How to measure the W mass with 10 MeV uncertainty

EP-IT Data Science Seminars

16 October 2024, CERN

David Walter (CERN) on behalf of the CMS Collaboration

Introduction

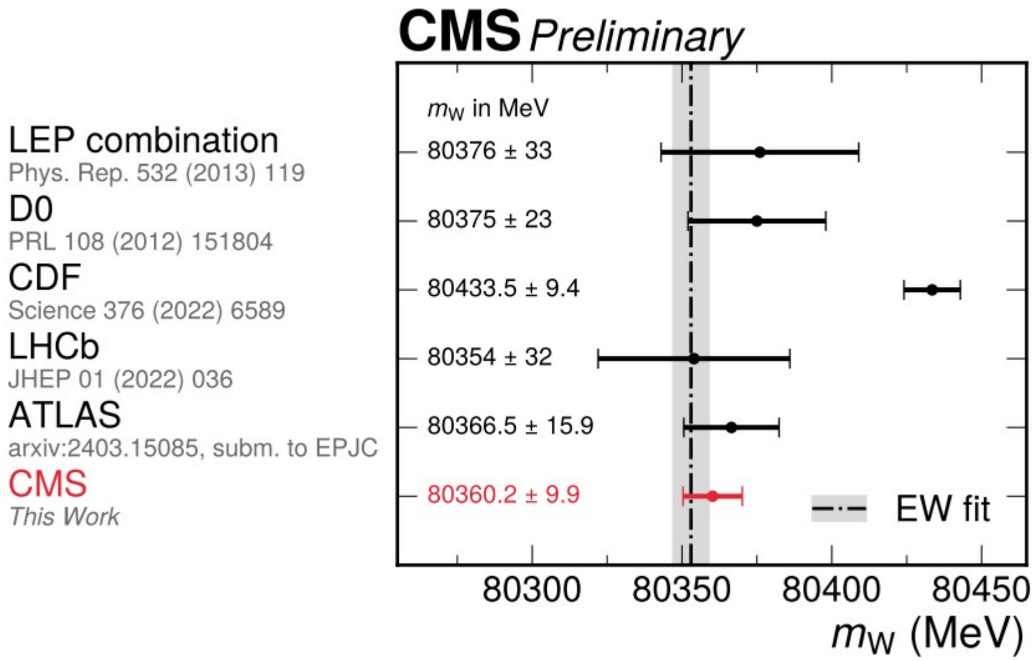
Analysis presented in LPC seminar last month

- Document: [CMS-PAS-SMP-23-002]

First measurement of m_W from CMS

- Most precise at LHC
- In agreement with the SM but in tension with CDF

This seminar will focus on the technical aspects



Use 16.8 fb^{-1} pp collision data at $\sqrt{s}$=13TeV

HL-LHC

Large inclusive W cross section

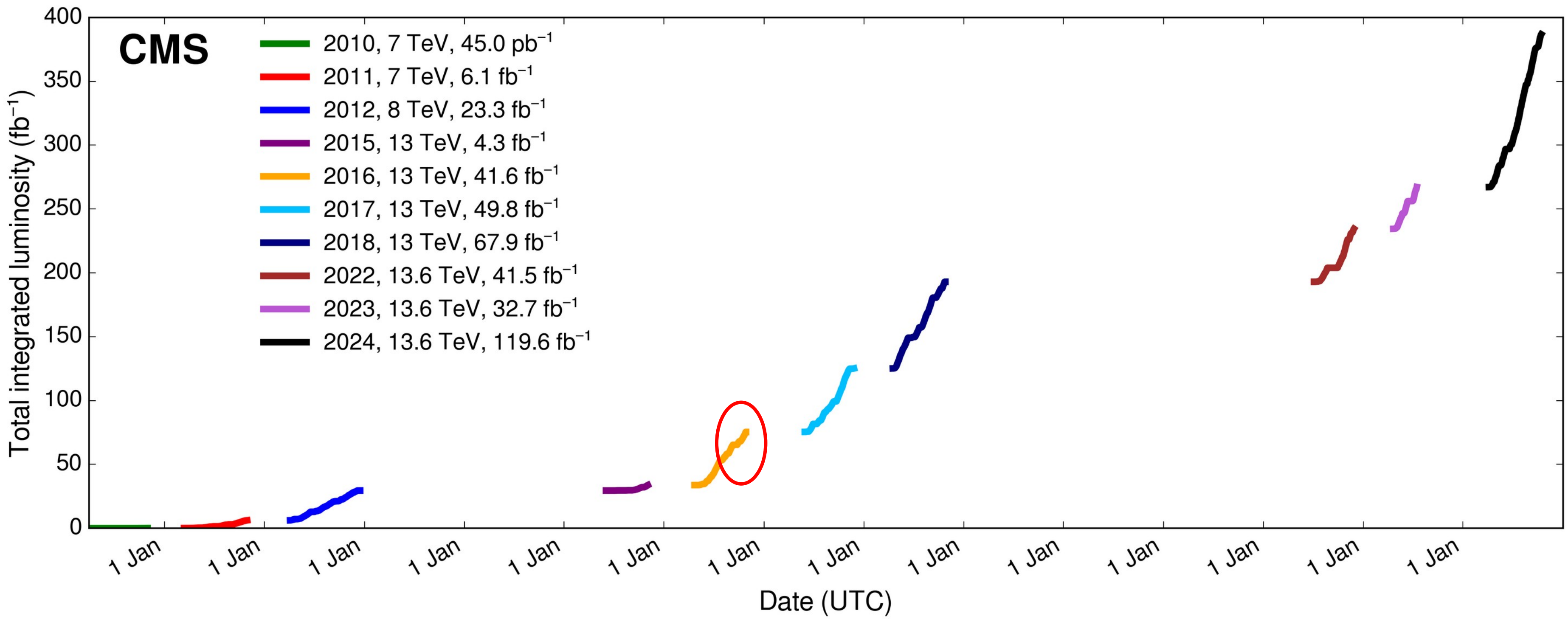
- 300M data and 4B MC events (4 times MC statistical power)

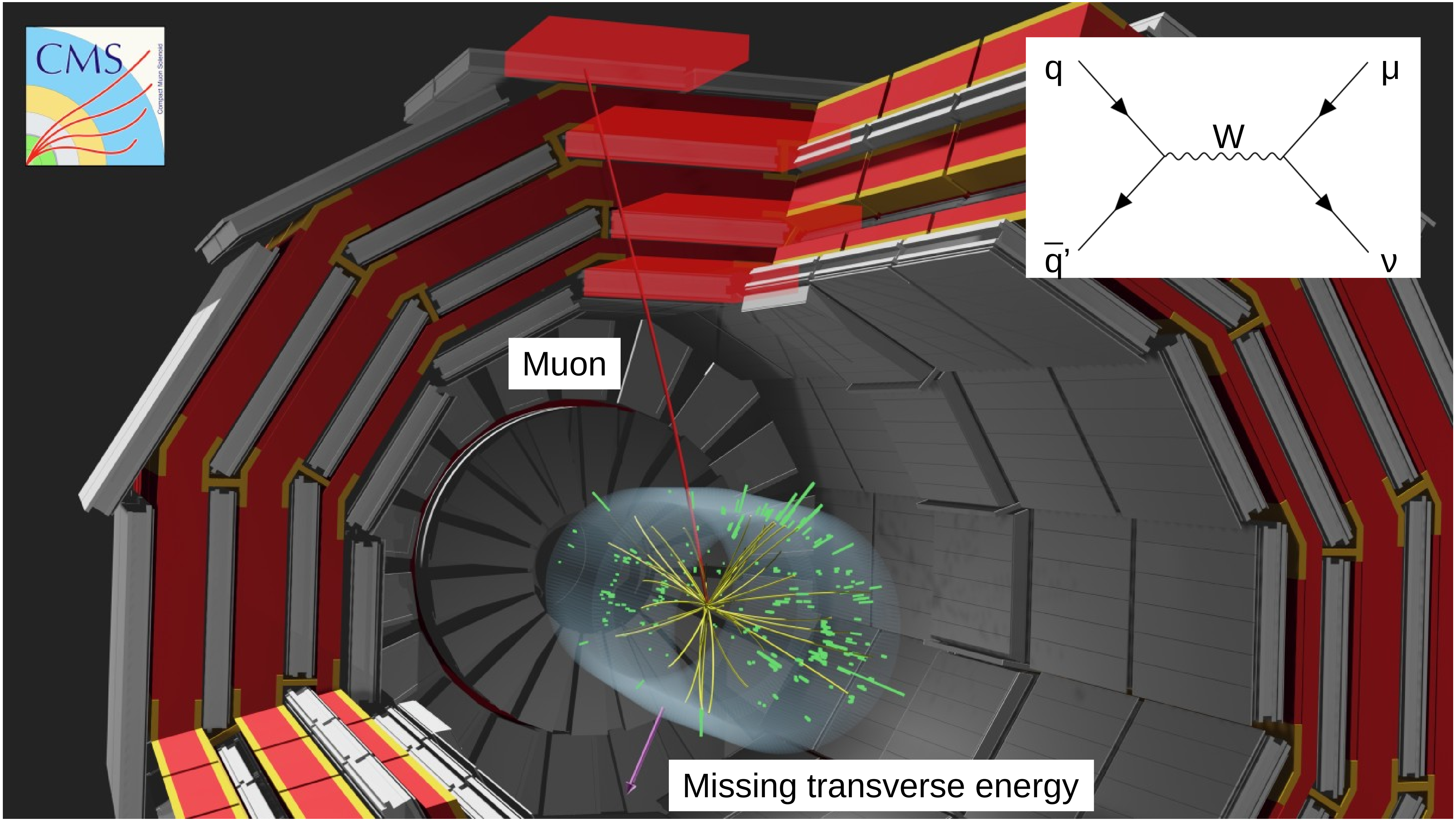
Largest dataset used for W boson mass analysis

- Opportunity to exploit multi dimensional information
- Challenging data processing

Much more data available now and in the years to come

→ Software developments have to keep up with technical challenges





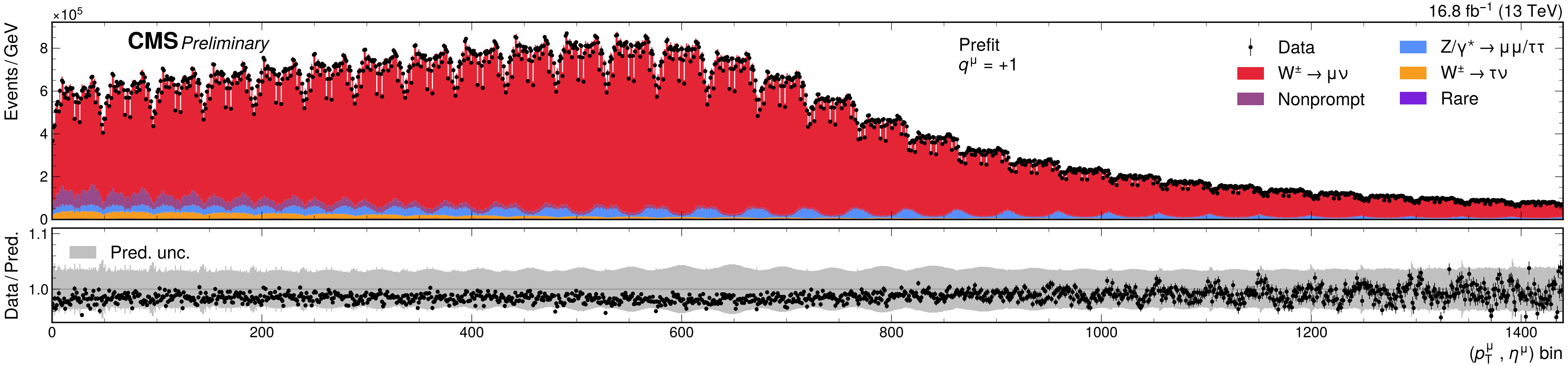
The measurement is performed using the muon kinematics only

