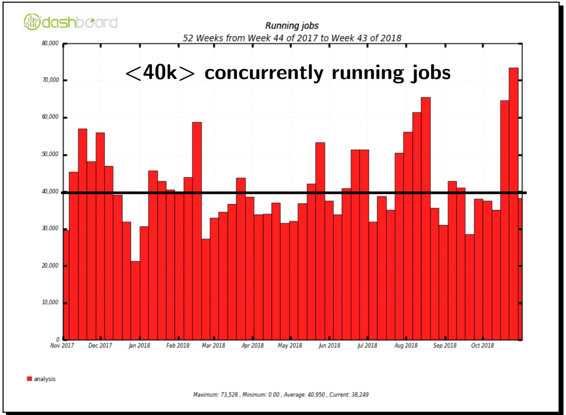
Boosting the efficiency of analysis jobs in CMS

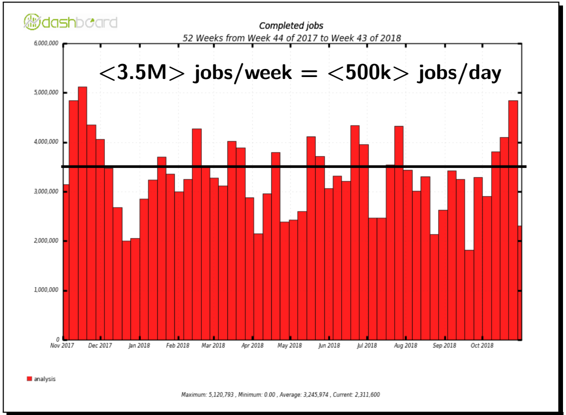
Leonardo Cristella
on behalf of the CMS collaboration

April, 2019

Introduction

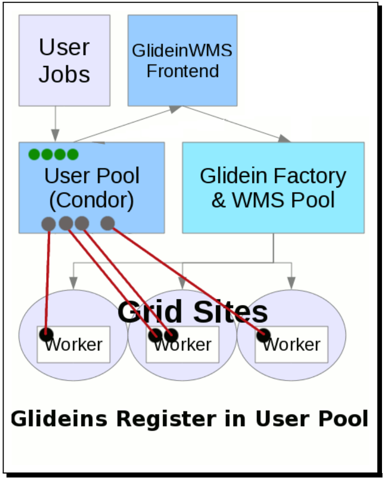
- CMS experiment has a workload management system that schedules and executes data processing, simulation and user analysis tasks on a distributed Grid infrastructure.
- **Past focus:** make jobs run, offer users **painless and transparent** access to the Grid. We have been largely successful.
- **Recent focus** on **efficiency and optimisation**: turnaround time, CPU efficiency, scalability of the system.
- **This contribution:**
  - improving the execution of "analysis jobs" $\equiv$ jobs submitted by users (few hundred different people at any time)
  - thus **without access** to the application itself





Global Pool & glideinWMS

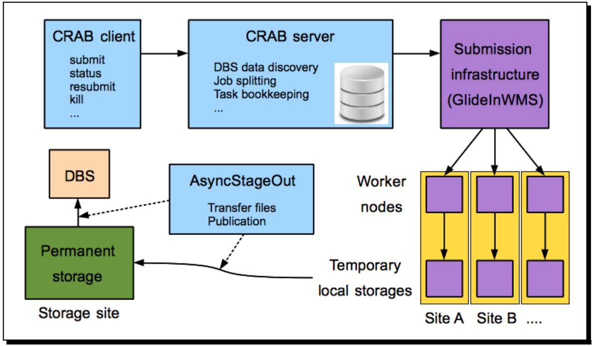
- Based on glideinWMS
  - **Users**: Vanilla HTCondor jobs via **ad hoc tools**: **WMAgent** for production, **CRAB** for analysis
  - **Glidein FrontEnd**: glideins (**PilotJobs**) → Grid Sites
  - **PilotJobs**: **48 hours, 8 cores**
  - **1 PilotJob** → **1 HTCondor** ***startd*** which joins the *Global Pool*
  - **1 PilotJob** runs **many single/multi-core jobs** and keeps reallocating freed up cores until the end of its lifetime
- **CMS takes ownership** of all issues of **pool fragmentation** due to running variable number of single/multi-core jobs of different length



