**WebdamLog Manual**

**Preface**

This document is intended for anyone who is looking to work or use the WebdamLog framework in a local/distributed environment on the DB group cluster.

**Version History**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Added by | Note | Date |
| 1.0 | Vera Zaychik  Diwakar Sharma | Initial version | Sep-03-2014 |
| 2.0 | Vera Zaychik | Updated for waltz, added section on graph generation | Jul-10-2015 |
|  |  |  |  |

**Target audience**

1.WebdamLog Research Group

2. New Users

3. Anyone who wants to execute tests using WebdamLog framework.

**Index**

1 Introduction 3

2 System setup for waltz 7

2.1 Setting up the local git Repo 7

2.2 Setting up the local svn Repo 7

2.3 Group Permissions 7

2.4 Using Database on waltz (for results) 7

3 Setting up the cluster – required programs and modules 8

3.1 Checking Python and Ruby versions 8

3.2 Installing missing libraries 8

3.3 Applying the message-loss bud patch 11

3.4 Setting up the host machine for use 12

4 Running scenarios 13

5 Fetching the data from database & generate plots 14

5.1 Database Schema 14

5.2 Database Queries 16

5.3 Building graphs from data 17

6 Maintenance 19

6.1 Maintaining Git repo 19

6.2 Maintaining svn repo 19

# Introduction

This documents describes the setup for Webdamlog experiments.

We use the following terminology:

Scenario: this is a setup of peers, data, and rules.  For example, an instance of the master-aggregator-follower with 1 master, 3 aggregators, each with 3 followers is a scenario.

Execution: this is a run of a scenario in which the peers run their programs, transmit data, and timing information is recorded.

Tick: a tick is a time step in the execution, but in terms of data it is a collection of statistics recorded at a time step for a particular peer in an execution of a scenario.

A. Overview of servers

**waltz.cs.drexel.edu**: this runs the mysql server and hosts the svn repository. This also starts the execution. It sends results to mysql and connects to the cluster nodes to start node executions.

**{master,slave01, slave02… slave16}.cs.drexel.edu**: these are the cluster nodes.  They are the hosts that peers run on.  They are only reachable from waltz.cs.drexel.edu.  Before execution, each of these nodes needs to have the (1) the latest source code and (2) the latest scenario files.

B. Version control

We use two version controlled repositories in order to simplify the transfer of files across remote machines:

1) Source code:

This holds the webdamlog source code, the java code for scenarios, and the python code for the execution.

URL: <https://github.com/vzaychik/webdamlog-engine>

2) Execution parameters and results:

In order to not pollute the source code repository with tons of updates from the executions, a local svn repository served from waltz is used for the scenario files.  It is accessible to all the cluster nodes.

URL: svn://waltz/var/svn/webdamlog-exp

It stores a few things:

webdamlog-exp/MAF        ← master aggregator follower scenarios

webdamlog-exp/PA        ← photo album scenarios

The MAF and PA directories hold scenarios, which are named with unique timestamps based on their creation time.  When, e.g. scenario 1386295711376 is created, its directory will contain the following:

the file netAddr.txt file, one directory “out\_[hostname]\_timestamp” for each host, and “1386295711376.pckl”.  The out\* directories contain the actual scenario files that datagen generates.

- When scenario 1386295711376 is executed, the results of a single execution are collected into a subdirectory such as MAF/1386295711376/exec\_1386423810107/ which contains a .pckl file and a subdirectory /bench\_files.  The latter has all the individual output files generated by different peers during the execution.  Those are files like “benchark\_time\_log\_aggregator1\_2013-12-07 08/43/36 -0500” which contain individual tick counts.  Note that the “benchmark\_\*” files are created at different hosts during an execution.  Each host runs svn add and commit so that the repository will contain the benchmark files from every peer and every host.