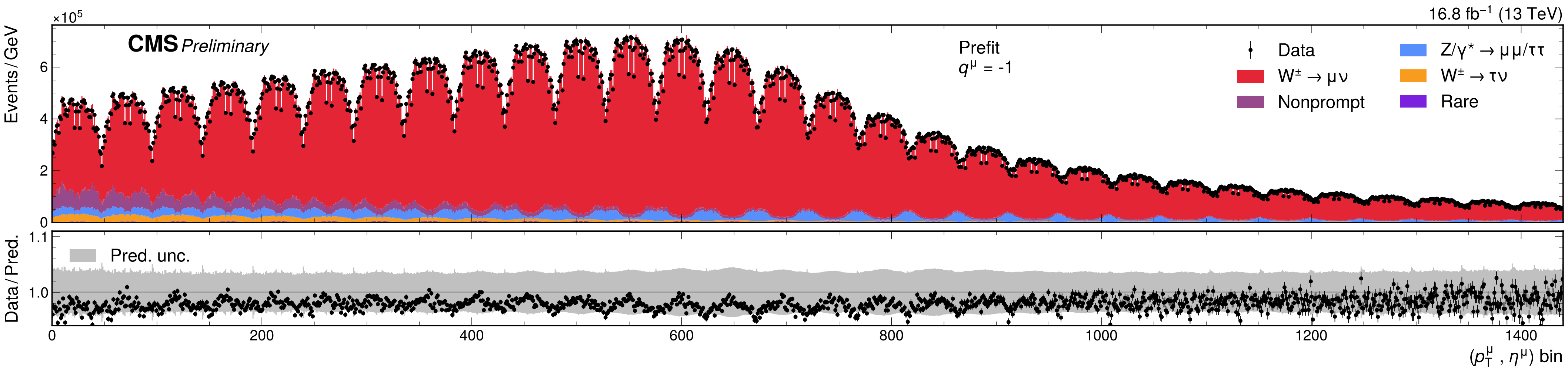
How we measure the W boson mass

Strategy to use large data sample and constrain theory uncertainties in-situ

Profile likelihood fit to single muon p_T η, charge distribution

- 2880 bins

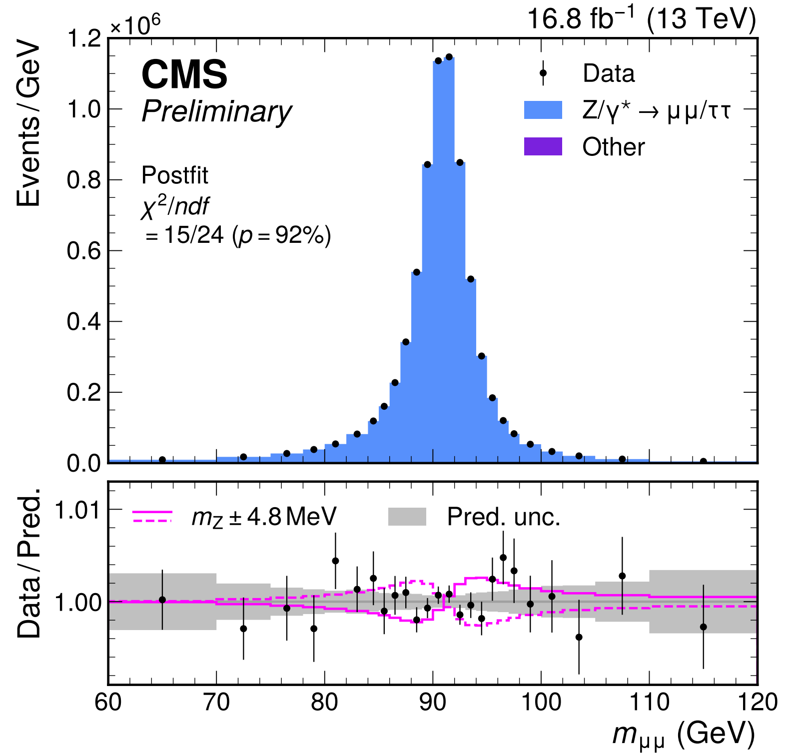


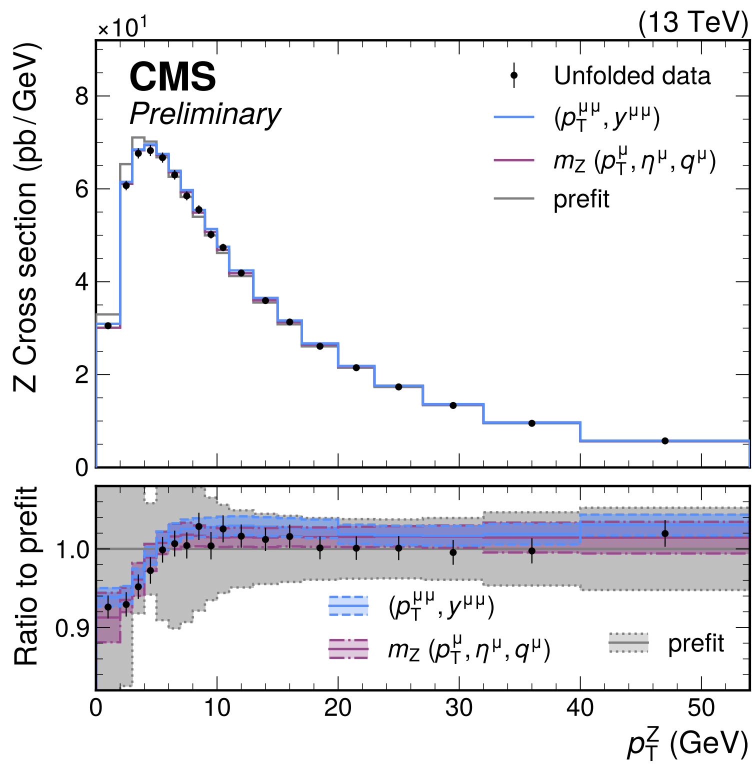


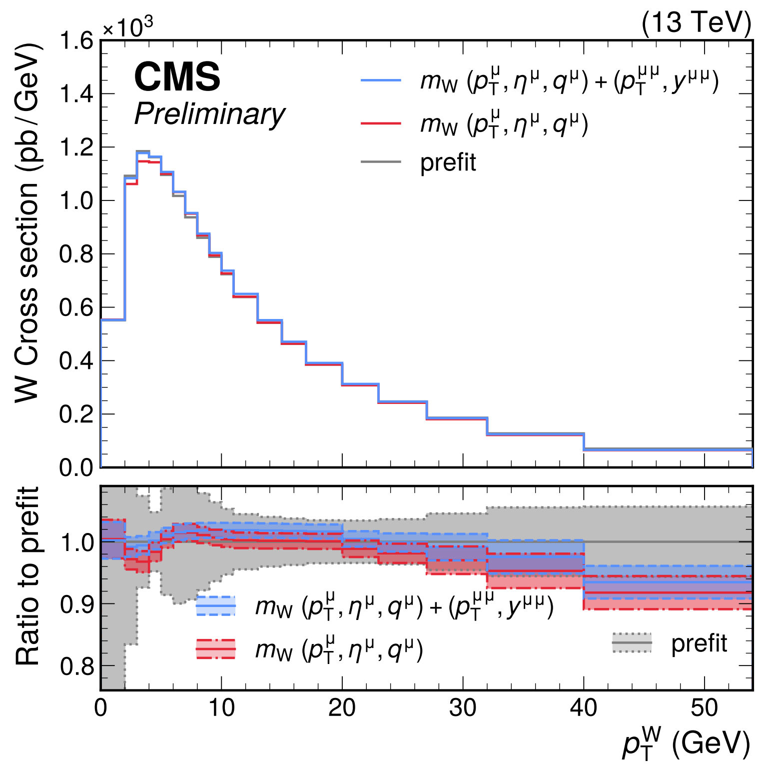
Multiple analyses in one

- Z dilepton m_{ll}, p_T^Z-y^Z, W-like
- Unfolding
- Helicity cross section fit
- Generator studies
- ...

Different configurations, including combined fits





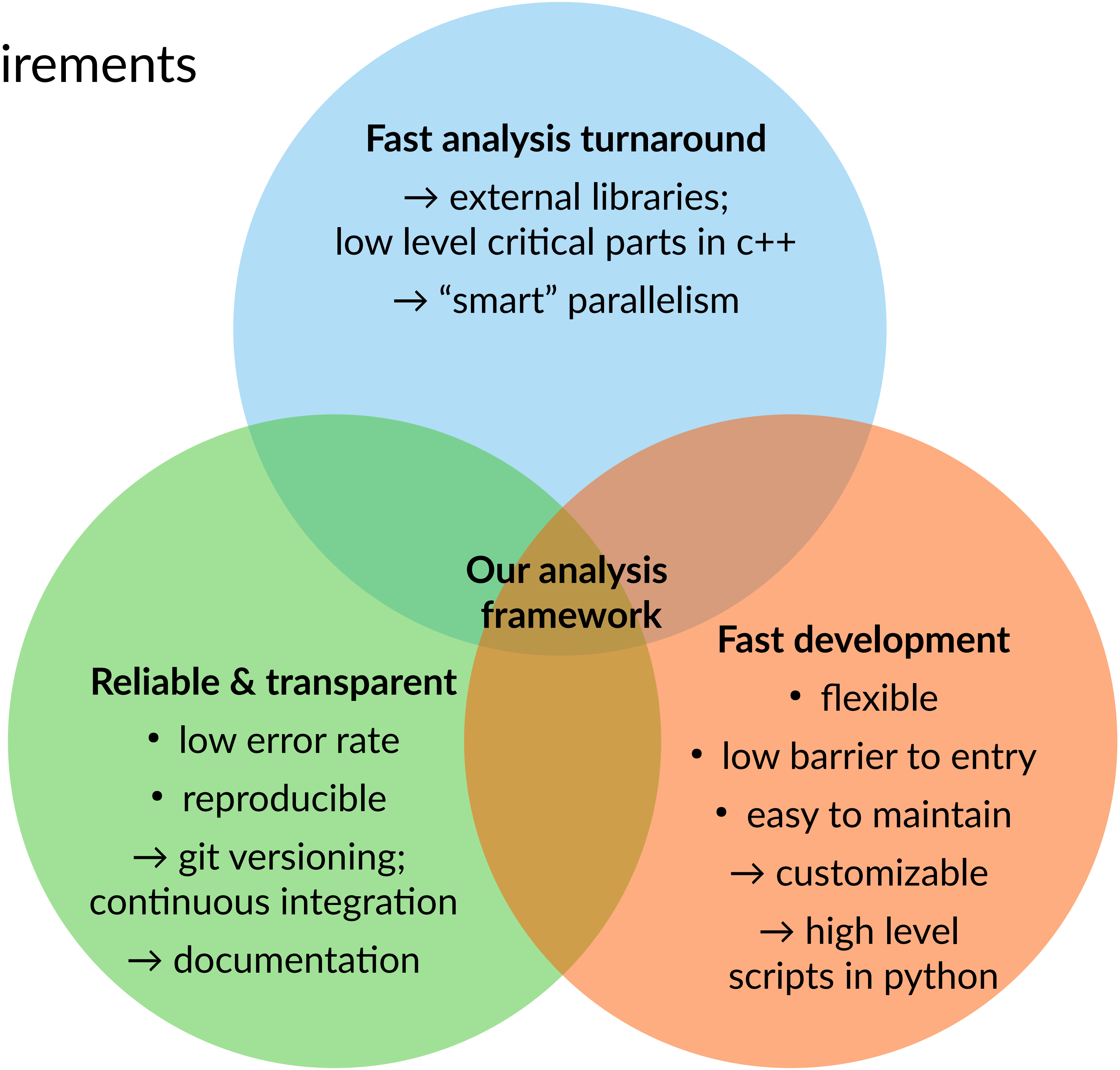


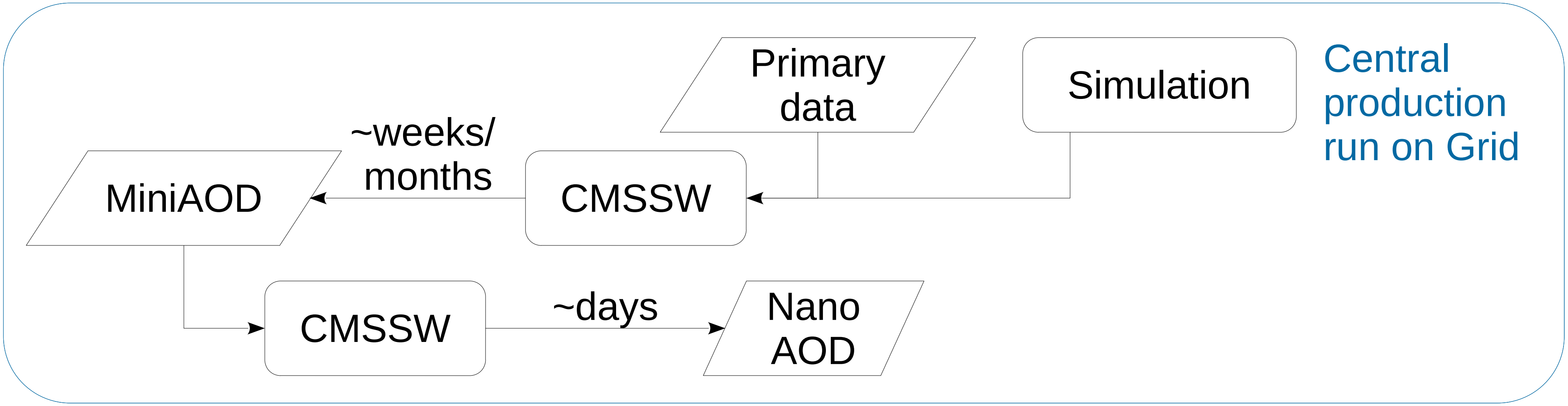
Precise treatment of uncertainties requires large amount of variations

- O(1000) parameters in single fit

<table>
<tr><th>Systematic uncertainties</th><th>W-like m_Z</th><th>m_W</th></tr>
<tr><td>Muon efficiency</td><td>3127</td><td>3658</td></tr>
<tr><td>Muon eff. veto</td><td>–</td><td>531</td></tr>
<tr><td>Muon eff. syst.</td><td colspan="2">343</td></tr>
<tr><td>Muon eff. stat.</td><td colspan="2">2784</td></tr>
<tr><td>Nonprompt background</td><td>–</td><td>387</td></tr>
<tr><td>Prompt background</td><td>2</td><td>3</td></tr>
<tr><td>Muon momentum scale</td><td colspan="2">338</td></tr>
<tr><td>L1 prefire</td><td colspan="2">14</td></tr>
<tr><td>Luminosity</td><td colspan="2">1</td></tr>
<tr><td>PDF (CT18Z)</td><td colspan="2">60</td></tr>
<tr><td>Angular coefficients</td><td>177</td><td>353</td></tr>
<tr><td>W MiNNLO$_{PS}$ μ_F, μ_R</td><td>–</td><td>176</td></tr>
<tr><td>Z MiNNLO$_{PS}$ μ_F, μ_R</td><td colspan="2">176</td></tr>
<tr><td>PYTHIA shower k_T</td><td colspan="2">1</td></tr>
<tr><td>p_T^V modeling</td><td>22</td><td>32</td></tr>
<tr><td>Nonperturbative</td><td>4</td><td>10</td></tr>
<tr><td>Perturbative</td><td>4</td><td>8</td></tr>
<tr><td>Theory nuisance parameters</td><td colspan="2">10</td></tr>
<tr><td>c, b quark mass</td><td colspan="2">4</td></tr>
<tr><td>Higher-order EW</td><td>6</td><td>7</td></tr>
<tr><td>Z width</td><td colspan="2">1</td></tr>
<tr><td>Z mass</td><td colspan="2">1</td></tr>
<tr><td>W width</td><td>–</td><td>1</td></tr>
<tr><td>W mass</td><td>–</td><td>1</td></tr>
<tr><td>$\sin^2\theta_W$</td><td colspan="2">1</td></tr>
<tr><td>Total</td><td>3750</td><td>4859</td></tr>
</table>

Requirements





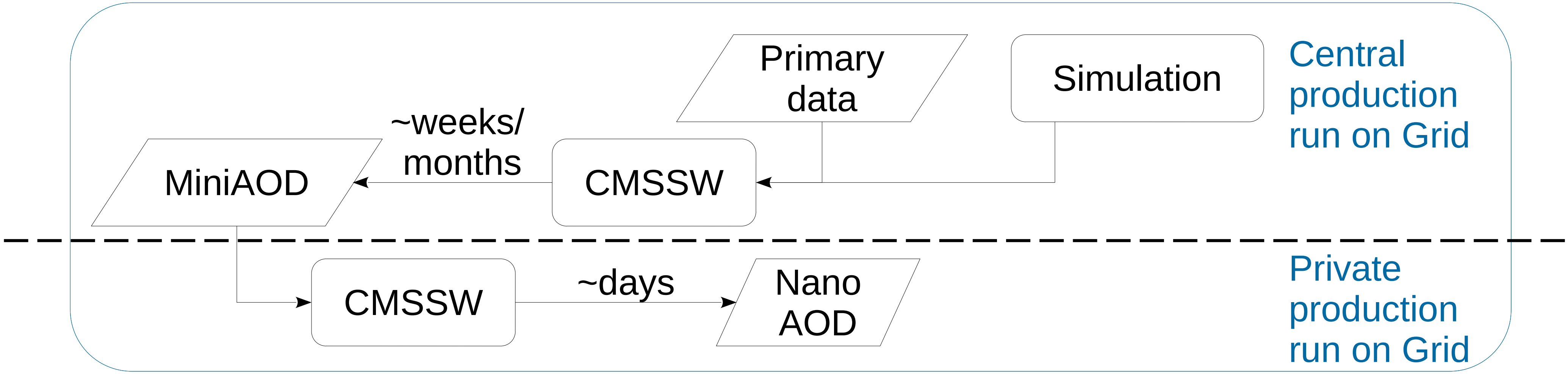
Shorten the gap between data and results: NanoAOD

Central supported compact CMS event data format [0,1]

- Flat ROOT TTree
  - Independent of experiment specific software
- High level physics objects
  - (p_T, η, φ, ID, ... of muons, electrons, jets, ...)
- ~2kB per event
- Good for ~50% of analyses

Data tier	Size (kB)
RAW	1000
Gen	<50
SIM	1000
DIGI	3000
RECO(SIM)	3000
AOD(SIM)	400
MiniAOD(SIM)	50
NanoAOD(SIM)	2

Analysis data formats



Shorten the gap between data and results: NanoAOD

Central supported compact CMS event data format [0,1]

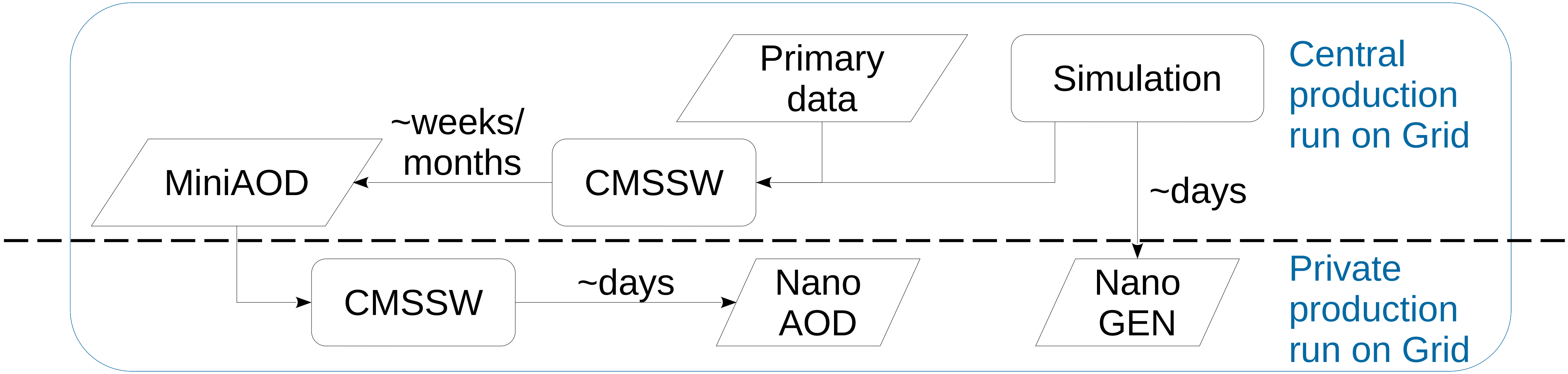
- Flat ROOT TTree
  - Independent of experiment specific software
- High level physics objects
  - (p_T, η, φ, ID, ... of muons, electrons, jets, ...)
- ~2kB per event

Easy customization with additional information

- Alternate PDFs, Info for muon track fit, ...

