes are taken to bundle data into larger "chunks", it is very expensive to cross from one socket to another. If there is significant data exchange in your graph, performance is enhanced by exchanging between cores on the same processor package.

It is best to set your threads to be all odd or all even since most systems alternate sockets as the CPU count is incremented by one. However, check your system to determine your specific architecture's thread-to-socket mapping:

```
cd /sys/devices/system/cpu
grep "" cpu*/topology/* -H
```

You will find information like the following for each CPU:

```
cpu7/topology/core_id:3
cpu7/topology/core_siblings:aaaaaa
cpu7/topology/core_siblings_list:1,3,5,7,9,11,13,15,17,19,21,23
cpu7/topology/physical_package_id:1
cpu7/topology/thread_siblings:0000a0
cpu7/topology/thread_siblings_list:5,7
```

This information tells us that "`cpu7`" is actually hyperthreaded with `cpu5`, and they are sharing the same arithmetic logic unit (ALU). In addition, there are 6 real ALUs on this processor package (id=1). If you want to move data between threads, you will see better performance by, for instance, pinning one thread to `cpu7` and the other to `cpu9`.

Of course, if the system has a single socket, concerns about crossing socket boundaries are moot, and the performance should be very similar if the graph is run on "adjacent" threads (e.g., 0, 1, 2, 3) or alternate threads (e.g., 1, 3, 5, 7). There is nothing special about running threaded WaterSlide on consecutive CPUs. However, adjacent threads make it easier to (1) locate which particular CPUs your graph is running on and (2) segregate different jobs from multiple users. For example, a graph requiring three threads can be run on CPUs 0, 1, and 2 (the default) and another user or instance of the same graph can run on CPUs 3, 4, and 5 (with the "-T 3" command-line option of `waterslide-parallel`).

5.3 Distributed Processing

WaterSlide also has capabilities for transporting data across a network socket. Kids that aid in transporting data across sockets include the *tcpthrow* and *tcpcatch*.

6 Conclusion

WaterSlide is an event-at-a-time architecture for processing metadata. It is designed to take in a set of streaming events from multiple sources and process them through a set of modules and return meaningful outputs. WaterSlide enables building of composeable pipelines for processing complex

events in high-throughput streaming environments while maintaining enough state-tracking to efficiently answer complex queries at multiple levels of detail, from full stream summaries down to details in individual events.

This document is intended to serve as an introductory manual for beginning to intermediate usage of the WATERSLIDE framework. With over 80 individual processors available, an exhaustive users guide is not feasible or desirable. Therefore, this document is meant to cover a number of the more useful processors at a reasonable level to enable someone picking up the tool to have a reasonable chance at developing a set of pipelines that will answer a variety of questions about events occurring in the streaming environment. Consistent use of the tool, along with further reading of the documentation available with each of the kids is necessary for developing further confidence and capability with the WATERSLIDE tool.

References

[1] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[2] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36:1–34, March 2004.

[3] Ahmed Metwally, Divyakant Agrawal, and Amr Abbadi. Efficient computation of frequent and top-k elements in data streams. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer Berlin / Heidelberg, 2005.

[4] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anandsudhakar Kesari. S4: Distributed stream computing platform. *KDCloud*, December 2010.