# TriggerRatesTools ‘Manual’

Marc-André Dufour
McGill University

July 29, 2008

# Contents

# 1 Introduction

This document aims at making the study of trigger rates in athena a little easier.

Approximating trigger rates are essential when designing a trigger menu and to choosing appropriate signatures that will maximize your acceptance of events while staying within bandwidth limitations. This tool is designed to make that kind of study easier.

TriggerRateTools originated as a combination of scripts that would make it much easier to run the trigger in athena and extract the trigger efficiencies. Due to increasing interest, this collection of tools was made more generic and more robust, and TriggerRateTools was born. With time, TriggerRateTools grew and started to include more features, like the overlap matrix (lists the overlap between any combination of two signatures), and support for prescales. This would have never been possible without without the interest and support of McGill University's ATLAS group, so I would like to thank all those that contributed to the development of the tools.

# 2 How TriggerRateTools works

## 2.1 The goal

The trigger software for ATLAS evolves very quickly, and it is now stable enough for users to start doing some 'trigger aware' analyzes. However, as advanced as the trigger software is, it is also a very complicated system that new users need to understand very well in order to obtain meaningful results in such an analysis. TriggerRateTools was intended to greatly simplify this process, and to allow a user with minimal knowledge of the trigger system to rapidly get rate estimates for some trigger menu.

TriggerRateTools is designed so that results can be quickly updated as well. Although the tool is capable of outputting results directly in the athena output, in a separate text file, or in a ROOT ntuple, it is partially the fact that it is an object-oriented tool that makes is easy to use while generating a very large set of results.

Although TriggerRateTools is easy to integrate to existing jobOptions, the tool also comes with its own set of scripts that allow for easy batch submission. This system can work with Torque, Condor, LxBatch, CAF, etc. through 'plugins'. The job submission can also be disabled to allow the user to run interactively with the scripts (see subsection 4.3 for details about configuration).

The tool is designed to be able to run over an RDO input, by re-running the full trigger simulation, but also to run over AODs with or without re-running the hypothesis algorithms (see diagram 1). Re-running over AODs is, however, still untested at the date this document is being written.

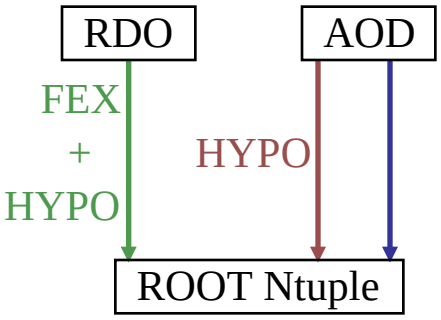


Figure 1: Different ways to run TriggerRateTools in Athena

## 2.2 The approach

Previously, TriggerRateTools was divided in 3 distinct steps that one needed to follow to obtain the rates. Since that time, the trigger, as well as the tool have matured. Hence, the three steps were merged into a single one, and TriggerRateTools is now an Athena algorithm.

The menu configuration is directly obtained from TrigDecisionTool while running an athena job, which makes the whole system significantly easier to use than it was. Although prescales need to be configured in order to be retrieved, they can also be ignored directly by the steering. That feature essential to TriggerRateTools since it will approximate the effect of prescales based on probability (see section 6.1) for details. This allows us to maximize use of statistics. Obtaining similar results using a random number generator to compute the effect of prescales would require us to run many thousand times over the same sample and averaging over the results.

TriggerRateTools was designed to be able to obtain the individual trigger rates for the various items composing the trigger, but its uniqueness resides in its ability to approximate rates for combinations of logically 'or'ed triggers. The menu is after all just a list of trigger chains 'or'ed together. Since the tool supports this feature, it is straightforward to compute as many lists as a user wants. Hence, we can study multiple menus simultaneously so long as their components are all initially defined in the original trigger code.

The tool computes rates, but also unique rates, independent fractions, and overlaps between various combinations of two triggers. Storing all that information is simple enough, but making it in such a way that it is easily retrievable isn't. Therefore, it was decided to make TriggerRateTools object-oriented. The objects used are called '(single) signatures' and 'complex signatures' which correspond respectively to the items and chains defined in the original trigger code, and the groups of triggers whose decision is approximated based on probabilities of its components. Hence, the L1 menu for instance, has its results stored in a complex signature. These objects are usable in both ROOT and athena, and their definition is in the package `Trigger/TrigAnalysis/TrigNtDecision`. TrigNtDecision is therefore required for TriggerRateTools to run in athena, or to access the information stored in TriggerRateTools' output ntuple directly in ROOT.