Data tier	Size (kB)
RAW	1000
Gen	<50
SIM	1000
DIGI	3000
RECO(SIM)	3000
AOD(SIM)	400
MiniAOD(SIM)	50
NanoAOD(SIM)	2

Analysis data formats



NanoGEN and NanoLHE

NanoAOD with only then GEN-related branches

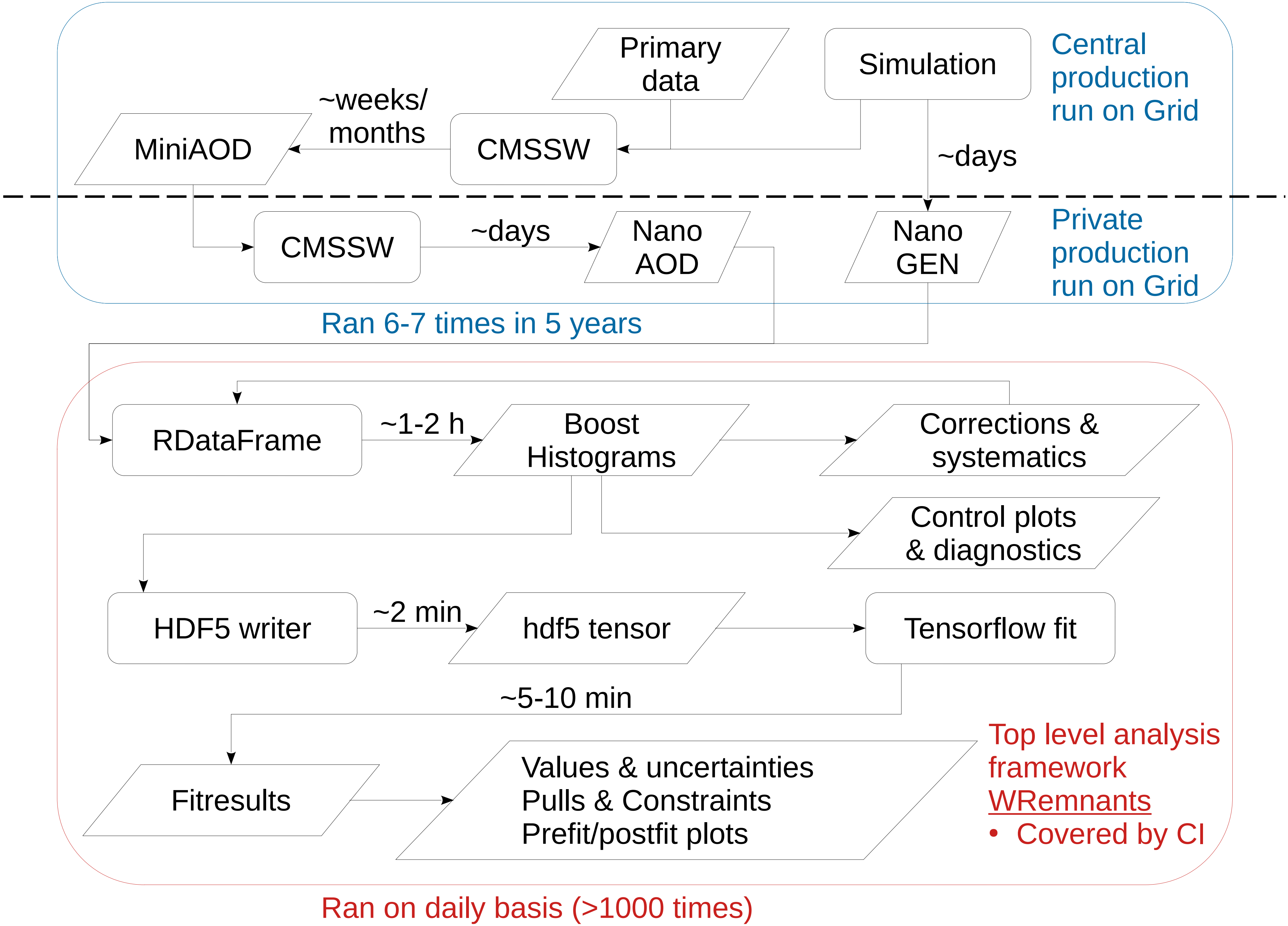
- Developed to validate MiNNLO event generator
  - Now centrally supported in CMS
- Producible directly from gridpack
- Lightweight, no detector simulation
- ~0.4kB per event

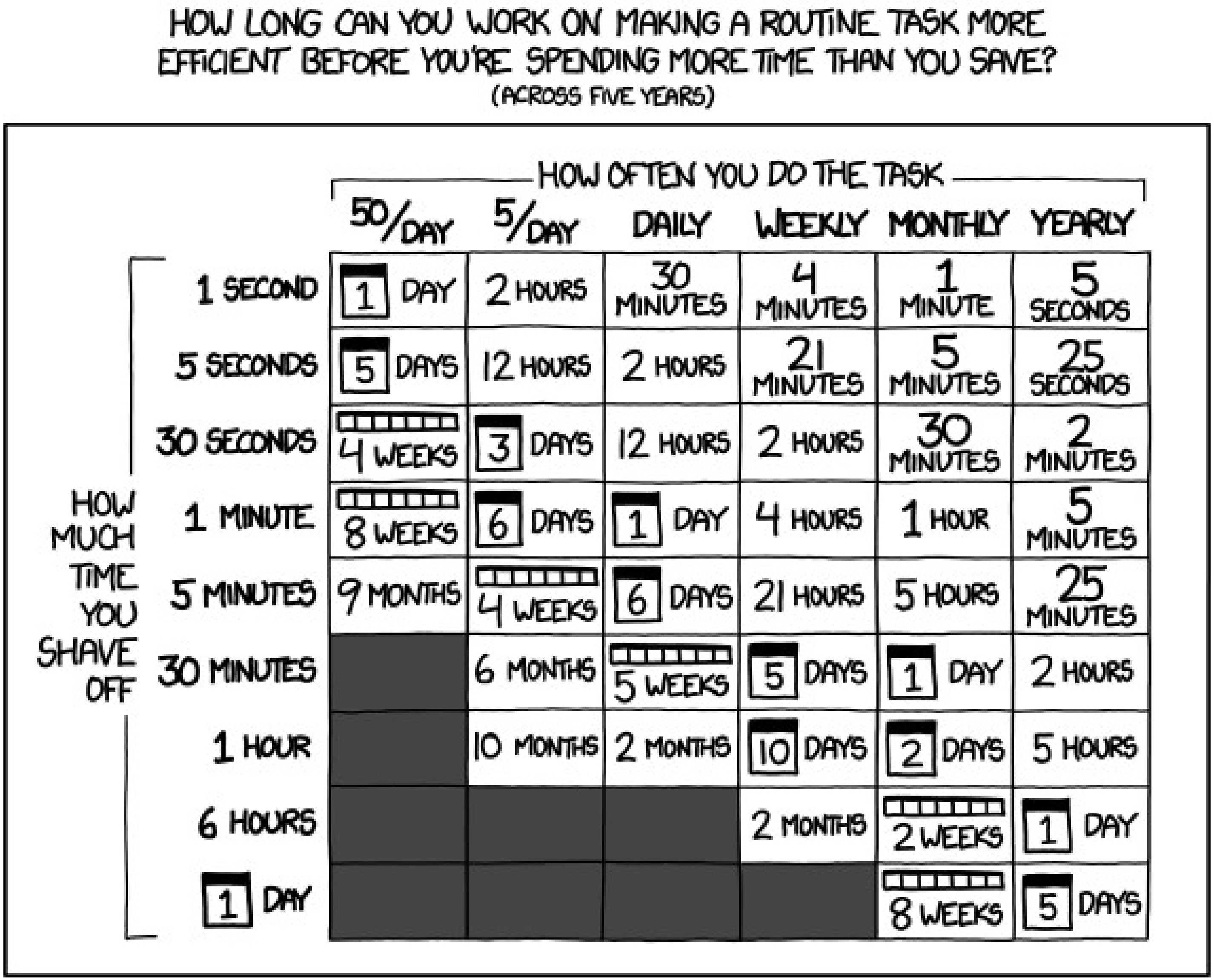
Large quantities produced

- O(100M) for MiNNLO validation
- O(10B) for EW uncertainties

Data tier	Size (kB)
RAW	1000
Gen	<50
SIM	1000
DIGI	3000
RECO(SIM)	3000
AOD(SIM)	400
MiniAOD(SIM)	50
NanoAOD(SIM)	2
NanoGEN	0.4

Analysis data formats


Primary
data
Simulation
Central
production
run on Grid
~weeks/
months
MiniAOD
CMSSW
~days
CMSSW
~days
Nano
AOD
Nano
GEN
Private
production
run on Grid
Ran 6-7 times in 5 years
RDataFrame
~1-2 h
Boost
Histograms
Corrections &
systematics
Control plots
& diagnostics
HDF5 writer
~2 min
hdf5 tensor
Tensorflow fit
~5-10 min
Fitresults
Values & uncertainties
Pulls & Constraints
Prefit/postfit plots
Top level analysis
framework
WRemnants
• Covered by CI
Ran on daily basis (>1000 times)



Where we started

High performance computing machines

Custom analysis framework executed locally

- No resubmission of failed jobs/ merging of jobs etc.
- Direct feedback on progress

Run on single high performance machine

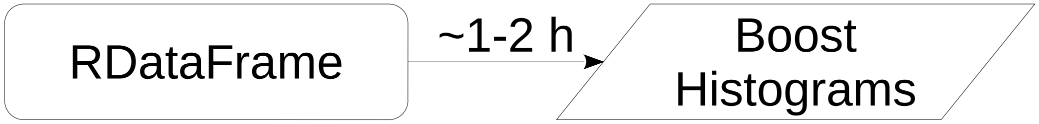
- Reading/writing on fast NVMe SSDs
  - Local or via network interface 100Gbit/s
- Reading from local CERN eos via xrootd
  - Network interface 100Gbit/s

	CERN	MIT/Pisa
CPU	2 x EPYC 7702	2 x EPYC 9654
cores	128	192
threads	256	384
memory	1TB	1.5/2TB

Possible upgrade for the future

- EPYC Turin machine with 384 cores/ 768 threads

ROOT RDataFrame

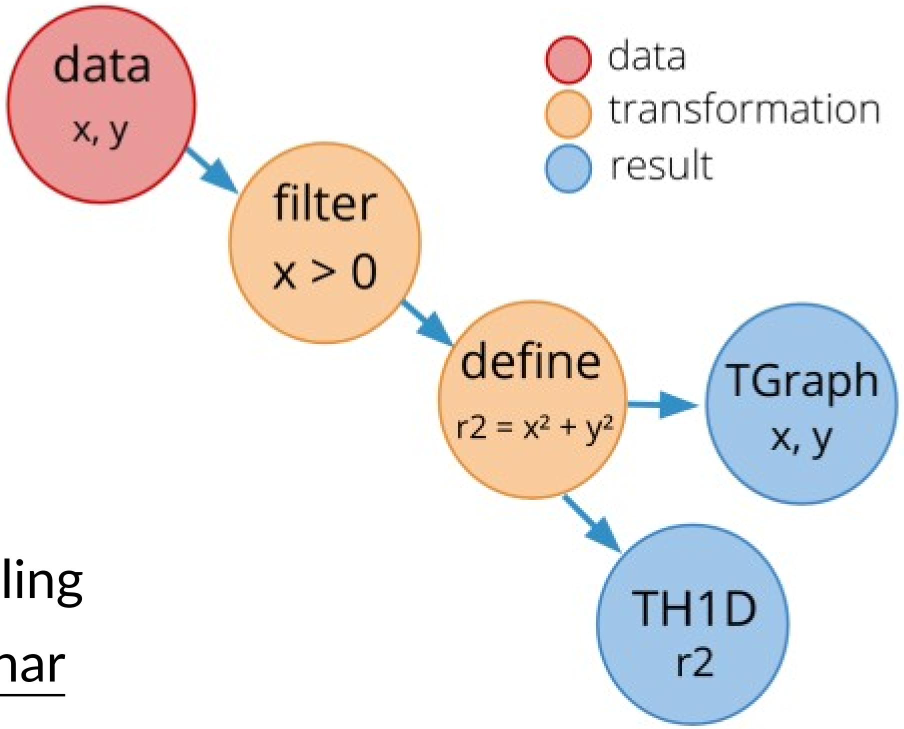


Select objects, filter events, fill histograms

- Pythonic, declarative, graph-style analysis

```
from ROOT import RDataFrame
df  = RDataFrame(dataset);
df2 = df.Filter("x > 0")
        .Define("r2", "x*x + y*y");

rHist = df2.Histo1D("r2");

g = df2.Graph("x","y")
```

Internal computation graph

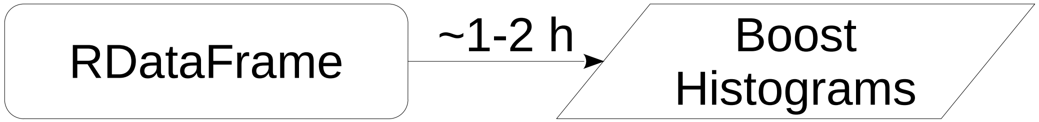


- Lazy execution: perform all operations in parallelized single event loop
- Executed on local machine
  - Plan to explore distRDF for multi-node scaling
- See RDF reference, documentation, EP seminar