Analysis jobs submission: CRAB

CMS Remote Analysis Builder (CRAB)

1. Turns a high level request (*run this executable on this set of data*) into a set of jobs whose execution is controlled by HTCondor DAGMAN
2. **Splitting**: one request (**"task"**) → many jobs
3. An *Asynchronous StageOut* component moves job outputs from remote site storage to the user preferred site
   - Optionally record the files in CMS *Dataset Bookkeeping System*



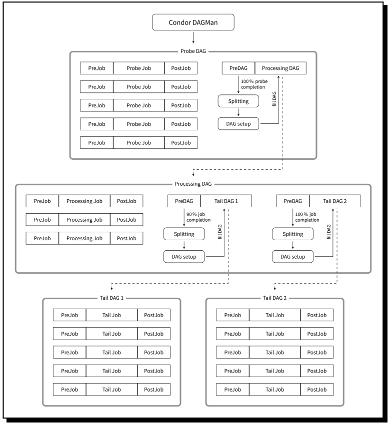
Three lines of work

Optimization of

1. **job splitting: Automatic splitting**
   Optimise **job running time** (splitting a large task in many jobs)
   - *Too long*: high chance of killing by glitch → wasted resource
   - *Too short*: high number of jobs, unnecessary load on infrastructure, too much time in overheads
2. **execution time requested by jobs: Time-request tuning**
   Optimise **job to pilot slot allocation** (tune the job time requirement)
   - Avoid killing/restarting pilots too soon, exploit the tail of each slot
   - Majority of jobs asks for 20h but the median runtime is only 30min or less, how close to the pilot end of life is OK to start them?
3. **execution site for jobs: Overflow**
   Optimise **jobs scheduling across sites** (overflow from busy site queues)
   - Jobs used to run where data are
   - The CMS global data federation allows jobs to read input data from a remote site
   - but can't fully ignore where data are

Automating Splitting: Theory

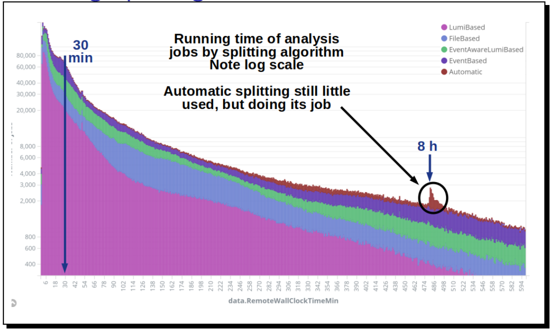
- **Before**: 1 task → 1 DAG
  - Splitting parameters **configured by the user**
  - May result in **thousands of very short jobs**
    - Bad for scaling
- **With Automatic splitting**: 1 task → a few DAG's
  - All decisions taken **out of user hands**
  - One Probe DAG to estimate **jobs time, memory, disk needs**
  - Splitting parameters **computed in a per event basis**
  - Target: **8-hours jobs**
  - One Processing DAG to do the actual work: jobs are set to run for a **fixed time**. If they don't complete all work, they finish **gracefully** and the remaining work is taken care by the tail jobs
  - Up to 3 tail stages - 3 Tail DAGS
    - complete remaining work (processing leftovers and failed jobs)