# 3 Toolkit Structure

## 3.1 File hierarchy

In the main package directory, you will find seven folders:

- `cmt/`: Contains the package's `setup.sh`, which is sourced to setup the package. Sourcing the `setup.sh` will set the environment variable `TriggerRateTools_PATH` which is required to run the batch submission scripts
- `config/`: Contains all the configurations, XSLT stylesheets, and batch system plugins (see section 3.1.1)
- `doc/`: Contains all the latest documentation pertaining to the package
- `examples/`: Contains sample analysis scripts that have been written. The ROOT macro to combine the results from several jobs is also in this directory
- `share/`: Contains python jobOptions that can be included in larger jobOptions
- `src/`: Contains the C++ source code of TriggerRateTools and EvtVetoTool
- `steps/`: Contains one sub-folder per step to be run for generating the rates.
- `TriggerRateTools/`: Contains the C++ header files of the tool and its helping tools
- `utilities`: Contains additional smaller tools that can be used to help in the configuration / operation of TriggerRateTools

### 3.1.1 `config/`

- `config_files/`: Contains the bash scripts that will export sample-specific variables that other scripts will be requiring.
- `plugins/`: Contains all the plugins that can be used by the tool. Currently these are only plugins to support various batch systems.
- `xslt/`: Contains sample XSLT files to format XML files such that they can viewed in a web browser.

- config_common.sh: bash script that exports the variables that are non-sample-specific. This is also where the job submission command can be configured[1]
- root_classes: Deprecated
- job.sh: generic script that is submitted to the queue, executes athena
- thresholds.dat: Deprecated
- thresholds_dummy.dat: Deprecated

#### 3.1.2 examples/

- CombineJobs.C: very generic script that is used to combine the output ntuples of multiple jobs from a single sample. The script will automatically combine all of the .root files that are in the same directory and create a file called TriggerRates.root. In order to work, TrigNtDecision must be checked-out and compiled.
- CombineSamples.C: very generic script that is used to combine multiple samples together. Each ROOT file should be from a sample that is not overlapping with any other it will be comined with. The script will automatically combine all of the .root files that are in the same directory and create a file called TriggerRates.root. In order to work, TrigNtDecision must be checked-out and compiled.
- ROOT_to_XML.C: very generic script that will take a file called TriggerRates.root and output an XML file called TriggerRates.xml that contains most of the quantities computed by the tool. In order to work, TrigNtDecision must be checked-out and compiled.
- Signal_Eff_vs_Bkgd_Rate.C: Deprecated

#### 3.1.3 utilities/

- get_sequence.sh: helper tool to list_signatures.sh.
- list_signatures.sh: parses the outputHLTsignature.xml and outputHLTsequence.xml files, which are automatically generated when running a job. These files contain the HLT configuration, that is the list of algorithms that should be run for each signature, as well as the name of the trigger element (TE) after each algorithm was run. The tool basiacally lists the TEs and a few more details about the signature in a readable format. Run the script without any arguments to learn how to use it. (Note that this tool is not optimized for speed)
- trigger_XML_config_parser.py: python script that uses three arguments: L1 xml configuration file, HLT xml configuration file, and output file name. The script will generate the list of trigger active, along with their cumulative prescales.

# 4 Configuration

## 4.1 JobOptions configuration

Although TriggerRateTools has its own job submission system to help the user, the tool can simply be added to any existing jobOptions. The basic configuration only requires the user to add two lines in their jobOptions,

```
include("TriggerRateTools/TriggerRateTools_common.py")
ServiceMgr.THistSvc.Output += ["TriggerRateTools DATAFILE='TriggerRates.root' OPT='RECREATE'"]
```

Hence, a sample jobOptions file that can be used to find some trigger rates for the 'default' menu would be

[1] Read the comments carefully when changing the job submission command.

```
doTrigger = True
TriggerModernConfig = True

doWriteAOD=False
doWriteESD=False
doWriteTAG=False
doAOD=False
doESD=False
doTAG=False
doCBNT=False
CBNTAthenaAware=False
useROOTNtuple=True

OutputLevel=3

EvtMax=50

PoolRDOInput = ["/scratch/heracles/dufourma/test_sample/dijets_J8/dijets_J8.RDO.pool.root"]
DetDescrVersion="ATLAS-CSC-01-02-00"

include("RecExCommon/RecExCommon_flags.py")
TriggerFlags.readHLTconfigFromXML=False
TriggerFlags.readLVL1configFromXML=False
TriggerFlags.Slices_all_setOff()
TriggerFlags.doLVL1=True
TriggerFlags.doLVL2=True
TriggerFlags.doEF=True
TriggerFlags.triggerMenuSetup = 'default'

include("RecExCommon/RecExCommon_topOptions.py")

include("TriggerRateTools/TriggerRateTools_common.py")
ServiceMgr.THistSvc.Output += ["TriggerRateTools DATAFILE='TriggerRates.root' OPT='RECREATE'"]
```

There are several options that can be set directly in the jobOptions to allow the user to further configure TriggerRateTools (see section 4.1.1).