Many optimizations conducted to ensure good thread scaling

- Now fully integrated in ROOT

ROOT RDataFrame



Critical parts in c++

- Functions: e.g. check if reco muon has a match to any gen muon

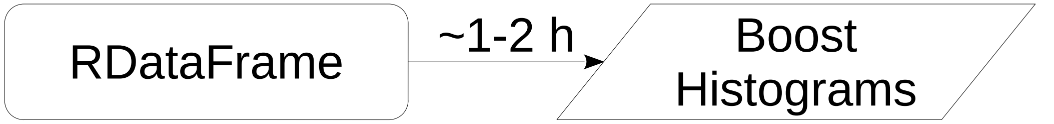
```
bool hasMatchDR2(const float& eta, const float& phi, const Vec_f& vec_eta, const Vec_f& vec_phi, const float dr2 = 0.09) {

  for (unsigned int jvec = 0; jvec < vec_eta.size(); ++jvec) {
    if (deltaR2(eta, phi, vec_eta[jvec], vec_phi[jvec]) < dr2) return true;
  }
  return false;

}
```

- Compiled at runtime using cling jitting
- And in python

```
df = df.Filter("wrem::hasMatchDR2(goodMuons_eta0,goodMuons_phi0,GenPart_eta[postfsrMuons],GenPart_phi[postfsrMuons],0.09)")
```

- Other examples much more complex – but follow same logic
- We also tried Numba, but found less efficient and not more convenient

ROOT RDataFrame



Critical parts in c++

- Helpers – classes that contain histograms with corrections and functions to apply them: e.g. reweight pileup spectrum in MC to the one in data

```
class pileup_helper {
public:

  pileup_helper(const TH1D &puweights) :
    puweights_(make_shared_TH1<const TH1D>(puweights)) {}

  // returns the pileup weight
  double operator() (float nTrueInt) const {
    return puweights_->GetBinContent(puweights_->FindFixBin(nTrueInt));
  }


private:
  std::shared_ptr<const TH1D> puweights_;
};
```

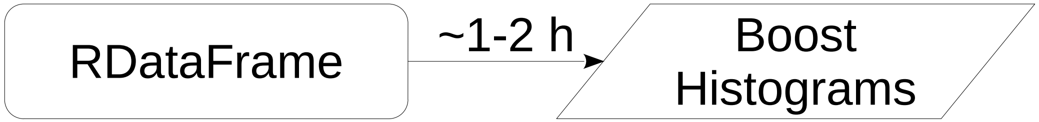
- And in python

```
helper = ROOT.wrem.pileup_helper(puweights)
```

```
df = df.Define("weight_pu", pileup_helper, ["Pileup_nTrueInt"])
```

- Other examples much more complex – but follow same logic

ROOT RDataFrame



Critical parts in c++

- Often templated (e.g. for histogram bins)
- Also using Eigen and tensorflow c++ libraries

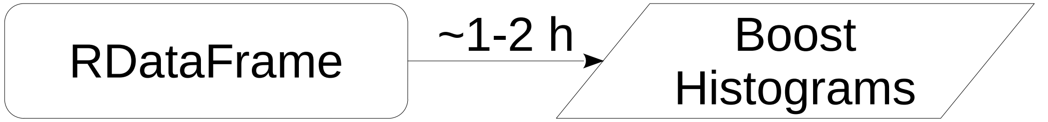
```
template <std::size_t NEtaBins>
class muon_prefiring_helper_stat {

public:

  static constexpr std::size_t NVar = NEtaBins + 1;
  using value_type = Eigen::TensorFixedSize<double, Eigen::Sizes<NVar, 2>>;

  muon_prefiring_helper_stat(const muon_prefiring_helper &other) :
    parameters_(other.parameters()), hotspot_parameters_(other.hotspot_parameters()) {}

  value_type operator() (const Vec_f& eta, const Vec_f& pt, const Vec_f& phi, const Vec_i& charge, const Vec_b& looseId, double nominal_weight = 1.0) const {
        [...]

      return res;
    }

  private:
    std::shared_ptr<const TH2D> parameters_;
    std::shared_ptr<const TH2D> hotspot_parameters_;

  };
```

- And in python

```
helper_stat = ROOT.wrem.muon_prefiring_helper_stat[netabins](helper)
```

```
df = df.Define("weight_newMuonPrefiringSF", muon_prefiring_helper, ["Muon_correctedEta", "Muon_correctedPt", "Muon_correctedPhi", "Muon_correctedCharge", "Muon_looseId"])
```

Histograms

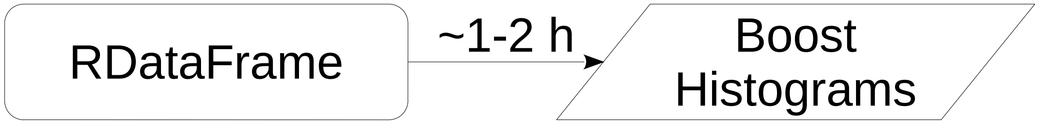


Strategy to perform computations on histograms later in analysis chain

- Allows for more flexibility
- E.g. data-driven nonprompt background prediction
- Nominal histogram is 5D

Axis	Bins
p_T^{μ}	30
η^{μ}	48
q^{μ}	2
I_{rel}^{μ}	2
m_T^{W}	3
All	17,280

Histograms



Strategy to perform computations on histograms later in analysis chain

- Allows for more flexibility
- E.g. data-driven nonprompt background prediction
- Nominal histogram is 5D
- Largest histograms with 8D and 20M bins
  - For efficiency scale factor 2D smoothed in p_T and u_T
- ~same histograms for 16 processes

Axis	Bins
p_T^μ	30
η^μ	48
q^μ	2
I_{rel}^μ	2
m_T^W	3
var. η^μ	48
var. q^μ	2
eig. vec.	12
All	19,906,560
All (w/ flow)	358,400,000

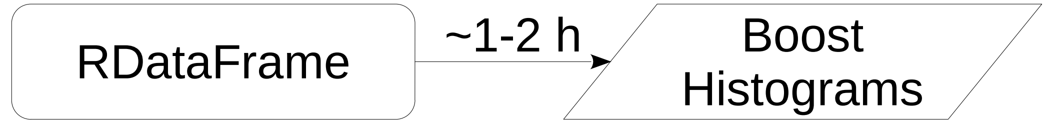
Significant memory consumption

→ For largest histogram: 2.5GB

→ For all: 13GB

Gets much worse if flow bins can't be disabled (as in root histograms)

Boost histograms



Previously: one root histogram copy for each thread

- But large memory consumption was a showstopper
- Long merging time when adding up at the end

Solution: use std:atomic<double> with c++ boost histograms

- All threads write in same histogram
- But can’t use python binding directly … (cppyy vs. pybind 11)

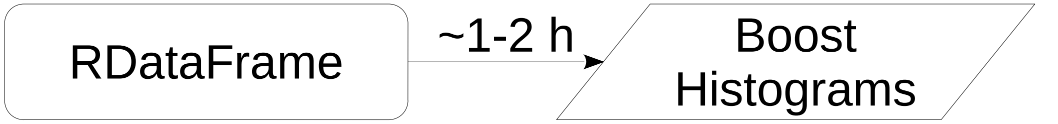
Custom copy conversion into python boost histograms

- Arbitrary number of axes
- Configurable underflow/overflow bins
- Convenient (numpy like) indexing/ manipulation

Histograms stored with pickle

- Using proxies dictionary in .hdf5 to allow lazy loading (code)
- Including meta data (e.g. number of processed events, cross section/luminosity, command, …)

Tensor axes



All systematic uncertainties represented by event weight variations

Traditionally one histogram per variation

- e.g. NNPDF provides 101 alternate PDF weights → 101 histograms

Better: a single histogram with an additional axis

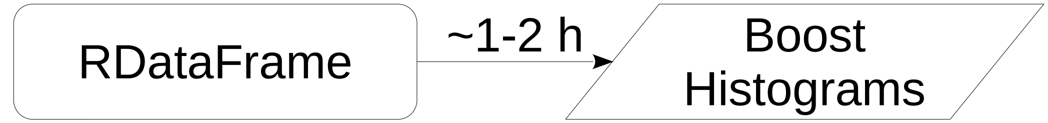
Even better: fill full array/tensor at once, only do bin lookup once

- Using Eigen tensors
- Arbitrary number of dimensions

Atomic boost histograms and tensor axes implemented in narf submodule

- More details given at ROOT Users Workshop 2022: link
- Not currently integrated in root; similar functionality in RHistogram?
    - Interest also from outside W mass analysis team

Histogram benchmark



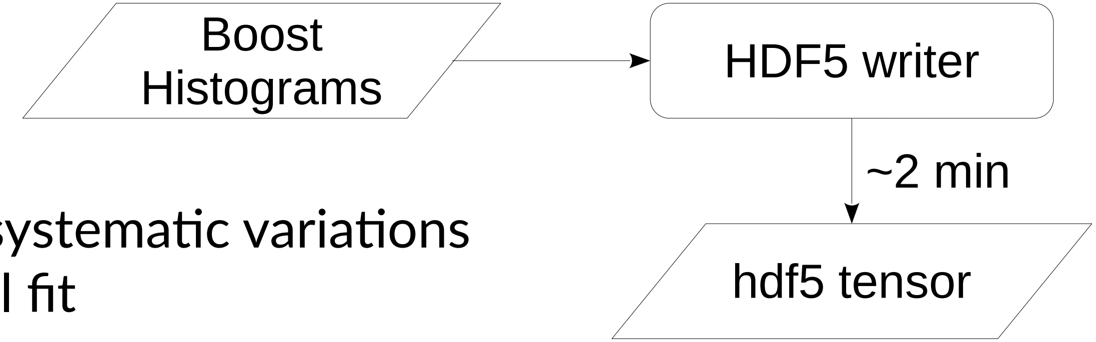
Using 400M events of CMS NanoAOD (W→μν) and filling 10 copies of pdf variation histograms

256 threads (2 EPYC 7702)

Hist Type	**Hist Config**	**Evt. Loop**	**Total**	**CPUEff**	**RSS**
ROOT THnD	10 x 103 x 5D	59m39s	74m05s	0.74	400GB
ROOT THnD	10 x 6D	7m54s	25m09s	0.27	405GB
Boost ("sta")	10 x 6D	7m07s	7m17s	0.90	9GB
Boost ("sta")	10 x (5D + 1-tensor)	1m54s	2m04s	0.81	9GB
Boost ("sta")	1 x (5D + 2-tensor)	1m32s	1m42s	0.77	9GB

- Root histograms slowed down by merging step
- Memory much lower with atomic accumulation
- Factor ~4 time reduction with tensor axes due to reduced lookup
- Some additional subtleties related to cash locality

HDF5 writer



Histograms with data, prediction, and systematic variations need to be casted into a tensor for final fit

- Purely python based (boost histograms, numpy, …)
- Flexibility & efficient implementation is essential
- Perform selections/ accumulations/ other computations on histograms
  - Signal selection
  - Data-driven nonprompt background estimation
    - Smoothing “on-the-fly” using least squares (code)
  - Modify systematic variations e.g. decorrelating/ combining

Sparse tensor implementation for unfolding

Binned profile maximum likelihood fit

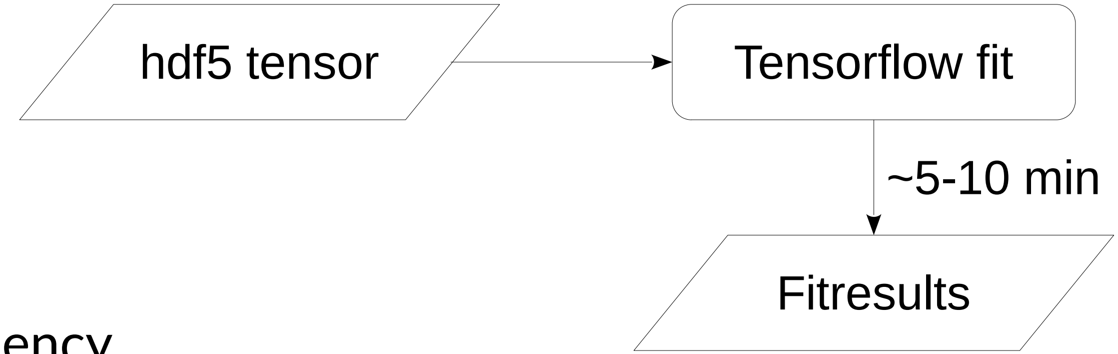
Log likelihood from Poisson distributed bin-by-bin event numbers

$$L = \sum_{ibin} \left(-n_{ibin}^{obs} \ln n_{ibin}^{exp} + n_{ibin}^{exp} \right) + \frac{1}{2} \sum_{ksyst} \left(\theta_{ksyst} - \theta_{ksyst}^{0} \right)^2$$

$$n_{ibin}^{exp} = \sum_{jproc} \mu_{jproc} \, n_{ibin,jproc}^{exp} \prod_{ksyst} \kappa_{ibin,jproc,ksyst}^{\theta_{ksyst}}$$

- Gaussian constraint nuisance parameters θ for systematic uncertainties
- Signal strength modifier *μ*
- Systematic variations in 3D tensor *κ*

Tensorflow fit



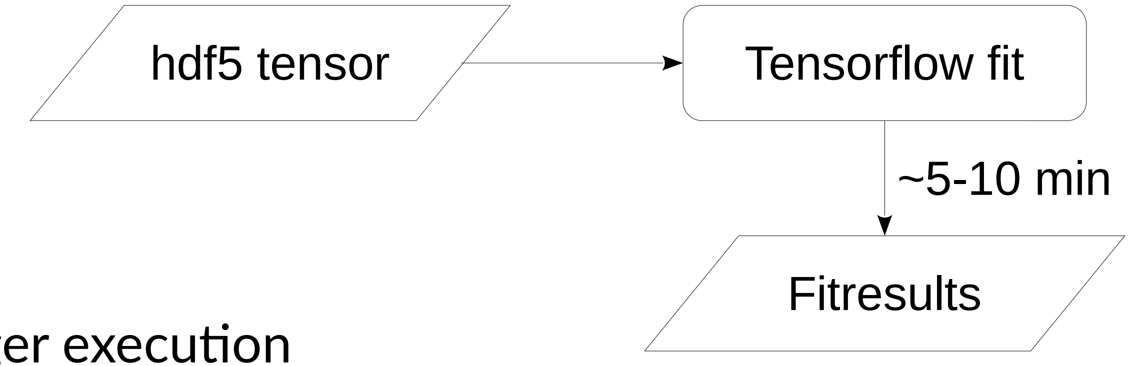
RooFit via minuit insufficient

- Limited numerical stability and efficiency
  - E.g. can not be parallelized

Tensorflow library with automatic gradient computation via back propagation for minimization:

- Quasi Newton trust region based minimizer to reliably find global minimum
  - Native tensorflow implementation; algorithm based on arXiv:1506.07222
- Fast, numerically accurate, stable
- Parallelized vector processing units and/or multiple threads
- Sparse tensor implementation to minimize memory consumption (if response matrix is close-to-diagonal, e.g. leptonic observables)
- Implemented in combineTF, see also PyHEP 2020: link

Tensorflow 2 fit



Re-written in Tensorflow 2:

- More developer-friendly due to eager execution
- Almost feature complete combineTF2 implementation
- More efficient computatoin of hessian and hessian vector products
- Trust-krylov minimizer from SciPy, computing the gradient and hessian-vector product in tensorflow 2
  - I.e. not using quasi-newton methods as in the combineTF1 case