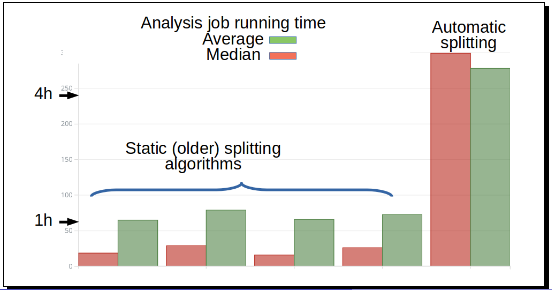


Automating Splitting: Practice

- **Before**: 1 task → 1 DAG
  - **Good**: splitting done in the TaskWorker (**one server**, centralized logs, **easy to debug**)
  - **Bad**: user finds best splitting by trial and error running same tasks **N times (invisible waste)**
  - **Overall**: non optimal, but in case of problems it's on the user to improve the splitting
- **With Automatic splitting**: 1 task → a few DAG's
  - **Good: it proved to work smoothly**
  - **Hard part**:
    - Splitting done on the schedd (**15 machines**, log scattered in user directories, hard to rerun in debug mode).
    - **Not all use cases** are addressed so far, e.g. for MonteCarlo generation there is currently no automatic splitting available.
  - **Overall**:
    - Significant efficiency gain
    - In case of problems, operator action needed
    - Large variation of worker nodes CPU power and data serving performance at sites introduces large uncertainty in jobs run times.
      This leads directly to the need for time-request tuning (next slides).

Automating Splitting: Results





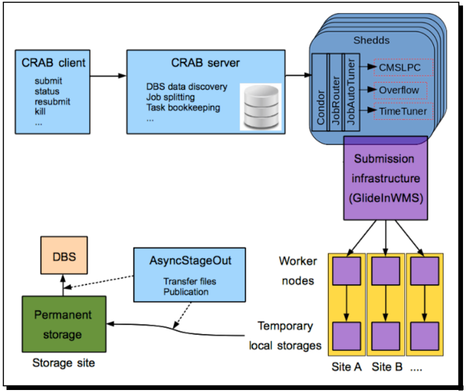
- In production since February 2018
- Users encouraged but not pushed
- Few issues, generally high satisfaction
- Extending usage requires education campaign: manpower issue
  - Next step once all commissioning work is completed
- **Current adoption is about 2%**
  - Expect to move most of the users by the start of Run-III

Editing job requirements: HTCondor JobRouter

Both following lines of work (Time-request tuning and Overflow) rely on modifying job requirements while jobs are idle in the HTCondor queue → different scheduling → freedom to optimise.

- **Tension**: **Global overview** (best decisions) vs. **Local action** in each schedd (efficient).

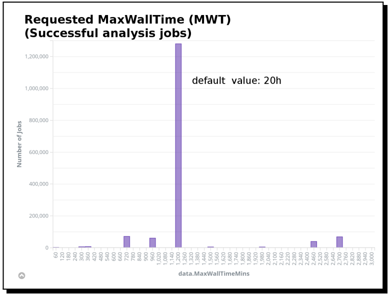
- **Criticality**: **spiral of death**
  Massive condor_qedit ⟹ schedd load ⟹ long negotiator time ⟹ starving pilots ⟹ job restarts ⟹ more load on the schedds
- **Solution**:
  - A central process to collect information and make **statistics based on a feed of HTC classAds to Elastic Search**.
  - HTCondor **JobRouter** to perform the actual classAd remapping locally in each schedd
  - Strategy already in use for central Production, profiting from **larger workflows** and much more **top/down control**
- **Slow feedback** → care in turning knobs
  - $O(10min)$ for jobs tuning vs. $O(hours)$ for HTCondor reaction