The .pckl files are serialized output of either Scenario objects or Execution objects.  These store all the information that is populated in the mysql database in the corresponding tables. As a consequence, the mysql database records can be completely reconstructed from the contents of the webdamlog-exp repository.

C. The python code:

The experimental framework is written in python. The framework loads the experiment configuration from the exp.cfg file, generates experiment files if they don’t already exist, runs each scenario the specified number of times, and loads the results into the mysql database.

1. wdlexec.py: main experimental driver. No arguments required.

2. models.py: this contains a description of the schema used for mysql. These fields can’t be changed without dropping the database tables in mysql and rebuilding, or modifying the mysql schema manually.  Also note the mysql configuration parameters are hardcoded here.

3. loadBenchmark.py: this contains the parsing function that reads benchmark\_\* files.  Also contains the function “refreshFromFileSystem” that traverses the file structure of webdamlog-exp and inserts new tuples into the database.  It looks up keys and won’t add a record that already exists.

4. scenario.py: generates a single scenario by calling datagen.

5. execution.py:  this contains the function “executeScenario(...)” that executes a scenario across hosts and records the output.  It calls webdamlog’s “run\_access\_remote” ruby program.  Run this file to actually execute scenarios.  But you need to know the scenIDs you want to run, which isn’t ideal.

The execution of scenarios just generates output to the filesystem.  Adding it to the database is a separate operation in loadBenchmark.

6. fab.py: this uses the fabric python library which makes it easy to use ssh to execute functions remotely.

**BUILDS**

*Build 3*

The scenario end condition is included based on the number of expected results. This build does not account for message loss patch and sometimes packets were lost for large scenario (10K etc.) or it will result in broken pipe leading to test failure. All data for this build has now been removed.

*Build 4*

Writeable information is not exchanged by peers on startup but instead loaded from a writeable.wdm file generated by datagen. This build also takes care of message-loss patch but the timing issues were present as master shuts down before receiving all the data from all the peers resulting in some peers running forever ultimately resulting in failure. Also the database schema is modified here to incorporate modes directly into execution database.

*Build 5*

This build takes care of the network issues caused earlier which resulted in master not getting all the data. This is the build which was used to generate the majority of results. Also changed how memory size was computed from using “words” to bytes.

*Build 6*

In build 5 it was discovered that very long messages from peer to peer result in extra tick processing. Build 6 corrects this.

*Build 7*

First version of formulas optimization. Very slow.

*Build 8*

Rewrote basic access control and both optimizations to do direct lookup on the acl relation instead of a join. Rewrote formulas optimization to avoid circular references. Removed all unnecessary rules and relations.

*Build 9*

Rewrote formulas optimization to get rid of extra relations and compute extended formulas on as-needed basis.

*Build 10*

Rewrote to keep Grant and Read tuples in separate relations for efficiency. Omega string changed from “All peers” to “**Ω**“. Changed the photo album experiment rule to avoid conditional on selection, which is slow in bud.

*Build 15*

The final SIGMOD submission build, which includes improvements to send time (calculate send buffer only if there is something new to send), using bud collections properly, reordering conditionals in bud rules to the front for efficiency, etc.

# System setup for waltz

In order to run the tests, a connection to remote network “waltz.cs.drexel.edu” is required which hosts all the python and ruby files required to use the WebdamLog framework. The account access can be provided by Vera Zaychik (zaychik@drexel.edu). Once the access has been provided, the user can ssh into the system. There is a special webdam user account already setup for the repeatability and other webdam experimental work. For that account, all setup steps in this section can be skipped as they have already been performed.

## Setting up the local git Repo

Once the user is logged into the system, s/he needs to set up the local git repo from github. In order to setup the local version of git repo, use the git clone command:

git clone https://github.com/vzaychik/webdamlog-engine.git

This will update a local git repo under current directory (/home/username/welbdamlog-engine)

If using the webdam username, this has already been set up.