### 4.1.1 Optional jobOptions configurations

There are several options that allow the user to get rates information that would normally require several modifications to the menu without having to modify the actual menu they are running. After including TriggerRateTools' common jobOptions, the following can be set in the main jobOptions as well

- `triggerRateTools.doTextOutput = False`:

  This option, if turned on, will output a file called rates.out in the run directory that contains a limited amount of information concerning the individual triggers in the menu and their rates.

- `triggerRateTools.doVeto = False`:

  This option, if turned on, will allow the user to apply a veto on the events to use in the rates calculations, using TriggerRateTools' AlgTool called EvtVetoTool (see section 7).

- `triggerRateTools.doRawTD = False`:

  This option, if turned on, will dump the raw TriggerDecision information in the generated Ntuple that contains the rates. The result dumped it the result of "`isPassedRaw()||isPassThrough()`" where the prescales are ignored because of an option set in the steering.

- `triggerRateTools.OutputLevel = 2`:

  This is the standard way of setting the output level of the tool.

- `triggerRateTools.xSection = 1.0 #barn`:

  Use this feature to set the cross-section corresponding to a specific sample directly in the jobOptions.

- `triggerRateTools.Luminosity = 1.0 #barn^1 sec^1`:

  Use this feature to set the luminsity to use directly in the jobOptions.

- `triggerRateTools.IgnoreList += []`:

  Simply list the trigger elements to ignore in the rates calculations, and it will be as though they were never part of the menu. To ignore a chain, the corresponding trigger elements should be listed for the L1, L2 and EF.
  e.g. `triggerRateTools.IgnoreList += ["L2_J50","EF_J50","L2_e10","EF_e10"]`

- `triggerRateTools.CplxAndList += []`:

  TriggerRateTools is able to emulate the logical "AND" of multiple triggers together. The syntax to be used is `[[name, level, item1, item2, ...],[name, level, ...]]`, where the name is the name that the emulated "AND" trigger will have, the level is the trigger level of the items it combines, and the items are the trigger item names. Note that there can be as many items as we wish in the list, but each item should only appear once. These emulated triggers are not counted towards the overall level rates, but overlaps will single trigger and other complex triggers (emulated "AND" or "OR") are computed automatically.
  e.g. `triggerRateTools.CplxAndList += [["Cplx_And_1","EF","EF_3J60","EF_3J25"], ["Cplx_And_2","EF","EF_3J180","EF_3J60","EF_3J25"]]`

- `triggerRateTools.CplxOrList += []`:

  TriggerRateTools is able to emulate the logical "OR" of multiple triggers together. The syntax used is the same as an emulated "AND" trigger. These emulated triggers are not counted towards the overall level rates, but overlaps will single trigger and other complex triggers (emulated "AND" or "OR") are computed automatically.
  e.g. `triggerRateTools.CplxOrList += [["Cplx_Or_1","EF","EF_3J60","EF_3J25"], ["Cplx_Or_2","EF","EF_3J180","EF_3J60","EF_3J25"]]`

- `triggerRateTools.PrescaleOverrideList += []`:

  In the event that a prescale was not properly set in the configuration, it is possible to override it with this list. The syntax is `[item,prescale]`, note that both the item name and the prescale need to be placed in quotation marks to be parsed properly.
  e.g. `triggerRateTools.PrescaleOverrideList += [["L2_g20","3"],["L2_g15","5"]]`

- `triggerRateTools.MenusList = []`:

  Since a menu is simple the logical "OR" of all the triggers composing it, TriggerRateTools also has the ability to emulate "Menus" in a very special way. The menus are configured here and not in the complex Or trigger because in this list, it is possible to add an additional factor by which to multiply the prescale for each element in the menu. This does not affect in any way the other rates generated by the tool, but it is possible using this method to test different sets of prescales for a single menu in a single run. For instance, an item with an initial prescale of 20 can use and additional prescale factor of 0.05 to yield an effective prescale of 1, or a factor of 2.5 to yield an effective prescale of 50. The syntax for this configuration is `[[name,level,item:factor,item:factor], [name,level,etc]]`.
  e.g. `triggerRateTools.MenusList += [["Menu1","L2","L2_g20:10","L2_g15:10"], ["Menu2","L2","L2_g20:1","L2_g10:1"], ["Menu3","L2","L2_g20:5","L2_g10:1"]]`

- `triggerRateTools.StreamsList = []`:

  The Streams lists is essentially the same as the menus list, but is kept separate to allow for more streams-specific features that can be foreseen. The configuration syntax is the same as for the MenusList.