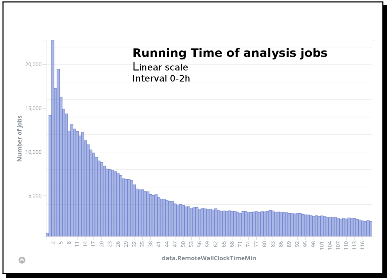


Time-request tuning: Theory

Jobs request a **MaxWallTime (MWT)** at submission, set by the user

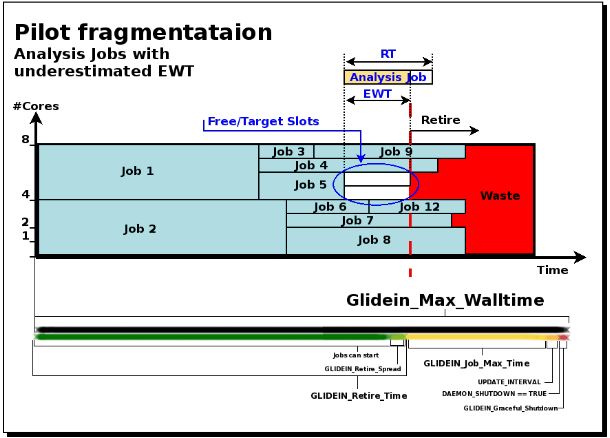
- MWT is a common classAd atribute for all jobs in a task
- HTCondor **kills** jobs which hit the MWT
- **Problem**: majority of jobs ask for the **default 20h** MWT but the median runtime $<$ **30min**
  - Although all task jobs are designed to be very similar, they have quite different runtimes.
    Users need to request a largish, safe, value.
  - But long jobs are more difficult to schedule inside multicore pilots
  - Automatic splitting will help but not solve
    - ★ Set a realistic limit for the Processing DAG
    - ★ Jobs running longer are resplitted so that a safe MWT can be set (still $O(hour)$)
- **Approach**: **introduce EstimatedWallTime (EWT).**
  - Use EWT to schedule, MWT to kill
  - EWT (realistic) $<<$ MWT (conservative)





Time-request tuning: Practice

- **Implemented solution:**
  - EWT computed **as soon as one job completes** and dynamically updated every 10min
  - EWT estimate algorithm tuned to contain most but not all jobs:
    - ★ pick **95th percentile** of collected run-Times and apply **correction** dependent on **#jobs**
  - EWT added and updated in each job via JobRouter
  - Jobs can **keep running** in the **pilot's tail** (the pilot's retire time) even after EWT expires, up to MWT
  - If a job reaches a pilot's **end of lifetime**, it is automatically and transparently **restarted** by HTCondor (but CPU is wasted)

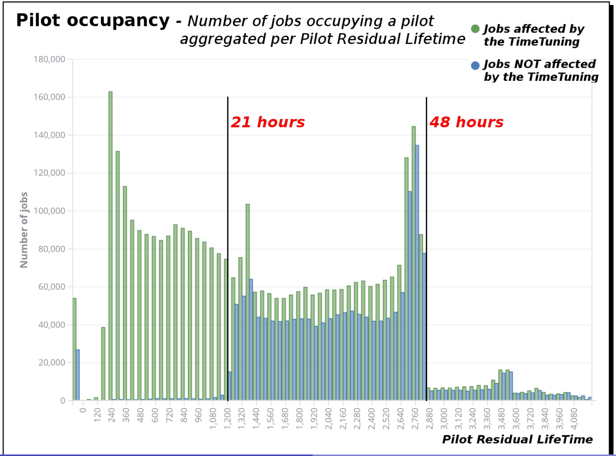


- **Issues to overcome:**
  - Limited statistics to work with, first jobs to complete may be **not representative** of the whole task
  - Properly measure what we gain (**less fragmentation**) and what we lose (**wasted CPU**)

Time-request tuning: Results

CONSERVATIVE SETUP

Jobs running less than EWT	95%
Jobs running longer but completed	4%
Jobs restarted once and completed	1%



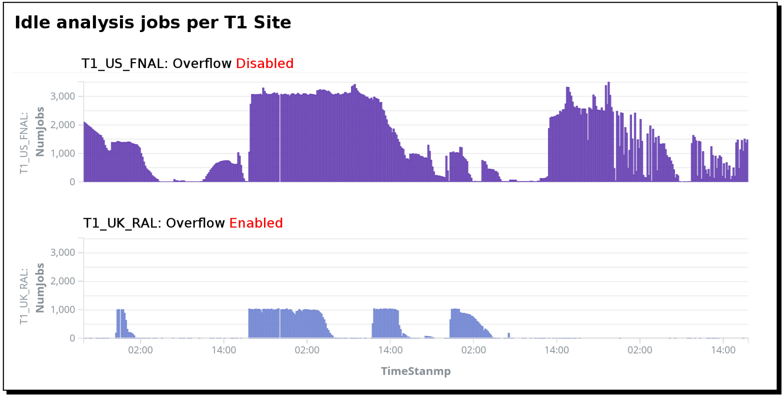
STILL VERY EFFECTIVE:

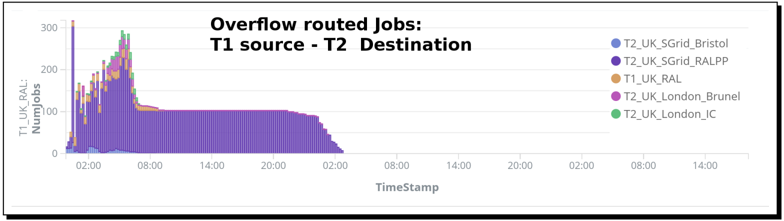
- Jobs which were TimeTuned filled mostly short living pilots
- Jobs which were not TimeTuned filled pilots longer than 21 hours

Overflow: Why, What, How

- **Problem**: jobs sent to sites which hosts the input data (smaller network latency)
  - **long waiting times** when target sites overloaded
- **Solution**: run some of those jobs elsewhere
  The CMS global data federation allows jobs to read input data from a **remote site**
- **Old way**: ad hoc glideinWMS FrontEnd group to define topology and running limits.
  Deployed for US sites since a few years, works but can't extend it:
  - US sites are large and homogeneous. Dedicated pilots $\rightarrow$ pool fragmentation.
  - GlideinWMS suffers with many FrontEnd groups
- **New way**: JobRouter **dynamically changes list of desired execution sites** for some jobs
  - Central overview opens to advanced **scheduling decisions**: e.g. add WAN information
- **Difficulties**
  - How much (more) remote reading can a given site handle?
  - More remote reads = more, harder to debug, failures
  - Large differences in site size and connectivity
  - Need to go over country boundaries
- **Approach**: **start slowly, watch carefully, push slowly, iterate**
  - Users stand "wait but OK" better than "fail and need to retry"
  - Can't be source of a DDoS attack on our sites

Overflow: Results





- **Current use limited to T1s**
  Which is where problem is bigger: analysis jobs get a small share at T1s but many datasets are only placed at one T1 due to disk space limitation.
- **Work in progress**
  Implementing a maximum overflow in a country and providing a way to substitute the old overflow.

Summary

- CMS operates a complex setup with $O(40k)$ analysis jobs running at any moment and where many parameters change constantly outside Analysis Operation control.
- We have to be careful. It is not easy to push changes in production transparently to the user community nor to disentangle effects of the various changes.
- And that while the monitoring infrastructure is being migrated/rebuilt.

But we managed to deploy the needed knobs and dials
to **optimise the job scheduling** in the CMS distributed analysis system
and we look forward to learn how to better tune the system.

Future lines of work

Part of the work requires guessing

- what users really need
  - Inferring the behaviour and needs of large tasks from small initial samples
  - Very tricky in case of many tasks with not many (< 100) jobs each
- the future
  - How sites and networks will react to load that we are about to place on them

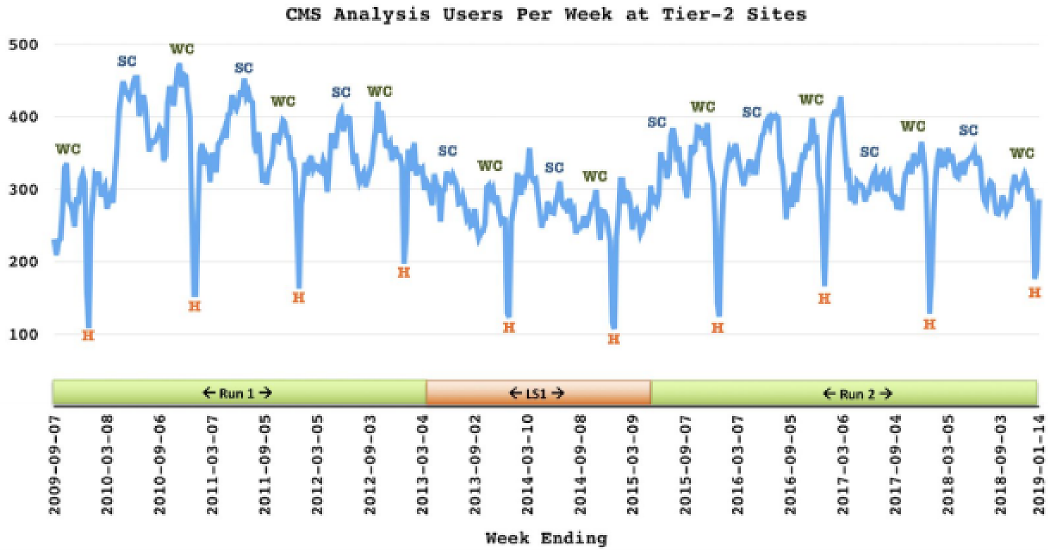
It is a good arena for:

- Central vs. Local control
- Infer large sample behaviour from limited statistics
- Machine Learning
- Network scheduling

Future will be more fun than the past!