## Setting up the local svn Repo

Similarly, set up a local svn repo in the parent directory using the below commands.

svn checkout (co) url:

*svn co file:///var/svn/webdamlog-exp*

Once the svn and git repository are setup, you can see the folders namely,

webdamlog-engine (for git repo)

webdamlog-exp (for svn repo)

## Group Permissions

The user should belong to the svngrp to be able to check out the repository.

## Using Database on waltz (for results)

Enter the MySQL command as shown below with given credentials

mysql –-user=webdam –-password=ilovedb

After entering the command, use webdamlog database.

# Setting up the cluster – required programs and modules

If using the webdam user account, skip this section.

The minimum version supported for python is 2.6. The minimum supported version used while writing the framework is Ruby 2.1.

Additionally, Ruby environment, specifically garbage collection, must be tuned as by default it’s tuned for very small scripts. Add these lines to the .bashrc file:

export RUBY\_GC\_HEAP\_INIT\_SLOTS=1000000

export RUBY\_HEAP\_SLOTS\_INCREMENT=1000000

export RUBY\_HEAP\_SLOTS\_GROWTH\_FACTOR=1

export RUBY\_GC\_MALLOC\_LIMIT=1000000000

export RUBY\_HEAP\_FREE\_MIN=500000

export RUBY\_HEAP\_MIN\_SLOTS=800000

## Checking Python and Ruby versions

Run commands ‘ruby –-version’ and ‘python –version’ and verify that correct versions are installed. If ruby is not installed, the easiest way is install is through rvm (see rvm page for instructions: <http://rvm.io/> )

Webdamlog requires several ruby gems which are listed below (some of these are installed with ruby by default):

* bud
* polyglot
* treetop
* ruby\_deep\_clone

Additionally several python modules needs to be present in order to run the executions: fabric and peewee. Both are installed in waltz site-wide.

Fabric – to run ssh commands from the python scripts

Peewee – to create DB schema and update database from python scripts

Configparser – to read in the config file for the plot generator

Matplotlib – to generate plots of results from the database

## Installing missing libraries

The below steps shows how to install the ruby gems and python libraries which are crucial in using the framework.

***Installing Gems locally into user home directory***

If the user does not have sudo privileges, then the gems can be installed locally into the user home directory.

To install gems into the same location as the ruby install, use

gem install gem\_name

This might result in error if the user does not have sudo permissions and the installed ruby is not local. In order to install the gem locally :

gem install --user-install gem\_name

This will install the gem to the local path. Finally, user has to add this value of path “/home/path/to/gems/bin” to PATH, if it does not already exists and then add it to bashrc. For eg. The ruby gem path is “/home/username/.gem/ruby/1.9.1/bin”

export PATH=$PATH:/home/username/.gem/ruby/1.9.1/bin

NOTE: In order to view the gem env to check the path type the below commands

-sh-4.1$ gem env

RubyGems Environment:

- RUBYGEMS VERSION: 2.2.0

- RUBY VERSION: 2.1.0 (2013-12-25 patchlevel 0) [x86\_64-linux]

- INSTALLATION DIRECTORY: /share/apps/ruby-2.1.0/lib/ruby/gems/2.1.0

- RUBY EXECUTABLE: /share/apps/ruby-2.1.0/bin/ruby

- EXECUTABLE DIRECTORY: /share/apps/ruby-2.1.0/bin

- SPEC CACHE DIRECTORY: /nfs/avid/users1/diwakar02/.gem/specs

- RUBYGEMS PLATFORMS:

- ruby

- x86\_64-linux

- GEM PATHS:

- /share/apps/ruby-2.1.0/lib/ruby/gems/2.1.0

- /nfs/avid/users1/diwakar02/.gem/ruby/2.1.0

- GEM CONFIGURATION:

- :update\_sources => true

- :verbose => true

- :backtrace => false

- :bulk\_threshold => 1000

- REMOTE SOURCES:

- https://rubygems.org/

- SHELL PATH:

- /share/apps/ruby-2.1.0/bin

- /nfs/avid/users1/diwakar02/.gem/ruby/2.1.0

- /usr/lib64/qt-3.3/bin

- /usr/pgsql-9.3/bin

- /usr/local/bin

- /bin

- /usr/bin

- /usr/local/sbin

- /usr/sbin

- /sbin

- /opt/eclipse

- /opt/ganglia/bin

- /opt/ganglia/sbin

- /usr/java/latest/bin

- /opt/maven/bin

- /opt/pdsh/bin

- /opt/rocks/bin

- /opt/rocks/sbin

-sh-4.1$

Installing the bud gem first will automatically install all missing gems that bud requires, so it is recommended to install in this order: bud, polyglot, treetop, ruby\_deep\_clone.

Finally, the LANG environment variable should be set to US.UTF-8. For most users, this is the case by default. To verify, use the “locale” command on console and check the LANG variable.

If the user receives the error as above, it can be corrected either by adding following in the environment.rb file

-sh-4.1$ locale

locale: Cannot set LC\_CTYPE to default locale: No such file or directory

locale: Cannot set LC\_ALL to default locale: No such file or directory

LANG=en\_US.iso885915

LC\_CTYPE=UTF-8

LC\_NUMERIC="en\_US.iso885915"

LC\_TIME="en\_US.iso885915"

LC\_COLLATE="en\_US.iso885915"

LC\_MONETARY="en\_US.iso885915"

LC\_MESSAGES="en\_US.iso885915"

LC\_PAPER="en\_US.iso885915"

LC\_NAME="en\_US.iso885915"

LC\_ADDRESS="en\_US.iso885915"

LC\_TELEPHONE="en\_US.iso885915"

LC\_MEASUREMENT="en\_US.iso885915"

LC\_IDENTIFICATION="en\_US.iso885915"

LC\_ALL=

-sh-4.1$

Encoding.default\_external = Encoding::UTF\_8

Encoding.default\_internal = Encoding::UTF\_8

***OR***

export LC\_ALL="en\_US.UTF-8"

to your ~/.profile, ~/.bash\_profile or similar.

***OR***

export LANG="en\_US.UTF-8"

export LC\_ALL="en\_US.UTF-8"

export LC\_CTYPE=en\_US.UTF-8

to .bashrc under HOME directory.

To test whether Ruby and all gems are installed and configured correctly for Webdamlog, run a simple Webdamlog test: ruby tc\_access\_basic.rb in test/40\_access\_control directory.

***Installing Python Libraries locally***

The required python modules can be installed using one of two methods.

Method A:

Use pip, i.e. ‘pip install peewee’ or ‘pip install –-user peewee’

If pip is not installed, it can be installed following the instructions on the pip webpage: <https://pip.pypa.io/en/latest/installing.html>

Method B:

The general procedure is to override the module's default installation location (usually /usr) and configure your $PYTHONPATH variable to include the new path as well as the standard path. (This applies to Python packages which are installed with "python setup.py install". C-based packages which use "./configure" can be prepared similarly)

a) (This step is only necessary if the installer complains about the install location not supporting .PTH files)

Add to your PYTHONPATH:

export PYTHONPATH=$PYTHONPATH:/home/<your username here>/lib64/python2.6/site-packages/

b) Download the source package, e.g.

curl –O URL of your module

and unpack

c) Install with a new prefix, for example:

python setup.py install --prefix=/home/<your username>

d) If not already done, do step a). This will need to be set every time you want to use this module, so it makes sense to add this line to your .profile

e) test the installation:

python -c "import modulename"

This should complete without an error message.

## Applying the message-loss bud patch

The message-loss patch is required so that the messages are not lost over the network, which caused hindrance earlier. This patch is available under the github repository and is named as “message-loss.patch”. In order to use this patch, following commands need to be performed.