## 4.2 Batch submission configuration

In order to set the main environment variables, you should source the setup script in the cmt directory of the package (e.g. `source cmt/setup.sh`).

### 4.2.1 config_common.sh

This configuration file contains the variables that will be used for all the jobs that you submit; it is automatically sourced when running a `steps/step?/step?.sh` script.

- `LUMINOSITY` : (float) luminosity that will be used to calculate the rates, in barns$^{-}1$ s$^{-1}$.
- `MAIN_OUTPUT_PATH` : (string) the main output path. Subdirectories will be created for each step of the process. (e.g. `"/raid/gaia/users/dufourma/batchJobs"`)
- `CONFIG_PATH` : (string) the path to the directory containing all the configuration files. (Automatically set, **DO NOT EDIT**)
- `SCRIPT_PATH`: (string) path to the steps directory. (Automatically set, **DO NOT EDIT**)
- `FOLDER` : (string) subfolder of the output step directory that will contain the output of all the jobs submitted while running the script. This variable is automatically generated to be the date and time at which the step script was run, in order to avoid any overwritting of previously generated results. The `PREFIX` is also automatically concatenated with this string to form the actual folder name.
- `RUN_DIR` : (string) the path to your 'run' directory (or the folder in which you would like to run your job). Note that subdirectories will be created for each job in order to avoid conflicts. Those directories are automatically removed at the end of the job, hence this directory can be in the `/tmp` of a computing node for instance. (e.g. `"/raid/gaia/users/dufourma/12.0.5/run"`)
- `PYTHON_DIR` : (string) name of the temporary folder that will be created to hold the python jobOptions that will be created. This folder will be created in the jobs's output directory (see `FOLDER`), and the `PREFIX` will be appended to its name.
- `NUM_RDOS_PER_STEP1` : (int) number of input RDO files to use for each job in step1.
- `NUM_AODS_PER_STEP2` : (int) number of input AOD files to use for each job in step2.
- `NUM_AODS_PER_STEP3` : (int) number of input AOD files to use for each job in step3.
- `COMMANDS[]` : (string array) list of commands that should be run to setup your work environment in athena (and possibly ROOT). The list of commands is executed in order of increasing indices and can be as long as required. It should not contain the ':' character.
- `DO_BATCH` : (bool) set to 'true' to automatically submit jobs to a batch system, or set to 'false' to run interactively.
- `SLEEP` : (int) number of seconds to wait between submitting two jobs (can be used to reduce the load of submitting jobs on servers).
- `BATCH_SYS` : (string) name of the folder in `config/plugins/` that contains the batch system configuration to be used.

### 4.2.2 config/config_files/config*.sh

You will only need to use one of these scripts at a time. Whenever you run a `step*.sh` script, the first argument must be the filename of the sample configuration file that you select. Note that in these files, any variable set in `config_common.sh` (see 4.2.1) can be overwritten.

- `PREFIX` : (string) prefix that will be used to identify all the output files that will be generated. (e.g. `"cern_ttbar"`)

- MAX_NUM_RDOS: (int) the total number of input RDO files that should be used for the analysis
- DET_DESCR : (string) the detector description as used by athena. (e.g. "ATLAS-CSC-01-02-00")
- DATA_PATH : (string) the path to the local folder containing RDOs to be used in step 1. The path can also be overridden at step 1 by providing a path as the second argument to the script step1.sh (see section 5).
- NON_LOCAL_ID : (string) the name of the dataset that should be run over, or the non-local path to the data files (e.g. files on castor). Either this variable or DATA_PATH should be set to an empty string, if both are defined,
- DATA_PATH has precedence.
- USE_FULL_PATH: (boolean) should be true if the NON_LOCAL_ID set should precede any file listed in the jobOptions
- XSECTION : (float) cross-section in barns of the sample that is being used.