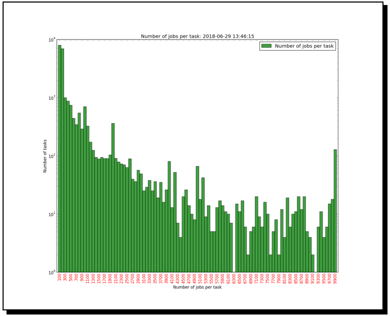
Backup slides

CRAB users vs time

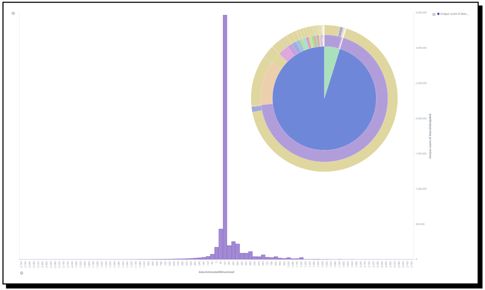


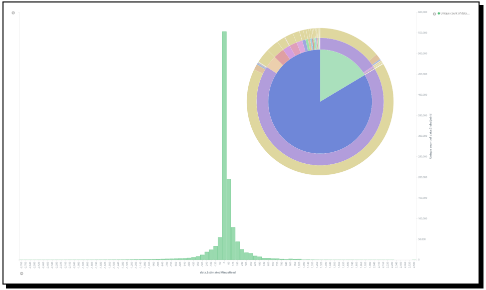
Current implementation of Time-request tuning

JobCountPerTask (LogScale):



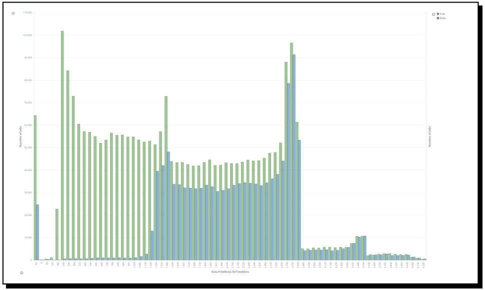
Current results

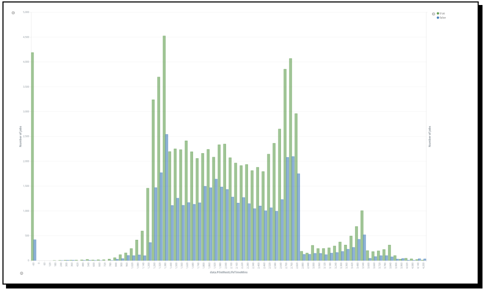




Current results

- – Few more plots showing the results –





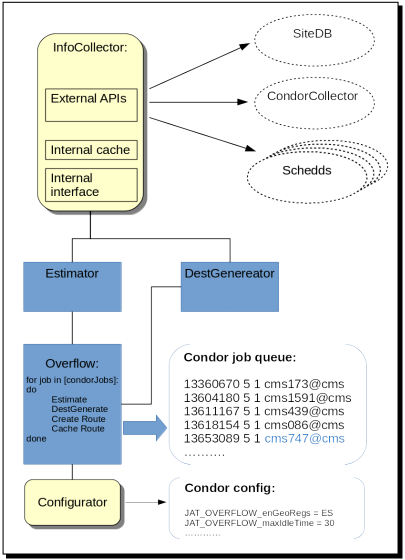
Overflow - Problem:

- **The primary need:** to achieve a better resource utilisation
- **The secondary need:** to protect the sites from being flooded with jobs they cannot process || serve data for them.
- **The old Overflow mechanism** - what does it suffer from:
  - Statically defined overflow regions - can't be based on other criteria characterizing "proximity"
  - Overflow matching decision happens in the timescale of pilot lifetimes - not flexible enough to respond to faster changes in the status of the distributed CPU and storage resources
  - Requires additional FE groups to be set - a limitation in practice to the different number of settings that could be configured at once.
  - Based on a special type of pilots - fragmentation of the resources, increasing wastage

Structure:

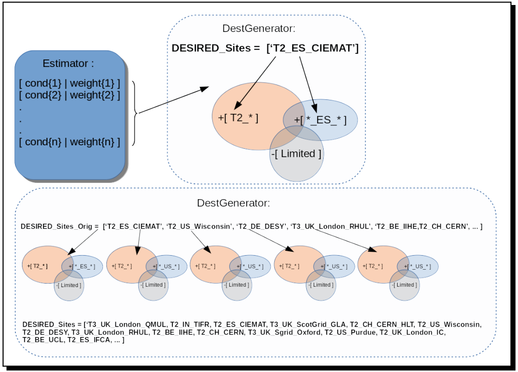
Three basic abstractions:

- **Information Lifetime:**
  - static
  - dynamic
- **The OverflowLevel:**
  - PERTASK
  - PERJOB
  - PERBLOCK
  - PERFILE
  - PERDATASET
- **The OverflowType:**
  - GEO
  - TIER1
  - TIER2
  - DATALOCATION
  - LOCALLOAD
  - SRCLOAD
  - DSTLOAD



Decision making & Subsets

- Weighted sum vs. Weighted single decision.
- Estimating the weights could be dynamic:
  In the future we can apply more elaborate mechanisms for estimating the optimal weights according to the prompt feedback about the reaction of the system.
- Subsets intersections.



New Methods for improving the accuracy of Automatic Time-request tuning

We estimate the Job Wallclock Time (EWT) based on the first completed jobs (minTaskStat) and continuously modify the Requested Wallclock Time of the idle jobs while gaining statistics. This is a method which has the intrinsic characteristics of a negative feedback amplifier. As expected, the error with respect to the Real Time (RT) follows an normal distribution':

$$err = EWT - RT \quad (1)$$

In order to minimise this error and avoid negative values we introduce a correction factor:

$$err = CorrFactor * EWT - RT$$
$$CorrFactor = f(n)$$
$$n : numberofcompletedjobs$$

Different correction factors considered:

- static correction factor, a Heaviside function:

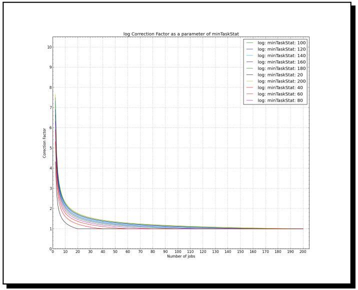
$$f(n) = \begin{cases} 1 & \text{if } n < \text{minTaskStat} \\ const & \text{if } n > \text{minTaskStat} \end{cases} \quad (2)$$

  minTaskStat - here is a config parameter which acts as a trigger for the mechanism

- logarithmic correction factor:

$$f(n) = \log_n(minTaskStat) \quad (3)$$

  - very steep
  - minTaskStat - is now a parameter defining the slope of the function that the correction factor will follow while gaining more statistics
  - a single parameter function
  - the negative error is still at around 16% (shows dependency on more than a single parameter)



New Methods for improving the accuracy of Automatic Time-request tuning

- polylogarithmic:
  motivation - commonly used for estimating the order of time or memory consumption

$$f(n) = \sum_{k=1}^{\epsilon} a_k (\log_n(minTaskStat))^k \qquad (4)$$

  – more moderate slope
  – high computational cost: $O(n^{\epsilon})$ for high values of ϵ
  – now we can easily put more than a single parameter in the function and decide the order/degree up to which we want to calculate and ϵ becomes the number of independent parameters.
  candidate parameters:
  - job dependent:
    - number of jobs in the workflow with error code diff 0
    - dataset characteristics: like number of lumisections
    - distance between slot and dataset ... etc.
  - infrastructure dependent:
    - network throughput of the slot
    - reliability of the (slot) ... etc