1. Go to the location where bud gem is installed. If it is local, it is probably in $HOME/.gem/ruby/2.1.0/gems/bud-0.9.7
2. Go to the lib folder and run (assuming git repo is in $HOME/webdam/webdamlog-engine)

patch –p6 < ~/webdam/webdamlog-engine/message-loss.patch

This should patch collections.rb, server.rb, and bud.rb

## Setting up the host machine for use

*What is the need for setting up host machines?*

As the webdamlog framework is intended to utilize multiple hosts to work in a distributed environment by assigning the peers onto different hosts so that they can run in parallel over the network. And for this reason special directories needs to be created in the local space under each machine. One of the requirements of the program is to work on multiple hosts across the environments and in order to utilize these hosts, following steps need to be performed.

**Step 1**: Generate the rsa keys for your system. Restrict yourself with id\_rsa.pub and do not share private key over any network.

Use the command:

ssh-keygen

*NOTE*: The name should be same for both public and private key otherwise it will result in an error.

**Step 2 (optional)**: Copy the key to the host if it does not share the home directory:

ssh-copy-id username@machinename

**Step 3**: Once you are logged into system, ssh into any host. It should not prompt you for the password.

**Step 4**: Create a user directory in a locally-mounted drive:

**Step 5**: Clone the git repo and svn repo from sec 1.1 and 1.2 respectively in partition2.

**Step 6**: Logout from host and repeat the above steps for other hosts.

# Running scenarios

**Step 1:** On waltz, setup the scenario in exp.cfg by setting the parameters for the number of followers, aggregators, policy, scenario, number of facts, etc. Each scenario will execute each desired mode (accessControl flag) the number of times indicated by numRuns flag.

Photo Album:

1. For the photo album scenario, set the scenarioType to PA.
2. Set the rootPath to the directory within which webdamlog-engine directory is located. Note: for the webdam user, leave this value blank.
3. Set the networkFile list to the list of facebook network files that you want to run. The files are located in the webdamlog-engine/network directory and are named using the facebook-u<number of peers>-i<connectivity>.txt scheme. The current version of exp.cfg lists all the files that were included in the SIGMOD paper experiment.
4. Set the policy to PUB, KNOWN, or “PUB KNOWN” to specify which policies to run, public or known or both respectively.
5. Leave the other parameters under the scenarioPA section as they are, unless you want to experiment with different database sizes.
6. Set the numRuns to the number of repetitions of each scenario. 5 repetitions were used in the paper and averaged.
7. Set the accessControl to the list of modes you want to test. The full list is “000 001 011 101 111” but for the networks greater than about 80-90 peers, it is recommended to only run “000 011 111” because modes with optimization 1 take exponentially longer as the network size increases.

Master-Aggregators-Followers:

1. For the MAF scenario, set the scenarioType to MAF.
2. Set the numFollowers to the number of followers, numAggregators to the number of aggregators, and the aggPerFollower to the apf value desired. Multiple values can be put in a list, i.e. for an experiment on the topology, can set aggPerFollower to “1 2 3”.
3. Set the policy to PUB, KNOWN, or “PUB KNOWN” to specify which policies to run, public or known or both respectively.
4. Set the numFacts to the desired number of facts each follower starts with. A space-separated list can be used to experiment with an increasing database size with the set network configuration.
5. Set the ruleScenario to JOIN\_OF\_UNIONS, UNION\_OF\_JOINS, or both, depending on which variant you want to experiment with.
6. Set the numPeersPerHost to the correct value depending on the number of total machines listed under hosts and the total number of peers. The master peer is allocated a host to itself, aggregators share hosts if numPeersPerHost value is greater than 1, as do followers, but followers and aggregators do not mix on a single host. For example, if numFollowers=20, numAggregators=2, and there are 16 hosts listed, then numPeersPerHost can be set to 2 and 4 hosts will not get any peers.
7. Set the numRuns to the number of repetitions of each scenario. 5 repetitions were used in the paper and averaged.
8. Set the accessControl to the list of modes you want to test. The full list is “000 001 011 101 111”.