### 4.3 Configuring a Batch Submission System

The batch system plugins are located in config/plugins/. Each system is contained in a separate folder, though this can also be used to setup different configurations of the same system. Each folder should contain two files:

- setup.sh : contains the main setup for the system, and is only ran once at the beginning of the step script.
- submit.sh : contains the job specific intructions to submit a job to the batch system, and is ran for each job.

These scripts allow the user to create job description files that can be submitted to more complex systems like Condor or the grid. The environment variables available in these scripts are

- setup.sh
  - OUTPUT_PATH = MAIN_OUTPUT_PATH/step?/PREFIX"_"FOLDER
  - All variables set in config/config_common.sh
  - All variables set in config/config_files/<YOUR_CONFIG_FILE>.sh
- submit.sh
  - All of the ones available for setup.sh
  - CONFIG_FILE: full path to config_common.sh (required by config/job.sh)
  - FILE: full path to the job's jobOptions file (required by config/job.sh)
  - COUNT_LBL: the job counter, formatted appropriately ($1 \rightarrow 001$)
  - JOB_NAME = PREFIX.COUNT_LBL
  - RUN_DIR_SPECIFIC : full path to the job's run directory, a subdirectory of the main RUN_DIR (required by config/job.sh)

If the script requires the listing of files using a special command, based on the dataset name, then this can be set with the LS_CMD variable. If each file needs to be preceded by a special set of characters, (e.g. "dcache:"), then this can be set with the PATH_PREFIX.

## 5 How to run the analysis

### 5.1 Using TriggerRateTools in your jobOptions

In order to use TriggerRateTools in your jobOptions, simply follow the steps listed in section 4.1 and run your job. At the end of your job (in the finalize step for TriggerRateTools), the individual trigger rates are printed to the screen directly. In order to extract the information in the ROOT file that is generated, skip ahead to section 5.3

## 5.2 Using TriggerRateTools' batch submission

In order to use the batch submission, first configure TriggerRateTools as explained in section 4.2. It is then essential to source TriggerRateTools' setup script (`TriggerRateTools/cmt/setup.sh`), which will setup the environment variable `TriggerRateTools_PATH` to its proper value. Once you have selected and edited your sample specific configuration file (say `TriggerRateTools/config/config_files/config_test.sh`), you can execute the following command to launch the job submission

```
$> cd $TriggerRateTools_PATH/steps/step1
$> ./step1.sh config_test.sh
```

The jobs should then be automatically submitted and the output directories, run directories, etc. will be created.

Once all the jobs are done, be sure to source TrigNtDecision's setup script (`TrigNtDecision/cmt/setup.sh`) after compiling it. The compilation of TrigNtDecision will created the library that ROOT will require to read the objects stored in the outputted ntuple by TriggerRateTools, while source the setup script will set the environment variable `TrigNtDecision_PATH` that is used in most of TriggerRateTools' pre-configured macros.

In order to combine the output all the jobs' results, go in your configured output path (`$MAIN_OUTPUT_PATH/step1/*`) and execute

```
$> ./finalize.sh
```

You should now have a file called `TriggerRates.root` in your output directory, which contains the information obtained as if the analysis was run in a single job.

## 5.3 Reading the results from the output ROOT file

**Note:** to use any of the macros you should first source TrigNtDecision's setup script (`TrigNtDecision/cmt/setup.sh`) after compiling it.

You should now have a root file that you generated (usually called `TriggerRates.root`) that contains all the calculated rates information. To unpack the stored information into an XML file, you can use the ROOT macro found in `TriggerRateTools_PATH/examples/ROOT_to_XML.C`. If this XML file is placed in the same directory as one of the XSLT style sheets found under `TriggerRateTools_PATH/config/xslt`, the XML file can be opened in a web browser and all the information will be presented in several table. Most of the data stored in the output is accessed by this script, and it can be read in order to know how to access specific information so as to build your own ROOT macro. Other accessing methods of the objects can be found in section 8.

# 6 Computed quantities

## 6.1 Approximating trigger rates

The first step in approximating trigger rates is to choose the appropriate input dataset. In order to compute the most realistic rates, we must run over the datasets with the largest cross-sections, and hence the most likely to occur in nature. However, these datasets typically contain very few events that satisfy the trigger criteria and hence very large datasets are required in order to reduce the statistical error associated with the rates. In order to design the $L = 10^{31}\ cm^{-2}\ s^{-1}$ menu, only a minimum bias dataset containing non-diffractive events of approximate cross-section of 70 mb was used. For higher luminosities, dijet samples are used to probe the high energy spectrum of the events.