Benchmark using MIT machine

- CPU: EPYC 9654
- GPU: Nvidia A30

	fit	fit + covariance
CombineTF1 CPU	1m49s	3m48s
CombineTF2 CPU	34s	47s
CombineTF2 GPU	36s	39s

GPU “only” used to calculate the gradient/hessian/hessian-vector-product

Continuous integration

Common framework among all analyzers

- Sharing as much code as possible among different efforts
- Reuse existing code, find/avoid bugs, save time
- Quickly developed with O(10) contributors, now at >500 pull requests (PRs)

However

- Updates often unintentionally affected other parts
  - Framework was constantly broken
- Sometimes not clear where certain changes came from

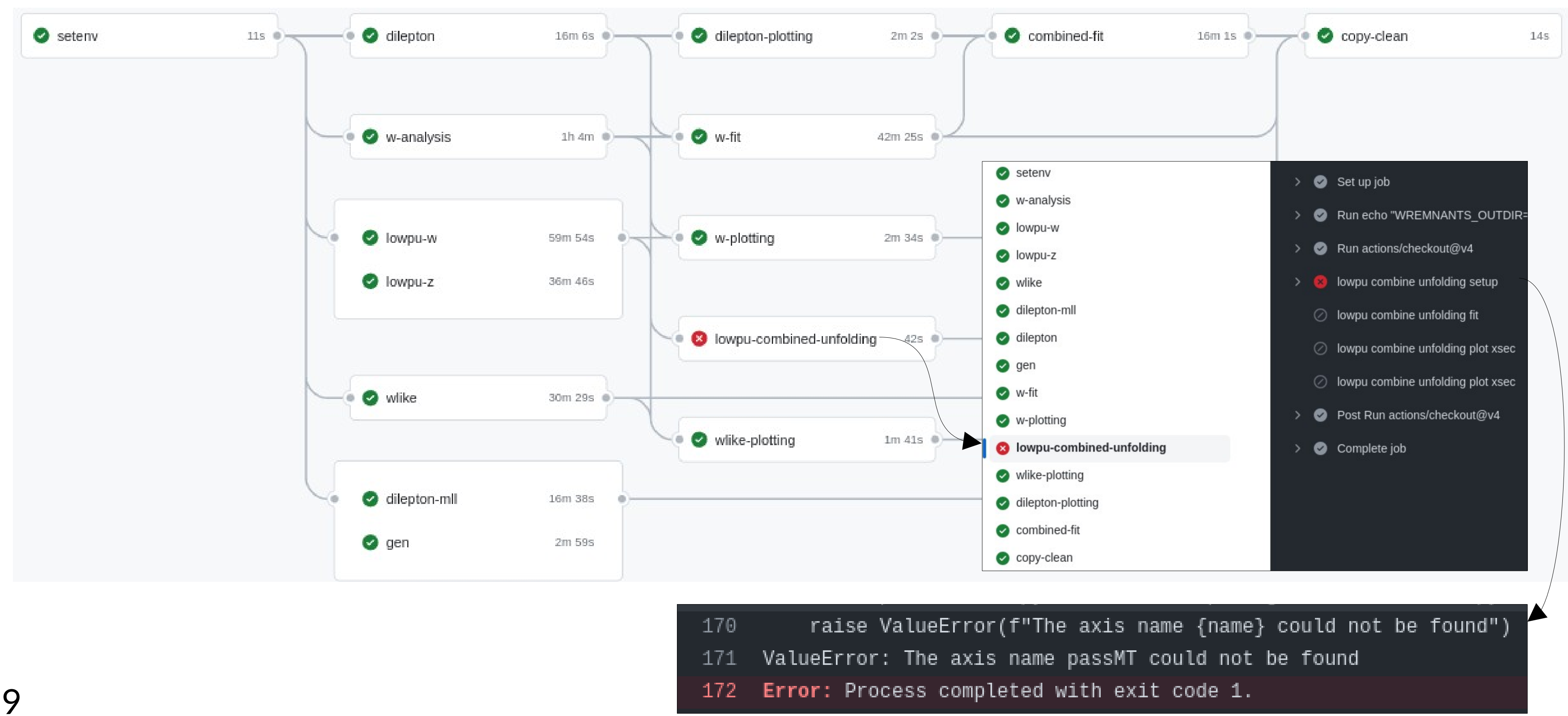
Solution → GitHub actions: platform for automate developer workflows

- Use continuous integration and deployment (CI/CD) pipeline
- Same tool as used for code development instead of third party integration
- Slim and easily to set up and manage (compared to e.g. Jenkins)

Github CI workflow

Different analysis chains implemented

- Independent jobs run in parallel, each job contains a set of steps
- Different arguments for plotting/ fitting for good code coverage
- Investigate failed jobs directly in Github actions



Github CI workflow

```
on:
  pull_request:
    branches: [ main ]            1)
  schedule:
    - cron: '30 5 * * 1-5'        2)
    - cron: '0 1 * * 2,4,6'       3)
  workflow_dispatch:              4)
```

Running full analysis chain (code)

1) For each PR on reduced set of files (~1%)
2) Scheduled each morning on reduced set of files (~1%) as reference for PR
3) Scheduled 3 times a week on (1:1) data:MC files to backtrack changes
   - All output files (e.g. histograms) stored on EOS for later use
   - Separate workflow to delete old files
4) Workflow dispatch on (1:1) data:MC files to manually run on chosen branch
   - To test a new feature (e.g. apply new nominal calibration/correction)

In the process of adding code checks

- Run in CI and as pre-commit hooks
- Syntax checks for python, c++, yaml, json files
- Linters: Black, Flake8, isort

Everything blinded

Github CI infrastructure

Maintained via service account with CMS access and eos area

- Self hosted runners to easy access resources and execute code on the CERN high-performance analysis machine used for this analysis

```
setenv:
  runs-on: [self-hosted, linux, x64]
```

- Repository with sub modules checked out including large file storage (lfs) support

```
    steps:
      - uses: actions/checkout@v4
        with:
          submodules: 'recursive'
          lfs: 'true'
```

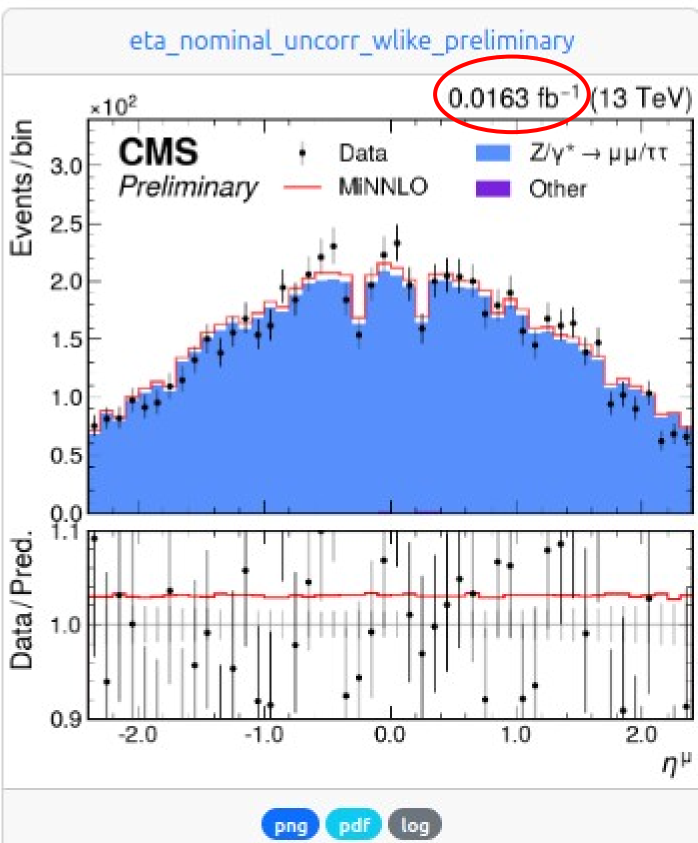
Network authentication via Kerberos, key stored in local keytab file

```
      - name: setup kerberos
        run: |
          kinit -kt ~/private/.keytab cmsmwbot@CERN.CH
          klist -k -t -e ~/private/.keytab
          klist
          echo "xrdfs root://eosuser.cern.ch// ls $EOS_DIR"
          xrdfs root://eosuser.cern.ch// ls $EOS_DIR

      - name: setup kerberos within singularity image
        run: |
          scripts/ci/run_with_singularity.sh kinit -kt ~/private/.keytab cmsmwbot@CERN.CH
          scripts/ci/run_with_singularity.sh klist -k -t -e ~/private/.keytab
          scripts/ci/run_with_singularity.sh klist
          echo "xrdfs root://eoscms.cern.ch// ls $EOS_DATA_DIR"
          scripts/ci/run_with_singularity.sh xrdfs root://eoscms.cern.ch// ls $EOS_DATA_DIR
```

Webpage support

Results initially created in local temporary folder

- Copied via xrdcp to CMS protected webpage
- CMS centrally maintained plot browser
- Automatic lumi scaling for using subset of data files



/ WMassAnalysis / PRValidation

Directories

- Archive
- Dispatch
- PR2XX
- PR3XX
- PR4XX
- PR500
- PR501

[...]

- PR553
- PR554
- ReferenceRuns
- ScheduledBuilds

Webpage support

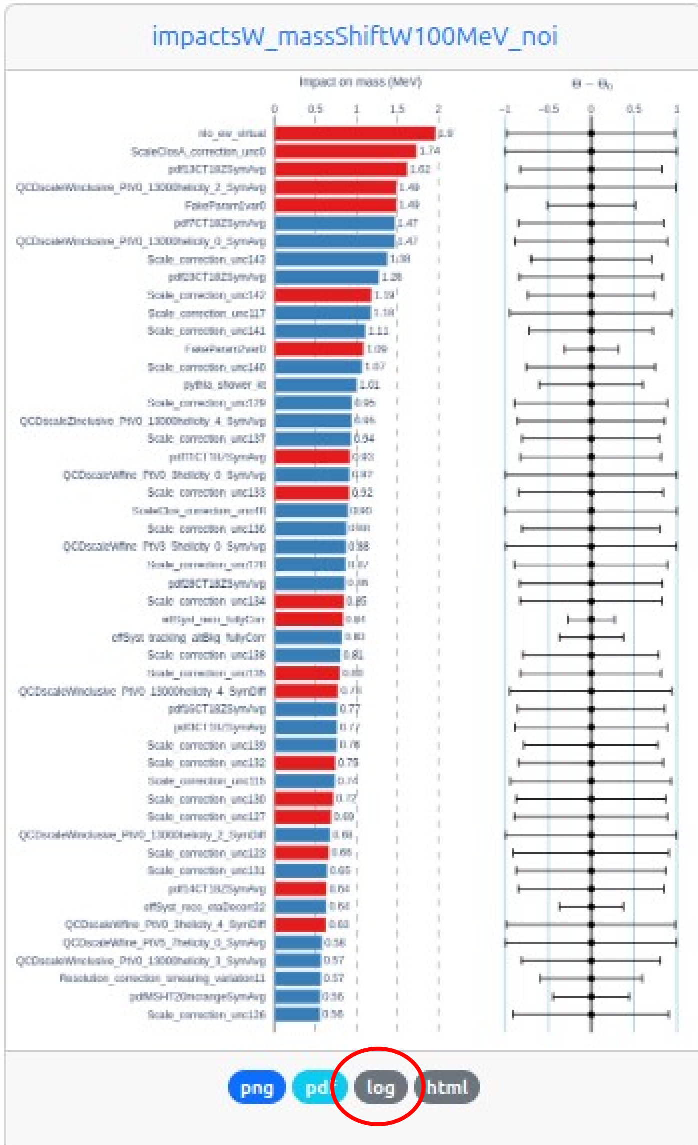
/ WMassAnalysis / PRValidation

Directories

- Archive
- Dispatch
- PR2XX
- PR3XX
- PR4XX
- PR500
- PR501

[…]

- PR553
- PR554
- ReferenceRuns
- ScheduledBuilds



Plots for nuisance parameter pulls, constraints, and impacts produce via plotly

- Available in interactive .html
- See all O(1000) nuisances with more digits

Each plot is produced with a .log file

.log files

```
--------------------------------------------------------------------------------
Script called at 2024-10-01 15:56:53.608388
The command was: scripts/plotting/postfitPlots.py
'/scratch/dwalter/CombineStudies/test/ZMassDilepton_ptll_yll/fitresults_123456789.root' --
legCols 1 --eoscp -f '241001_test' --yscale '1.25'
--------------------------------------------------------------------------------
```

Check exact event yields

```
Yield information for Stacked processes
--------------------------------------------------------------------------------
                        Process       Yield  Uncertainty
0               $\gamma$-induced    1796.97        94.01
1  Z/$\gamma^{\star}\to\tau\tau$    1582.57        77.61
2                          Other    1630.27        14.59
3    Z/$\gamma^{\star}\to\mu\mu$ 1931915.83       336.74

Yield information for Unstacked processes
--------------------------------------------------------------------------------
     Process         Yield  Uncertainty
0       Data  1936925.64      2547.78
1  Inclusive  1936925.64       313.63

===> Sum unstacked to data is 100.00%
--------------------------------------------------------------------------------
```

Command chain used to produce the plot

→ Look up how to run specific scripts

```
Meta info from input file AnalysisOutput

--------------------------------------------------------------------------------
{
     "time": "2024-10-01 15:39:52.107792",
     "command": "scripts/combine/setupCombine.py -i
'/scratch/dwalter/results_histmaker/test/mz_dilepton.hdf5' --fitvar 'ptll-yll' --lumiScale
100 --realData -o '/scratch/dwalter/CombineStudies/test'",
     "args": {
          "outfolder": "/scratch/dwalter/CombineStudies/test",
          "inputFile": [
               "/scratch/dwalter/results_histmaker/test/mz_dilepton.hdf5"
          ],
          "postfix": null,
          "verbose": 3,
```

[...]