**Step 2:** Run the wdlexec python script

python wdlexec.py > <outputfile> 2>&1 &

This will redirect all output to a file and run in background. Running in the background is recommended because the execution of the full experiment takes many hours – depending on the parameters and the number of variations to run, an experiment may take a day or more.

Every few hours verify that the experiment is still executing successfully by either looking over the outputfile or grepping for errors in it.

Occasionally, an error “Unexpected Bud error: #<EventMachine::ConnectionNotBound: received ConnectionUnbound for an unknown signature” may happen. This can be ignored as it does not impact the execution and happens on shutdown.

Occasionally, a Segmentation error happens on shutdown due to some race condition bug in the eventmachine library. This does not affect the experimental results. The result files from this execution are still checked into svn and valid. To check whether this happened:

grep Segmentation <outputfiles>

**Step 3:** When the desired experimental runs are complete, add them to the database using the loadBenchmark python script. The numeric argument is the earliest timestamp which to examine.

svn up

python loadBenchmark.py 0

Only executions not already in the database will be added, so 0 argument works, but can take a long time if there are a lot of scenarios and executions.

# Fetching the data from database & generate plots

## Database Schema

The webdamlog database consists of various tables listed below:-

mysql> show tables;

+--------------------------------+

| Tables\_in\_webdamlog |

+--------------------------------+

| execution |

| scenario |

| tick |

+--------------------------------+

3 rows in set (0.01 sec)

mysql>

1. **Scenario table** – Scenario table consists of the scenarios generated using the Julia’s java program under webdamlog-engine/datagen. The entire schema of Scenario is shown below:

***scenID*** *= BigIntegerField(primary\_key=True)*

***scenType*** *= CharField(null=True) # MAF or PA*

***numFollowers*** *= IntegerField(null=True) # numFollowers - number of peers at the lowest layer*

***numAggregators*** *= IntegerField(null=True) # numAggregators - number of aggregators (middle layer)*

***aggPerFollower*** *= IntegerField(null=True) # aggregatorsPerFollower - degree of follower nodes*

***policy*** *= CharField(null=True) # policy - one of PUBLIC, PRIVATE, KNOWN*

***numFacts*** *= IntegerField(null=True) # numFacts - number of facts per extensional relation on a follower peer.*

***ruleScenario*** *= CharField(null=True) # scenario - one of UNION\_OF\_JOINS and JOIN\_OF\_UNIONS*

***valRange*** *= IntegerField(null=True) #facts at follower peers are drawn randomly from the interval [0, valRange)*

***numExtraCols*** *= IntegerField(null=True) #number additional of non-key columns*

***numHosts*** *= IntegerField(null=True) # number of hosts*

***hosts*** *= CharField(null=True) # optional argument; name of the file (on the local system) that lists names or IP addresses of the instances, one name or IP address per line*

***numPeersPerHost*** *= IntegerField(null=True)*

***networkFile*** *= CharField(null=True)*

1. **Execution Table** – The execution table consists of the executions along with execID, mode etc. The entire schema of execution table is shown below:

***execID*** *= BigIntegerField(primary\_key=True) – execution id*

***scenID*** *= ForeignKeyField(Scenario) – the value of scenario id from the scenario tables which maps to the execution table*

***timeToRun*** *= FloatField() – the time master node takes (generally 60 sec)*

***mode*** *= IntegerField() – ( 0 – for no access, 1 – Access ON, 3 – Optim 1)*

***runTime*** *= FloatField() – time taken in completing the execution*

***success*** *= BooleanField() – (1- true, 0- false)*

***build*** *= IntegerField() – ( build number , currently tested for build number 5)*

1. **Tick table** – The tick table contains the tickcount and ticktime information for various peers namely master, aggregators and peers. The entire schema for tick table is shown below:

***execID =*** *ForeignKeyField(Execution)*

***peerName =*** *CharField(null=True) # description of peer that is benchmarked*

***fileName =*** *CharField(null=True) # filename containing benchmarking of ticks*

***tickIndex =*** *IntegerField(null=True) # index of the tick*

***tickTime1*** *= FloatField(null=True) # first tick processing*

***tickTime2 =*** *FloatField(null=True) # read from channel, rewrite strata, do wiring*

***tickTime3 =*** *FloatField(null=True) # bootstrap or invalidate tables as necessary*

***tickTime4 =*** *FloatField(null=True) # main processing (running until fixpoint)*

***tickTime5 =*** *FloatField(null=True) # write on channel*

***tickTime6 =*** *FloatField(null=True) # end of tick*

***tickCount1*** *= IntegerField(null=True) # number of tuples across all peer non-system collections*

***tickCount2*** *= IntegerField(null=True) # number of "words"*

***tickCount3*** *= IntegerField(null=True) # number of collections this peer is**delegating to other peers (intermediary relations)*

***tickCount4 =*** *IntegerField(null=True) # number of rules this peer is delegating to other peers*

***tickCount5*** *= IntegerField(null=True) # number of tuples this peer is sending to other peers*

## Database Queries

Some queries have been written to fetch the data from the mysql database. If the plots are desired instead, skip to the next section.

**Query1**: Query for getting the total time and Fix point time

select F.facts, F.policy, F.scenario, F.network, F.mode, F.numFollower, F.numAgg, F.APF, AVG(F.Difft4t3), AVG(F.Time6), count(F.Difft4t3) from (select s.scenID, s.numFollowers as numFollower, s.numAggregators as numAgg, s.aggPerFollower as APF, s.policy as policy, s.networkFile as network, s.numFacts as Facts, s.valRange as valRange, s.ruleScenario as scenario, s.numHosts, s.numPeersPerHost, e.execID, e.mode as mode, sum(t.tickTime4 - t.tickTime3) as Difft4t3, sum(t.tickTime6) as Time6, t.tickCount1 as c1 from scenario as s, execution as e, tick as t, (select execID\_id, peername, max(id) as id from tick where peerName like 'sue%' group by execID\_id, peerName) as MT where t.execID\_id=e.execID and t.peerName like 'sue%' and e.execID=MT.execID\_id and t.peerName=MT.peerName and t.id<MT.id and s.scenID=e.scenID\_id and s.scenType='PA' and e.build=15 group by e.execID, s.numFacts, t.peerName) as F group by F.Facts, F.policy, F.scenario, F.network, F.mode, F.numFollower, F.numAgg, F.APF;

This will output all the data for the peers (not the master sue) in the Photo Album scenario. Substitute both occurrences of ‘peer%’ with ‘sue%’ to get the results for peer sue instead.

For the MAF scenario:

select F.facts, F.policy, F.scenario, F.network, F.mode, F.numFollower, F.numAgg, F.APF, AVG(F.Difft4t3), AVG(F.Time6), count(F.Difft4t3) from (select s.scenID, s.numFollowers as numFollower, s.numAggregators as numAgg, s.aggPerFollower as APF, s.policy as policy, s.networkFile as network, s.numFacts as Facts, s.valRange as valRange, s.ruleScenario as scenario, s.numHosts, s.numPeersPerHost, e.execID, e.mode as mode, sum(t.tickTime4 - t.tickTime3) as Difft4t3, sum(t.tickTime6) as Time6, t.tickCount1 as c1 from scenario as s, execution as e, tick as t, (select execID\_id, peername, max(id) as id from tick where peerName like 'fol%' group by execID\_id, peerName) as MT where t.execID\_id=e.execID and t.peerName like 'fol%' and e.execID=MT.execID\_id and t.peerName=MT.peerName and t.id<MT.id and s.scenID=e.scenID\_id and s.scenType='MAF' and e.build=14 group by e.execID, s.numFacts, t.peerName) as F group by F.Facts, F.policy, F.scenario, F.network, F.mode, F.numFollower, F.numAgg, F.APF;

This will output all the data for the follower peers in the MAF scenario. Substitute both occurrences of ‘fol%’ with ‘master%’ or ‘agg%’ to get the data for the master or aggregator peers, respectively.