In order to compute trigger rates, the full trigger simulation (level-1, level-2 & event filter) is run on RDO input files. For each level, the individual trigger rates are computed using the equation

$$R = \mathcal{L} \times \frac{n_{accepted}}{n_{total}} \times \frac{1}{\prod_{l=lowest\ level}^{current\ level} P_l}, \tag{1}$$

where $R$ is the rate, $\mathcal{L}$ is the instantaneous luminosity, $P_l$ is the prescale applied at a specific level, $n_{accepted}$ is the number of events accepted after the specific level studied (and hence all the previous lower levels as well), and $n_{total}$ is the total number of events in the dataset.

In order to compute the overall acceptance rate of a specific menu, the overlap in acceptance from different trigger items needs to be correctly taken into account. Over a large data sample, the average probability $Pr_i$ that an event would be accepted by by a trigger item $i$ is given by

$$Pr_i(event) = D_i(event)/P_i, \tag{2}$$

where $D_i$ is the unprescaled decision of accepting an event or not, and $P_i$ is the overall prescale associated with the trigger item. The probability that two triggers accept an event simultaneously is then given by

$$Pr_{12}(evt) = Pr_1(evt) \times Pr_2(evt). \tag{3}$$

The overall probability of accepting an event by a menu of two triggers is thus given by

$$Pr_{menu}(evt) = Pr_1(evt) + Pr_2(evt) - Pr_1(evt) \times Pr_2(evt) \tag{4}$$

This computation, although simple in the case of two triggers, rapidly becomes increasingly complex as the number of items in the menu increases. Fortunately, this problem can be solved recursively. In reality, the decision of accepting an event will always be either true or false, but in order to approximate the average rate over a long run, we use this system of probabilities.

## 6.2 Quick definition of generated quantities

- **Efficiency for signature** $S_1$: the efficiency $\varepsilon$ is the ratio of the number of events that passed the signature $S_1$ ($N_{passed\ S_1}$) to the total number of events that were studied ($N_{total}$).

$$Eff(S_1) = \varepsilon(S_1) = \frac{N_{passed\ S_1}}{N_{total}} \tag{5}$$

- **Rate for signature** $S_1$: the rate $R$ is the product of the $S_1$'s efficiency ($\varepsilon$), of the instantaneous luminosity ($\mathcal{L}$) and of the cross-section ($\sigma$), divided by $S_1$'s cumulative prescale ($P_1$).

$$Rate(S_1) = R(S_1) = \varepsilon(S_1) \times \mathcal{L} \times \sigma \times \frac{1}{P_1} \tag{6}$$

- **Independent number for signature** $S_1$: defined as the sum of the inverse number of signatures (from the same level as $S_1$) that accepted the event ($N_{passed\ level}$). This sum extends over all the events that passed .

$$N_{indep}(S_1) = \sum_{passed\ S_1} \left( \frac{1}{N_{passed\ level}} \right) \tag{7}$$

- **Independent fraction for signature** $S_1$: defined as the normalized independent number, it can be computed as the independent number divided by the the number of events that passed $S_1$ ($N_{passed\ S_1}$)

$$f_{indep}(S_1) = \frac{N_{indep}(S_1)}{N_{passed\ S_1}} = \frac{\sum_{passed\ S_1} \left( \frac{1}{N_{passed\ level}} \right)}{N_{passed\ S_1}} \tag{8}$$

- **Unique fraction for signature** $S_1$: defined as the number of events that only passed $S_1$ ($N_{only\ passed\ S_1}$) over the total number of events that passed $S_1$ ($N_{passed\ S_1}$).

$$f_{unique}(S_1) = \frac{N_{only\ passed\ S_1}}{N_{passed\ S_1}} \tag{9}$$

- **Unique rate for signature** $S_1$: defined as the rate times the unique fraction.

$$R_{unique}(S_1) = f_{unique}(S_1) \times R(S_1) = \frac{N_{only\ passed\ S_1}}{N_{total}} \times \mathcal{L} \times \sigma \times \frac{1}{P_1} \tag{10}$$

- **Overlap rate between signatures** $S_1$ **and** $S_2$: defined as the rate of events for which both signatures were satisfied ($N_{passed\ S_1 \& S_2}$). Hence, the overlap rate is computed as

$$OverlapRate(S_1, S_2) = \frac{N_{passed\ S_1 \& S_2}}{N_{passed\ S_2}} \times \mathcal{L} \times \sigma \times \frac{1}{P_1 \times P_2} \tag{11}$$

# 7 EvtVetoTool

In order to obtain trigger rates for higher luminosities, it is next to impossible to obtain high enough statistics using only minimum bias data that is generated. We therefore try to combine minimum bias rates with rates obtained from dijet samples. However, since these are overlapping samples, we must find some way to remove the overlap in the calculations. The solution is to apply a cut on the events used, based on the leading jet transverse momentum. If a jet with momentum higher than some threshold is found in minimum bias, the event is not accounted, while if the leading jet in a dijet event is lower than some threshold, the event is again discarded. This is done using TriggerRateTools' EvtVetoTool, which is nothing else than an Athena AlgTool.