Git commit hash

```
     "git_hash": "\"4713c27278391e1df49f86834c6d122cff8beba5\"
",
```

Local untracked changes

→ Each plot is reproducible

```
     "git_diff": "diff --git a/scripts/combine/saturatedGOF.py
b/scripts/combine/saturatedGOF.py
index 1b9fb2e9..1df140a8 100644
--- a/scripts/combine/saturatedGOF.py
+++ b/scripts/combine/saturatedGOF.py
@@ -15,7 +15,7 @@ tree.GetEntry(0)

 fitresult_h5py = combinetf_input.get_fitresult(args.infile.replace(\".root\",\".hdf5\"))
 meta = ioutils.pickle_load_h5py(fitresult_h5py[\"meta\"])
-nbins = sum([np.product([len(a) for a in info[\"axes\"]]) for info in
meta[\"channel_info\"].values()])
+nbins = sum([np.prod([len(a) for a in info[\"axes\"]]) for info in
meta[\"channel_info\"].values()])
 ndf = nbins - tree.ndofpartial
```

Many interesting features not discussed today

Other analysis ingredients

- Efficiencies
  - Using tag and probe fits, smoothing of scale factors in 1D/2D
- Helicity cross section corrections & uncertainties
  - Based on Eigen
- Muon calibration
  - Object to event weight variations via CDF transform
- Recoil calibration
  - Functional fit based on JAX, evaluation with tensorflow lite c++
- …

Summary

Increasing amount of data opens new opportunities

- Software developments must be ahead to fully exploit potential

Fast analysis turnaround was essential for this complex measurement

- RDF provides a convenient and efficient library
  - Initially showstoppers observed in scaling
  - Extensive work on critical parts to improve RDF and histogram implementation
- Full analysis runs in ~hours

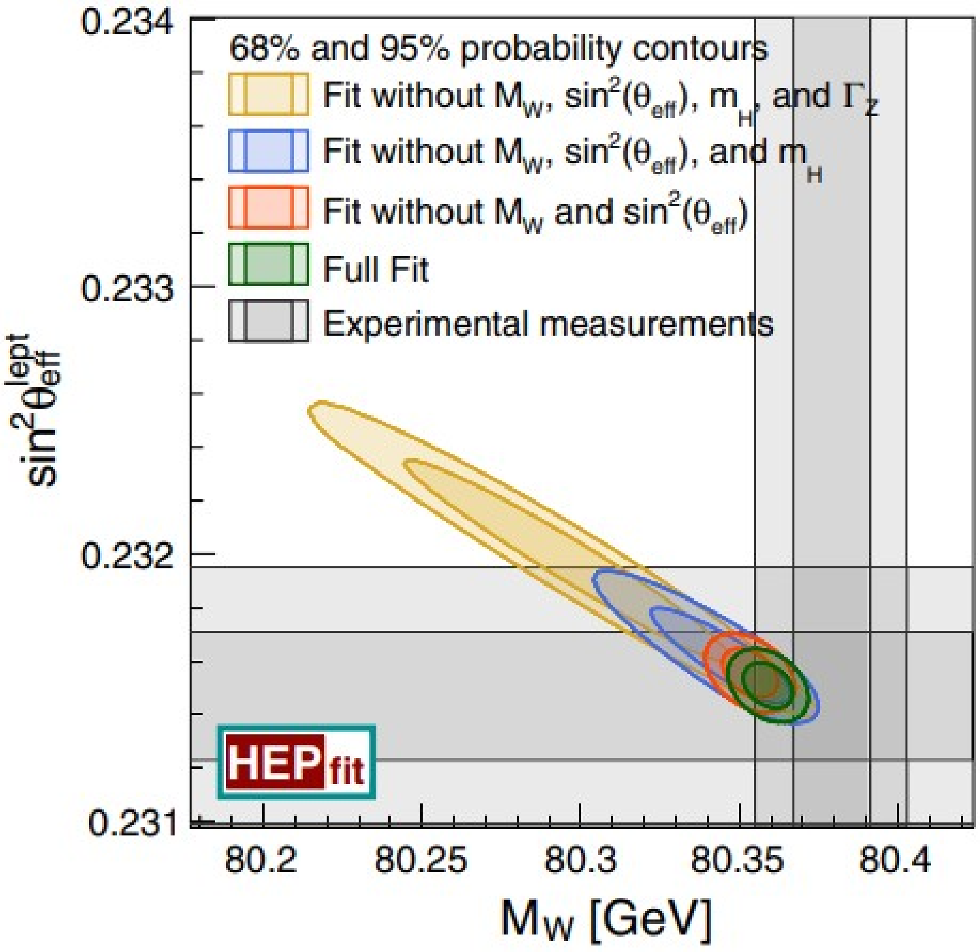
Challenging collaborative work with increasing number of contributors

- Github CI/CD pipeline has turned out to be extremely useful
- Time savings in PR reviews, spot/avoid bugs, backtrack changes
- Always ensure working implementation for different analyses/ configurations

Many areas identified for further improvements

Backup

Precision measurements of standard model parameters provide opportunity to over constrain the theory and pose stringent tests



Indirect prediction (~6MeV) more precise than direct measurement (~10MeV) and in tension (CDF)

→ Call for more precise measurements

Lumitools

Automatic computation of integrated luminosity of processed data

- CMS data is organized by fill, run, luminosity block (~24s)
  - Use .csv file containing integrated luminosity information
  - Provided by the CMS BRIL group
- Processed with RDataFrame, read non-ROOT data
- Guarantees consistent luminosity calculation
- Convenient for running on subset of data

Implemented in lumitools

- Could be used standalone

Histogram benchmark

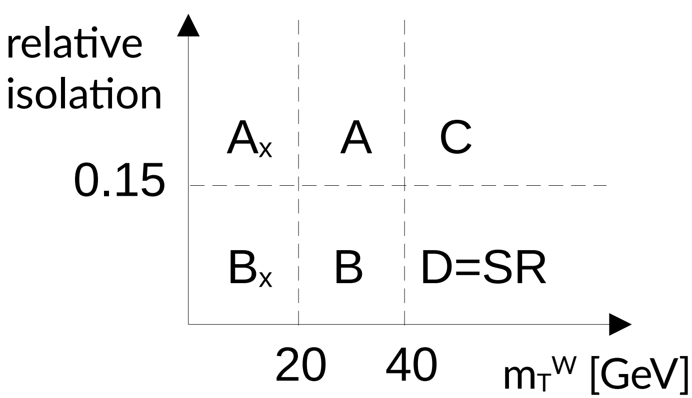
Hist Type	Hist Config	Evt. Loop	Total	CPUEff	RSS
ROOT THnD	10 x 103 x 5D	59m39s	74m05s	0.74	400GB
ROOT THnD	10 x 6D back	7m54s	25m09s	0.27	405GB
ROOT THnD	10 x 6D front	13m52s	30m27s	0.42	406GB
Boost ("sta")	10 x 6D back	7m07s	7m17s	0.90	9GB
Boost ("sta")	10 x 6D front	3m22s	3m33s	0.86	9GB
Boost ("sta")	10 x (5D + 1-tensor)	1m54s	2m04s	0.81	9GB
Boost ("sta")	1 x (5D + 2-tensor)	1m32s	1m42s	0.77	9GB

- In the tensor/array weight-case the weights for the different systematic idxs are contiguous in memory by construction
- In the N+1-d histogram case it depends on the array ordering
- TH1/2/3 and boost-histograms have fortran array ordering → systematic idx axis is best at the front
- THn has C array ordering → systematic idx axis is best at the back
- The difference is about a factor of 2 for both root and boost hists (but still > 50% additional gain from tensor filling)
- Largely accounted simply by skipping the extra FDIVs needed for redundant value-to-index conversion for the 5 axes

QCD multijet background estimation

Estimated from data using extended ABCD method

- Prediction from yields in sideband regions in bins of high relative isolation and low m_T^W
- Prompt background in sideband region subtracted from simulation
  - Repeated for each systematic variation ~O(1000) times
- Evaluated in fine bins in p_T^μ, η^μ, q^μ

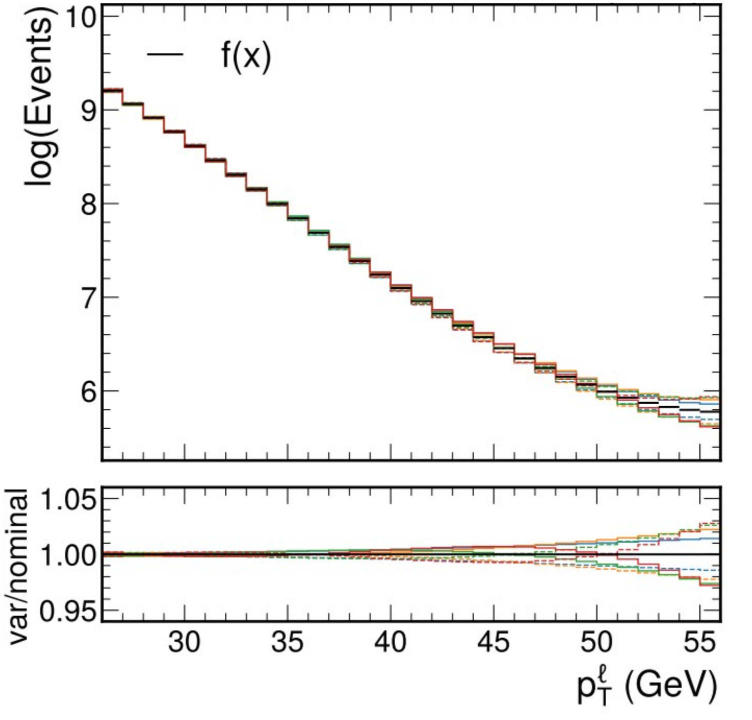


$$D = C \cdot \underbrace{\frac{A_x B^2}{B_x A^2}}_{\text{fakerate}}$$

QCD multijet background estimation

Smoothing each sideband region in p_T^{μ} with exponential "on-the-fly"

- Maintain good statistical properties
- Smoothing in 5 regions, 96 bins for η^{μ}, q^{μ}
- Repeated for O(1000) systematic variations
- Robust and efficient calculation required
  - Use polynomial in log space
  - Analytic solutions using least squares



$$f_i(p_{\mathrm{T}}) = \mathrm{e}^{P_i(p_{\mathrm{T}})}$$

$$f_{\mathrm{D}}(p_{\mathrm{T}}) = \mathrm{e}^{\sum_i w_i P_i(p_{\mathrm{T}})}$$

  - Systematic uncertainties from eigenvector decomposition
  - Everything done in ~seconds

More complex procedures tested

  - E.g. using integrated Bernstein polynomials with nnls to enforce monotonicity

How to measure the W mass with 10 MeV uncertainty

EP-IT Data Science Seminars

16 October 2024, CERN

David Walter (CERN) on behalf of the CMS Collaboration

Introduction

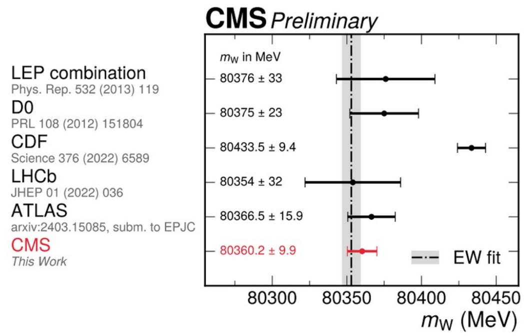
Analysis presented in LPC seminar last month

- Document: [CMS-PAS-SMP-23-002]

First measurement of m_W from CMS

- Most precise at LHC
- In agreement with the SM
  but in tension with CDF

This seminar will focus
on the technical aspects



Use 16.8 fb^{-1} pp collision data at $\sqrt{s}$=13TeV

HL-LHC

Large inclusive W cross section

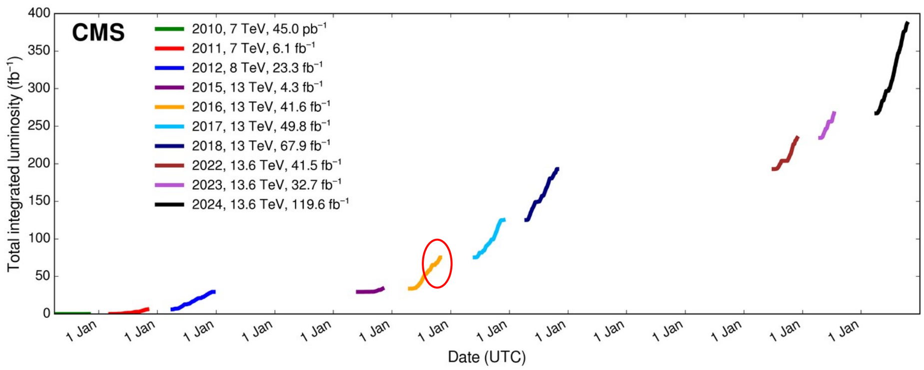
- 300M data and 4B MC events (4 times MC statistical power)

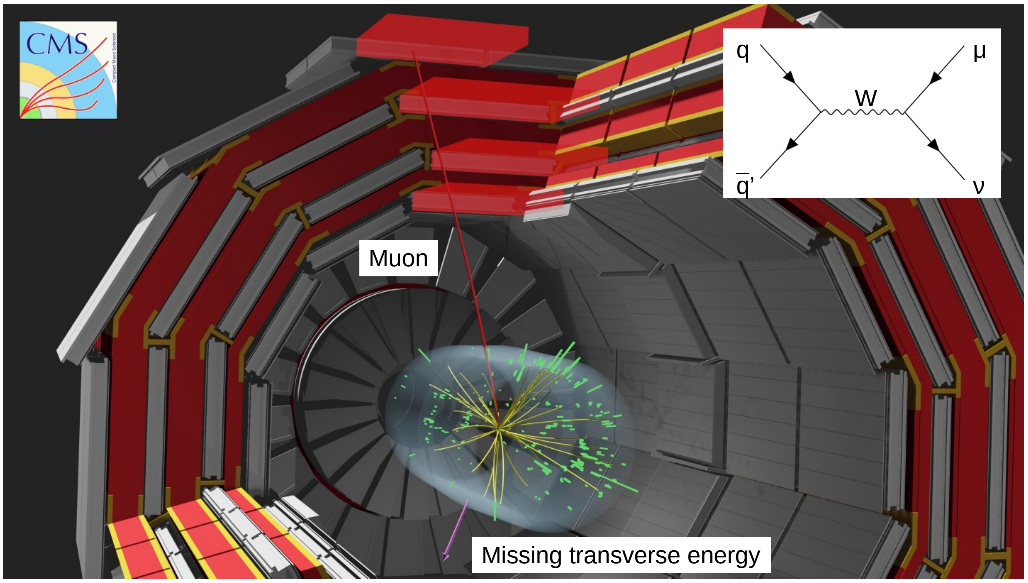
Largest dataset used for W boson mass analysis

- Opportunity to exploit multi dimensional information
- Challenging data processing

Much more data available now and in the years to come

→ Software developments have to keep up with technical challenges





The measurement is performed using the muon kinematics only

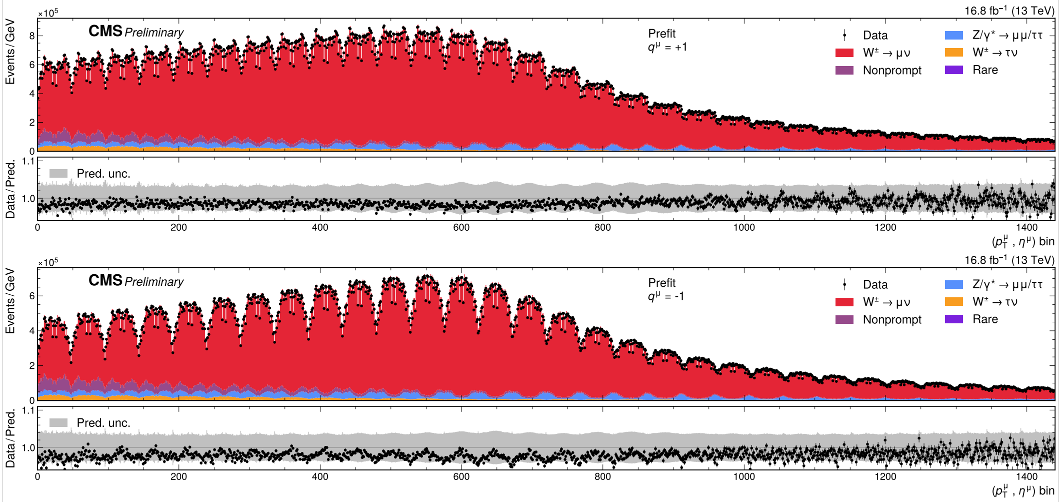
4

How we measure the W boson mass

Strategy to use large data sample and constrain theory uncertainties in-situ

Profile likelihood fit to single muon p_T η, charge distribution

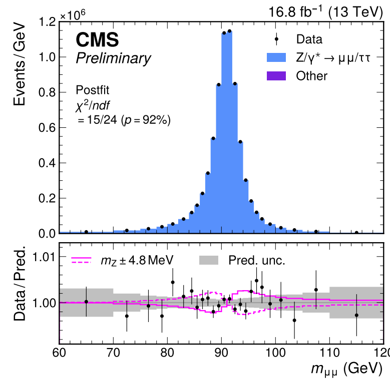
- 2880 bins

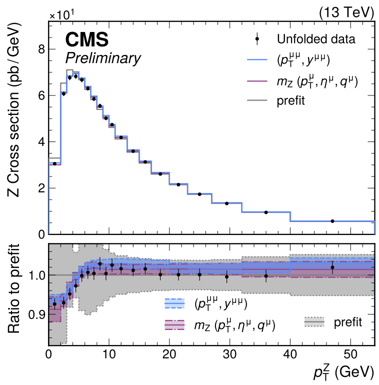


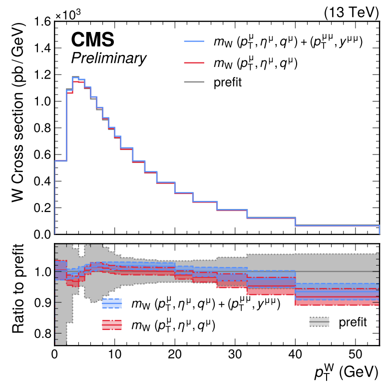
Multiple analyses in one

- Z dilepton m_{ll}, p_T^Z-y^Z, W-like
- Unfolding
- Helicity cross section fit
- Generator studies
- ...