**Query 2:** Query for getting space data ( tickcount2 and tickcount5)

SELECT J.numFacts, J.scenario, J.policy, J.numFollower, J.numAgg, J.APF, J.mode, SUM(J.sumC2) as Words, SUM(J.sumC5) as Network FROM (Select F.numFacts as numFacts, F.scenario as scenario, F.policy as policy, F.mode as mode, F.EID as ID, F.PNAME, SUM(F.C2) as sumC2, F.C5 as sumC5, F.numFollower as numFollower, F.numAgg as numAgg, F.APF as APF FROM ( Select s.numFacts, s.policy as policy, s.ruleScenario as scenario, e.mode as mode, s.numFollowers as numFollower, s.numAggregators as numAgg, s.aggPerFollower as APF, t.execID\_id as EID, t.peerName as PNAME, max(t.tickCount2) as C2, SUM(t.tickCount5) as C5 FROM scenario as s, tick as t, execution as e WHERE t.execID\_id = e.execID AND s.scenID = e.scenID\_id AND t.peerName like 'agg%' and e.build = 15 GROUP BY s.numFacts, s.ruleScenario, s.policy, t.execID\_id, t.peerName ) as F GROUP BY F.numFacts, F.scenario, F.policy, F.mode, F.EID, F.PNAME, F.numFollower, F.numAgg, F.APF ) as J GROUP BY J.numFacts, J.scenario, J.policy, J.mode, J.numFollower, J.numAgg, J.APF;

## Building graphs from data

Once the experiments have completed, the results can be plotted using two options.

*Option 1*: Use the queries in the section above, save the results to a csv file, load the csv into excel and after performing data wrangling, generate graphs.

*Option 2*: Preferred option. Use the graph generator in the webdamlog-engine/python directory that connects to the database and generates a file for each graph specified in the config file.

To run:

python create-graphs.py <config> <savedir>

The checked-in graphs-gen.config file contains the configuration to generate all the plots that were used in the paper. These can be modified or other graphs added. The format is a line-separated list of queries to be plotted, where each query is represented by the following values on individual lines:

* [query-number] such as [query-1]
* type: at the moment only “line” type is supposed fully.
* name: the name for the file to which this plot will be saved
* xlabel: the name for the x axis label
* xlabel: the name for the y axis label
* sql: the SQL query that generates the series to be plotted. The series are generated based on the columns, while the first column values are used as x axis values.

# Maintenance

Many a times user will be required to update the git and pull the latest code changes into local repository. Below steps come handy while performing the same.

## Maintaining Git repo

Some useful commands are available in git, which can be seen from the git documentation, however for project readiness, important commands are listed below:

1. git checkout master ( checking out the master branch, however not recommended unless some code needs to be pushed)
2. git pull ( in order to pull the latest code changes into local copy)
3. git add filename ( adding a file to be committed)
4. git status ( shows the status of files added, committed, untracked etc.)
5. git commit –m “message” ( commits the file added)
6. git push ( adds the code into the github)
7. To merge: follow following steps
8. git checkout master
9. git pull
10. git checkout branch ( you are working on)
11. git merge master
12. git commit
13. git push

## Maintaining svn repo

Similar to git, there comes occasion when svn needs to be updated which can be seen at svn website. However, in terms of project, many a times some files and folders need to be deleted which can be done in this manner

svn rm folder name/filename

svn commit –m “your custom message” folder name/filename