EvtVetoTool's code is very compact and was separated from the main algorithm so that it is easily configurable. I is easily modifiable to use any quantity available through StoreGate, so that the veto is not limited to jet quantities, or any other specific quantity for that matter. Veto'ed events are not counted towards the total number of events used, so the effective cross-section of the new sample should be modified accordingly.

# 8 ROOT Objects

## 8.1 TD_Signature methods

- `Signature()` : default constructor.
- `Signature(TString name, TString level, double prescale)` : constructor.
- `virtual Signature()` : destructor.
- `TString* getName()` : returns a pointer to the TString containing the name of the Signature.
- `TString* getLevel()` : returns a pointer to the TString containing the level (L1,L2,EF) of the Signature.
- `double getXsec()` : returns the cross-section that was used, in barns.
- `double getLumi()` : returns the luminosity that was used in inverse barns per second.
- `double getPrescale()` : returns the prescale that was set of the signature.
- `int getNumEvts()` : returns the total number of events that were used with this signature (passed or not). Veto'ed events are not counted.
- `double getEff()` : returns the efficiency of the signature.
- `double getEffErr()` : returns the binomial error on the efficiency for the signature.
- `double getPeff()` : returns the prescaled efficiency of the signature.
- `double getRate()` : returns the acceptance rate (in Hz) of the signature.
- `double getRateErr()` : returns the binomial error on the acceptance rate.
- `double getIndepFrac()` : returns the independent fraction of the signature.
- `double getUniqueFrac()` : returns the unique fraction of the signature.
- `double getUniqueRate()` : returns the unique rate of the signature.
- `double getNumPassed()` : returns the number of events that passed the signature's criteria.
- `double getNumUnique()` : returns the number of events for which only this signture's criteria were satisfied.
- `double getIndep()` : return the not normalized independent fraction.
- `double getOverlapRateWith(TString sig)` : returns the overlap rate between the current signature, and another signature whose name is given by '`sig`'.

- `double getOverlapRateAt(int pos)` : returns the overlap rate stored at position '`pos`' in the vector of overlaps.
- `double getOverlapNumAt(int pos)` : returns prescaled number of events that overlap stored at position '`pos`' in the vector of overlaps.
- `TString* getOverlapNameAt(int pos)` : returns a pointer to the overlap name stored at position '`pos`' in the vector of overlaps.
- `int getOverlapNumSize()` : returns the size of the vector of overlap values.
- `int getOverlapNamesSize()` : returns the size of the vector of overlap names.
- `void setName(TString new_name)` : sets the signature's name to '`new_name`'.
- `void setLevel(TString new_level)` : sets the signature's level to '`new_level`'.
- `void setXsec(double new_xsec)` : sets the cross-section to '`new_xsec`'.
- `void setLumi(double new_lumi)` : sets the luminosity to '`new_lumi`'.
- `void setPrescale(double new_prescale)` : sets the signature's prescale to '`new_prescale`'.
- `void setNumEvts(double new_num_evts)` : sets the total number of events analyzed to '`new_num_evts`'.
- `void setNumPassed(double new_num_passed)` : sets the number of events that satified the signature to '`new_num_passed`'.
- `void setPnumUnique(double new_pnum_unique)` : sets the prescaled number of events that only satisfied the signature to '`new_pnum_unique`'.
- `void setIndep(double new_indep)` : sets the independent number of the signature to '`new_indep`'.
- `void addOverlap(TString sig)` : append a new signature with name '`sig`' to the overlap lists.
- `void setOverlapNum(int pos, double value)` : set number of overlaping events with the signature at position '`pos`' to be '`value`'.
- `void incNumEvts(int inc)` : increment the number of events studied by an amount '`inc`'.
- `void incNumPassed(double inc)` : increment the number of events that passed the signature by an amount '`inc`'.
- `void incPnumUnique(double inc)` : increment the prescaled number of events that only passed the signature by an amount '`inc`'.
- `void incIndep(double inc)` : increment the independent number by an amount '`inc`'.
- `void incOverlapNum(int pos, double inc)` : increment the number of overlapping events with the signature at position '`pos`' by an amount '`inc`'.
- `void clearOverlaps()` : replaces the existing vectors associated with overlaps by empty ones.