Different configurations, including combined fits







6

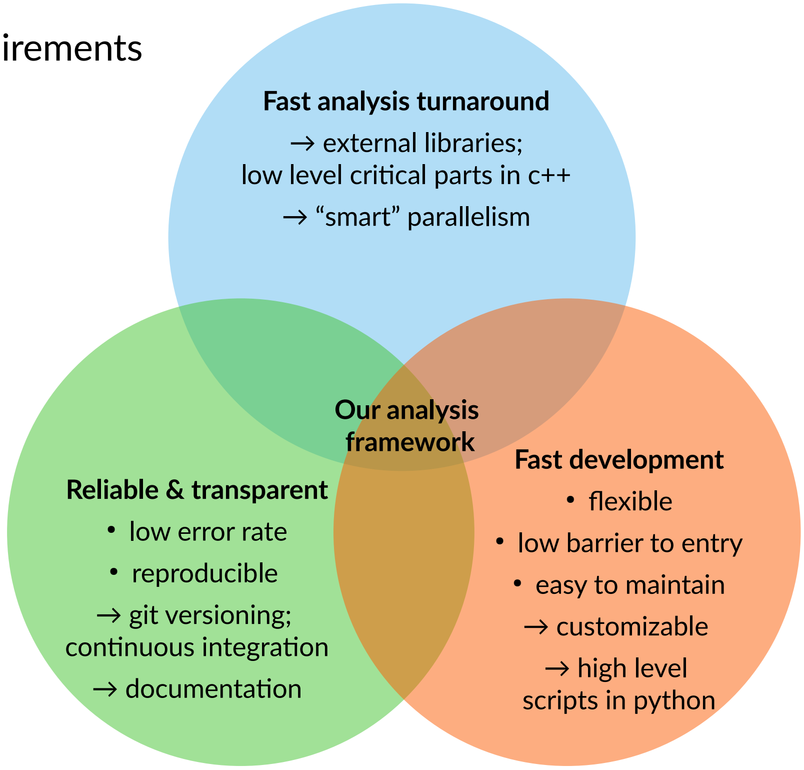
Precise treatment of uncertainties requires large amount of variations

- O(1000) parameters in single fit

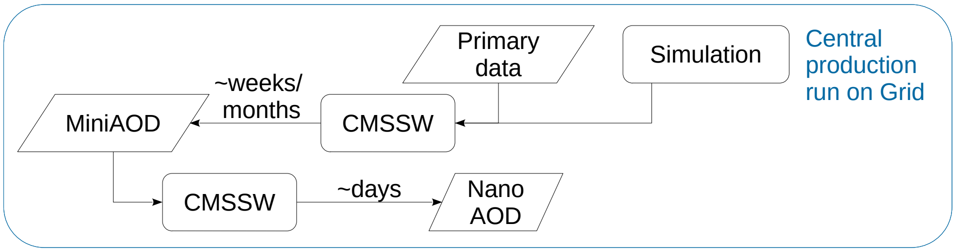
<table>
<tr><th>Systematic uncertainties</th><th>W-like m_Z</th><th>m_W</th></tr>
<tr><td>Muon efficiency</td><td>3127</td><td>3658</td></tr>
<tr><td>Muon eff. veto</td><td>–</td><td>531</td></tr>
<tr><td>Muon eff. syst.</td><td colspan="2">343</td></tr>
<tr><td>Muon eff. stat.</td><td colspan="2">2784</td></tr>
<tr><td>Nonprompt background</td><td>–</td><td>387</td></tr>
<tr><td>Prompt background</td><td>2</td><td>3</td></tr>
<tr><td>Muon momentum scale</td><td colspan="2">338</td></tr>
<tr><td>L1 prefire</td><td colspan="2">14</td></tr>
<tr><td>Luminosity</td><td colspan="2">1</td></tr>
<tr><td>PDF (CT18Z)</td><td colspan="2">60</td></tr>
<tr><td>Angular coefficients</td><td>177</td><td>353</td></tr>
<tr><td>W MiNNLO$_{PS}$ μ_F, μ_R</td><td>–</td><td>176</td></tr>
<tr><td>Z MiNNLO$_{PS}$ μ_F, μ_R</td><td colspan="2">176</td></tr>
<tr><td>PYTHIA shower k_T</td><td colspan="2">1</td></tr>
<tr><td>p_T^V modeling</td><td>22</td><td>32</td></tr>
<tr><td>Nonperturbative</td><td>4</td><td>10</td></tr>
<tr><td>Perturbative</td><td>4</td><td>8</td></tr>
<tr><td>Theory nuisance parameters</td><td colspan="2">10</td></tr>
<tr><td>c, b quark mass</td><td colspan="2">4</td></tr>
<tr><td>Higher-order EW</td><td>6</td><td>7</td></tr>
<tr><td>Z width</td><td colspan="2">1</td></tr>
<tr><td>Z mass</td><td colspan="2">1</td></tr>
<tr><td>W width</td><td>–</td><td>1</td></tr>
<tr><td>W mass</td><td>–</td><td>1</td></tr>
<tr><td>$\sin^2\theta_W$</td><td colspan="2">1</td></tr>
<tr><td>Total</td><td>3750</td><td>4859</td></tr>
</table>

7

Requirements



8



Shorten the gap between data and results: NanoAOD

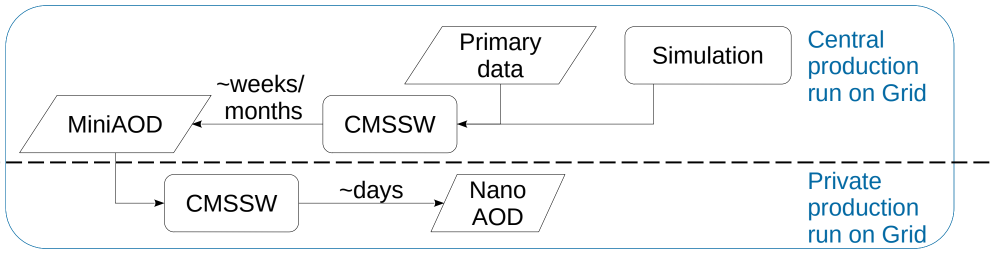
Central supported compact CMS event data format [0,1]

- Flat ROOT TTree
  - Independent of experiment specific software
- High level physics objects
  - (p_T, η, φ, ID, ... of muons, electrons, jets, ...)
- ~2kB per event
- Good for ~50% of analyses

Analysis data formats

Data tier	Size (kB)
RAW	1000
Gen	<50
SIM	1000
DIGI	3000
RECO(SIM)	3000
AOD(SIM)	400
MiniAOD(SIM)	50
NanoAOD(SIM)	2

9



Shorten the gap between data and results: NanoAOD

Central supported compact CMS event data format [0,1]

- Flat ROOT TTree
  - Independent of experiment specific software
- High level physics objects
  - (p_T, η, φ, ID, … of muons, electrons, jets, …)
- ~2kB per event

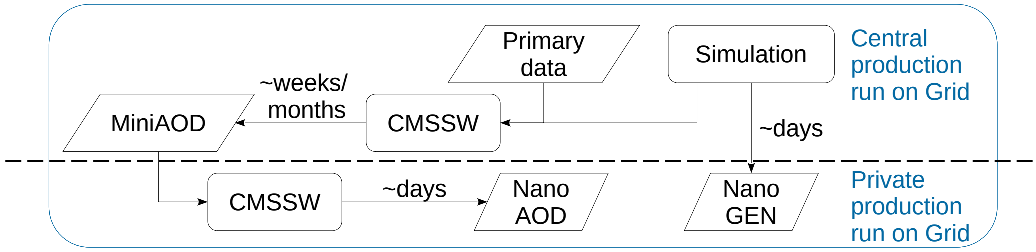
Easy customization with additional information

- Alternate PDFs, Info for muon track fit, …

Data tier	Size (kB)
RAW	1000
Gen	<50
SIM	1000
DIGI	3000
RECO(SIM)	3000
AOD(SIM)	400
MiniAOD(SIM)	50
NanoAOD(SIM)	2

Analysis data formats

10



NanoGEN and NanoLHE

NanoAOD with only then GEN-related branches

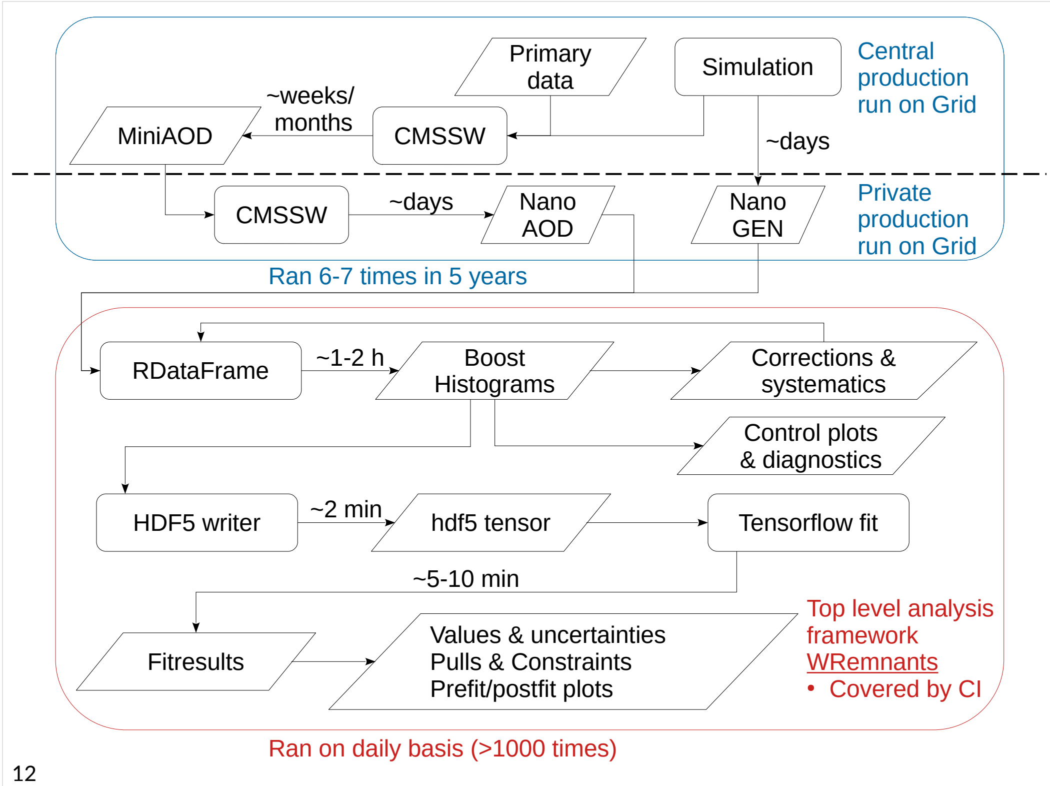
- Developed to validate MiNNLO event generator
  - Now centrally supported in CMS
- Producible directly from gridpack
- Lightweight, no detector simulation
- ~0.4kB per event

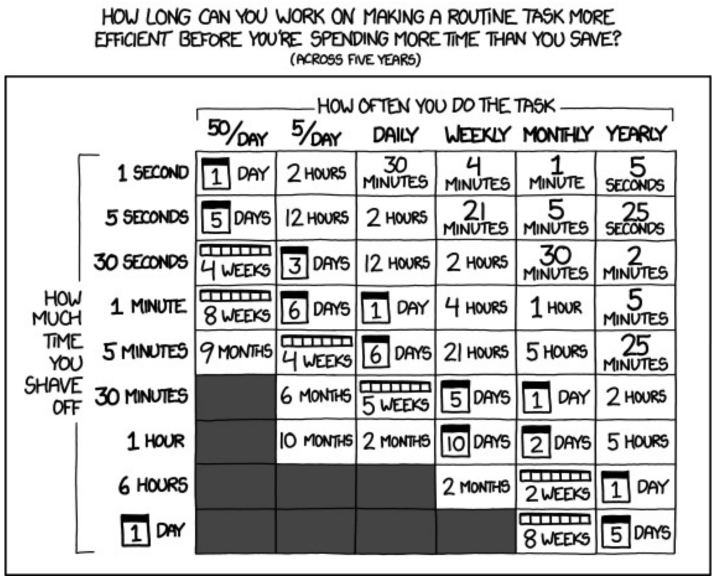
Large quantities produced

- O(100M) for MiNNLO validation
- O(10B) for EW uncertainties

Analysis data formats

Data tier	Size (kB)
RAW	1000
Gen	<50
SIM	1000
DIGI	3000
RECO(SIM)	3000
AOD(SIM)	400
MiniAOD(SIM)	50
NanoAOD(SIM)	2
NanoGEN	0.4


Primary data
Simulation
Central production run on Grid
~weeks/ months
MiniAOD
CMSSW
~days
CMSSW
~days
Nano AOD
Nano GEN
Private production run on Grid
Ran 6-7 times in 5 years
RDataFrame
~1-2 h
Boost Histograms
Corrections & systematics
Control plots & diagnostics
HDF5 writer
~2 min
hdf5 tensor
Tensorflow fit
~5-10 min
Top level analysis framework
WRemnants
• Covered by CI
Fitresults
Values & uncertainties
Pulls & Constraints
Prefit/postfit plots
Ran on daily basis (>1000 times)
12



Where we started

High performance computing machines

Custom analysis framework executed locally

- No resubmission of failed jobs/ merging of jobs etc.
- Direct feedback on progress

Run on single high performance machine

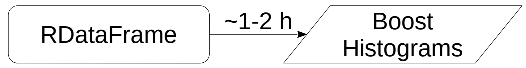
- Reading/writing on fast NVMe SSDs
  - Local or via network interface 100Gbit/s
- Reading from local CERN eos via xrootd
  - Network interface 100Gbit/s

	CERN	MIT/Pisa
CPU	2 x EPYC 7702	2 x EPYC 9654
cores	128	192
threads	256	384
memory	1TB	1.5/2TB