## 8.2 TD_CplxSignature methods

- TD_CplxSignature() : default constructor.
- TD_CplxSignature(TString name, TString type, TString level) : constructor.
- virtual TD_CplxSignature() : destructor.
- TString* getName() : returns a pointer to the TString containing the name of the complex signature.
- TString* getType() : returns a pointer to the TString containing the type of the complex signature.
- TString* getLevel() : returns a pointer to the TString containing the level (L1,L2,EF) of the complex signature.
- double getXsec() : returns the cross-section that was used, in barns.
- double getLumi() : returns the luminosity that was used in inverse barns per second.
- double getPrescale() : returns the effective prescale that can be computed for the complex signature.
- int getNumEvts() : returns the total number of events that were used with this signature (passed or not). Veto'ed events are not counted.
- double getEff() : returns the efficiency of the signature.
- double getEffErr() : returns the binomial error on the efficiency for the signature.
- double getPeff() : returns the prescaled efficiency of the signature.
- double getRate() : returns the acceptance rate (in Hz) of the signature.
- double getRateErr() : returns the binomial error on the acceptance rate.
- double getNumPassed() : returns the number of events that passed the complex signature's criteria.
- double getPnumPassed() : returns the prescaled number of events that passed the complex signature's criteria.
- int getSigIndexAt(int pos) : return the index corresponding to the component signature at position 'pos' in the list of component signatures of the complex signature.
- double getSigPrescaleAt(int pos) : return the additional prescale corresponding to the component signature at position 'pos' in the list of component signatures of the complex signature.
- TString* getSigAt(int pos) : return a pointer to the component signature's name at position 'pos' in the list of component signatures of the complex signature.
- double getOverlapRateWith(TString sig) : returns the overlap rate between the current signature, and another signature whose name is given by 'sig'.
- double getOverlapRateAt(int pos) : returns the overlap rate stored at position 'pos' in the vector of overlaps.
- double getOverlapNumAt(int pos) : returns prescaled number of events that overlap stored at position 'pos' in the vector of overlaps.
- TString* getOverlapNameAt(int pos) : returns a pointer to the overlap name stored at position 'pos' in the vector of overlaps.
- int getSigIndicesSize() : returns the size of the vector of component signature indices.
- int getSigNamesSize() : returns the size of the vector of component signature names.

- `int getOverlapNumSize()` : returns the size of the vector of overlap values.
- `int getOverlapNamesSize()` : returns the size of the vector of overlap names.
- `void setName(TString new_name)` : sets the signature's name to '`new_name`'.
- `void setType(TString new_type)` : sets the signature's type to '`new_type`'.
- `void setLevel(TString new_level)` : sets the signature's level to '`new_level`'.
- `void setXsec(double new_xsec)` : sets the cross-section to '`new_xsec`'.
- `void setLumi(double new_lumi)` : sets the luminosity to '`new_lumi`'.
- `void setNumEvts(double new_num_evts)` : sets the total number of events analyzed to '`new_num_evts`'.
- `void setNumPassed(double new_num_passed)` : sets the number of events that satified the complex signature to '`new_num_passed`'.
- `void setPnumPassed(double new_pnum_passed)` : sets the prescaled number of events that satified the complex signature to '`new_num_passed`'.
- `void addSig(TString sig, int index, double ps)` : append the name, index and additional prescale factor of a component signature to the appropriate vectors.
- `void addOverlap(TString sig)` : append a new (complex) signature with name '`sig`' to the overlap lists.
- `void setOverlapNum(int pos, double value)` : set number of overlapping events with the (complex) signature at position '`pos`' to be '`value`'.
- `void incNumEvts(int inc)` : increment the number of events studied by an amount '`inc`'.
- `void incNumPassed(double inc)` : increment the number of events that passed the complex signature by an amount '`inc`'.
- `void incPnumPassed(double inc)` : increment the prescaled number of events that passed the complex signature by an amount '`inc`'.
- `void incOverlapNum(int pos, double inc)` : increment the number of overlapping events with the signature at position '`pos`' by an amount '`inc`'.
- `void clearSigs()` : replaces the existing vectors associated with component signatures by empty ones.
- `void clearOverlaps()` : replaces the existing vectors associated with overlaps by empty ones.