Possible upgrade for the future

- EPYC Turin machine with 384 cores/ 768 threads

ROOT RDataFrame

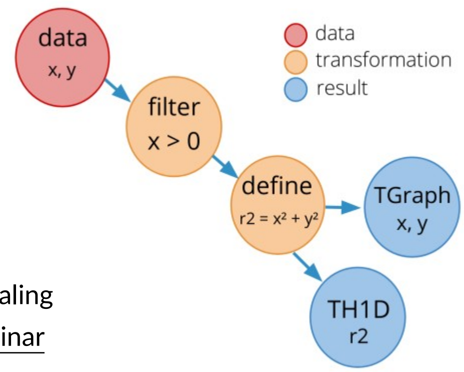


Select objects, filter events, fill histograms

- Pythonic, declarative, graph-style analysis

```
from ROOT import RDataFrame
df  = RDataFrame(dataset);
df2 = df.Filter("x > 0")
        .Define("r2", "x*x + y*y");
rHist = df2.Histo1D("r2");
g = df2.Graph("x","y")
```

Internal computation graph



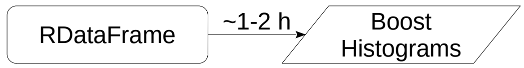
- Lazy execution: perform all operations in parallelized single event loop
- Executed on local machine
  - Plan to explore distRDF for multi-node scaling
- See RDF reference, documentation, EP seminar

Many optimizations conducted to ensure good thread scaling

- Now fully integrated in ROOT

15

ROOT RDataFrame



Critical parts in c++

- Functions: e.g. check if reco muon has a match to any gen muon

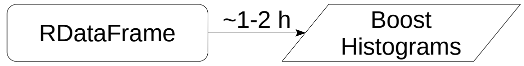
```
bool hasMatchDR2(const float& eta, const float& phi, const Vec_f& vec_eta, const Vec_f& vec_phi, const float dr2 = 0.09) {

  for (unsigned int jvec = 0; jvec < vec_eta.size(); ++jvec) {
    if (deltaR2(eta, phi, vec_eta[jvec], vec_phi[jvec]) < dr2) return true;
  }
  return false;

}
```

- Compiled at runtime using cling jitting
- And in python

```
df = df.Filter("wrem::hasMatchDR2(goodMuons_eta0,goodMuons_phi0,GenPart_eta[postfsrMuons],GenPart_phi[postfsrMuons],0.09)")
```

- Other examples much more complex – but follow same logic
- We also tried Numba, but found less efficient and not more convenient

ROOT RDataFrame



Critical parts in c++

- Helpers – classes that contain histograms with corrections and functions to apply them: e.g. reweight pileup spectrum in MC to the one in data

```
class pileup_helper {
public:

  pileup_helper(const TH1D &puweights) :
    puweights_(make_shared_TH1<const TH1D>(puweights)) {}

  // returns the pileup weight
  double operator() (float nTrueInt) const {
    return puweights_->GetBinContent(puweights_->FindFixBin(nTrueInt));
  }


private:
  std::shared_ptr<const TH1D> puweights_;
};
```

- And in python

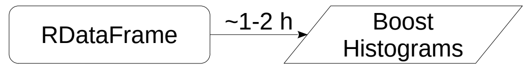
```
helper = ROOT.wrem.pileup_helper(puweights)
```

```
df = df.Define("weight_pu", pileup_helper, ["Pileup_nTrueInt"])
```

- Other examples much more complex – but follow same logic

17

ROOT RDataFrame



Critical parts in c++

- Often templated (e.g. for histogram bins)
- Also using Eigen and tensorflow c++ libraries

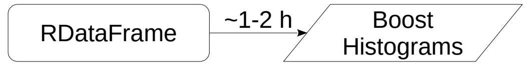
```
template <std::size_t NEtaBins>
class muon_prefiring_helper_stat {

public:

  static constexpr std::size_t NVar = NEtaBins + 1;
  using value_type = Eigen::TensorFixedSize<double, Eigen::Sizes<NVar, 2>>;

  muon_prefiring_helper_stat(const muon_prefiring_helper &other) :
    parameters_(other.parameters()), hotspot_parameters_(other.hotspot_parameters()) {}

  value_type operator() (const Vec_f& eta, const Vec_f& pt, const Vec_f& phi, const Vec_i& charge, const Vec_b& looseId, double nominal_weight = 1.0) const {
      [...]

      return res;
    }

  private:
    std::shared_ptr<const TH2D> parameters_;
    std::shared_ptr<const TH2D> hotspot_parameters_;

  };
```

- And in python

```
helper_stat = ROOT.wrem.muon_prefiring_helper_stat[netabins](helper)
```

```
df = df.Define("weight_newMuonPrefiringSF", muon_prefiring_helper, ["Muon_correctedEta", "Muon_correctedPt", "Muon_correctedPhi", "Muon_correctedCharge", "Muon_looseId"])
```

Histograms

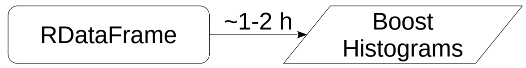


Strategy to perform computations on histograms later in analysis chain

- Allows for more flexibility
- E.g. data-driven nonprompt background prediction
- Nominal histogram is 5D

Axis	Bins
p_T^{μ}	30
η^{μ}	48
q^{μ}	2
I_{rel}^{μ}	2
m_T^{W}	3
All	17,280

Histograms



Strategy to perform computations on histograms later in analysis chain

- Allows for more flexibility
- E.g. data-driven nonprompt background prediction
- Nominal histogram is 5D
- Largest histograms with 8D and 20M bins
  - For efficiency scale factor 2D smoothed in p_T and u_T
- ~same histograms for 16 processes

Axis	Bins
p_T^μ	30
η^μ	48
q^μ	2
I_{rel}^μ	2
m_T^W	3
var. η^μ	48
var. q^μ	2
eig. vec.	12
All	19,906,560
All (w/ flow)	358,400,000

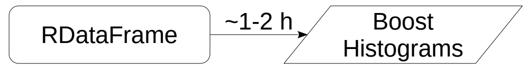
Significant memory consumption

→ For largest histogram: 2.5GB

→ For all: 13GB

Gets much worse if flow bins can't be disabled (as in root histograms)

Boost histograms



Previously: one root histogram copy for each thread

- But large memory consumption was a showstopper
- Long merging time when adding up at the end

Solution: use std:atomic<double> with c++ boost histograms

- All threads write in same histogram
- But can't use python binding directly … (cppyy vs. pybind 11)

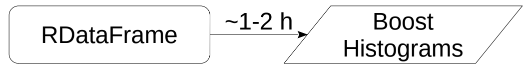
Custom copy conversion into python boost histograms

- Arbitrary number of axes
- Configurable underflow/overflow bins
- Convenient (numpy like) indexing/ manipulation

Histograms stored with pickle

- Using proxies dictionary in .hdf5 to allow lazy loading (code)
- Including meta data (e.g. number of processed events, cross section/luminosity, command, …)

Tensor axes



All systematic uncertainties represented by event weight variations

Traditionally one histogram per variation

- e.g. NNPDF provides 101 alternate PDF weights → 101 histograms

Better: a single histogram with an additional axis

Even better: fill full array/tensor at once, only do bin lookup once

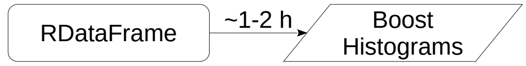
  - Using Eigen tensors
  - Arbitrary number of dimensions

Atomic boost histograms and tensor axes implemented in narf submodule

- More details given at ROOT Users Workshop 2022: link
- Not currently integrated in root; similar functionality in RHistogram?
  - Interest also from outside W mass analysis team

22

Histogram benchmark



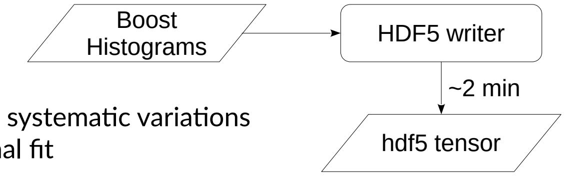
Using 400M events of CMS NanoAOD (W→μν) and filling 10 copies of pdf variation histograms

256 threads (2 EPYC 7702)

Hist Type	Hist Config	Evt. Loop	Total	CPUEff	RSS
ROOT THnD	10 x 103 x 5D	59m39s	74m05s	0.74	400GB
ROOT THnD	10 x 6D	7m54s	25m09s	0.27	405GB
Boost ("sta")	10 x 6D	7m07s	7m17s	0.90	9GB
Boost ("sta")	10 x (5D + 1-tensor)	1m54s	2m04s	0.81	9GB
Boost ("sta")	1 x (5D + 2-tensor)	1m32s	1m42s	0.77	9GB

- Root histograms slowed down by merging step
- Memory much lower with atomic accumulation
- Factor ~4 time reduction with tensor axes due to reduced lookup
- Some additional subtleties related to cash locality

HDF5 writer



Histograms with data, prediction, and systematic variations need to be casted into a tensor for final fit

- Purely python based (boost histograms, numpy, ...)
- Flexibility & efficient implementation is essential
- Perform selections/ accumulations/ other computations on histograms
  - Signal selection
  - Data-driven nonprompt background estimation
    - Smoothing "on-the-fly" using least squares (code)
  - Modify systematic variations e.g. decorrelating/ combining

Sparse tensor implementation for unfolding

Binned profile maximum likelihood fit

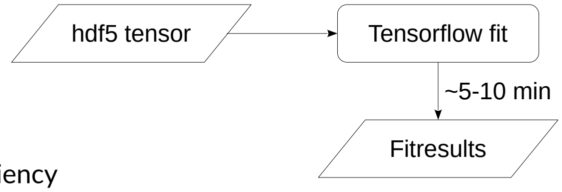
Log likelihood from Poisson distributed bin-by-bin event numbers

$$L = \sum_{ibin} \left(-n_{ibin}^{obs} \ln n_{ibin}^{exp} + n_{ibin}^{exp} \right) + \frac{1}{2} \sum_{ksyst} \left(\theta_{ksyst} - \theta_{ksyst}^{0} \right)^2$$

$$n_{ibin}^{exp} = \sum_{jproc} \mu_{jproc} n_{ibin,jproc}^{exp} \prod_{ksyst} \kappa_{ibin,jproc,ksyst}^{\theta_{ksyst}}$$

- Gaussian constraint nuisance parameters θ for systematic uncertainties
- Signal strength modifier μ
- Systematic variations in 3D tensor κ

Tensorflow fit



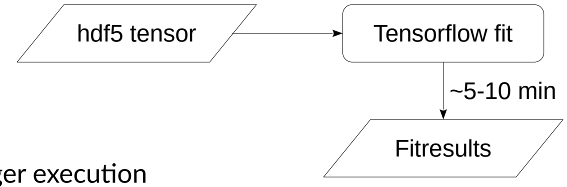
RooFit via minuit insufficient

- Limited numerical stability and efficiency
  - E.g. can not be parallelized

Tensorflow library with automatic gradient computation via back propagation for minimization:

- Quasi Newton trust region based minimizer to reliably find global minimum
  - Native tensorflow implementation; algorithm based on arXiv:1506.07222
- Fast, numerically accurate, stable
- Parallelized vector processing units and/or multiple threads
- Sparse tensor implementation to minimize memory consumption (if response matrix is close-to-diagonal, e.g. leptonic observables)
- Implemented in combineTF, see also PyHEP 2020: link

Tensorflow 2 fit



Re-written in Tensorflow 2:

- More developer-friendly due to eager execution
- Almost feature complete combineTF2 implementation
- More efficient computatoin of hessian and hessian vector products
- Trust-krylov minimizer from SciPy, computing the gradient and hessian-vector product in tensorflow 2
  - I.e. not using quasi-newton methods as in the combineTF1 case

Benchmark using MIT machine

- CPU: EPYC 9654
- GPU: Nvidia A30

	fit	fit + covariance
CombineTF1 CPU	1m49s	3m48s
CombineTF2 CPU	34s	47s
CombineTF2 GPU	36s	39s

GPU “only” used to calculate the gradient/hessian/hessian-vector-product

Continuous integration

Common framework among all analyzers

- Sharing as much code as possible among different efforts
- Reuse existing code, find/avoid bugs, save time
- Quickly developed with O(10) contributors, now at >500 pull requests (PRs)

However

- Updates often unintentionally affected other parts
  - Framework was constantly broken
- Sometimes not clear where certain changes came from

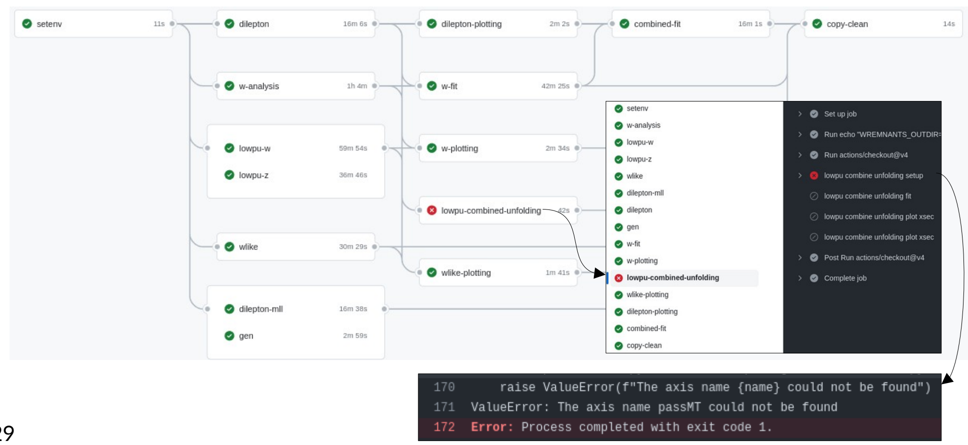
Solution → GitHub actions: platform for automate developer workflows

- Use continuous integration and deployment (CI/CD) pipeline
- Same tool as used for code development instead of third party integration
- Slim and easily to set up and manage (compared to e.g. Jenkins)

Github CI workflow

Different analysis chains implemented

- Independent jobs run in parallel, each job contains a set of steps
- Different arguments for plotting/ fitting for good code coverage
- Investigate failed jobs directly in Github actions



Github CI workflow

```yaml
on:
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '30 5 * * 1-5'
    - cron: '0 1 * * 2,4,6'
  workflow_dispatch:
```

1)
2)
3)
4)

Running full analysis chain (code)

1) For each PR on reduced set of files (~1%)

2) Scheduled each morning on reduced set of files (~1%) as reference for PR

3) Scheduled 3 times a week on (1:1) data:MC files to backtrack changes

   - All output files (e.g. histograms) stored on EOS for later use
   - Separate workflow to delete old files

4) Workflow dispatch on (1:1) data:MC files to manually run on chosen branch

   - To test a new feature (e.g. apply new nominal calibration/correction)

In the process of adding code checks

- Run in CI and as pre-commit hooks
- Syntax checks for python, c++, yaml, json files
- Linters: Black, Flake8, isort

Everything blinded

30

Github CI infrastructure

Maintained via service account with CMS access and eos area

- Self hosted runners to easy access resources and execute code on the CERN high-performance analysis machine used for this analysis

```
setenv:
  runs-on: [self-hosted, linux, x64]
```

- Repository with sub modules checked out including large file storage (lfs) support

```
steps:
  - uses: actions/checkout@v4
    with:
      submodules: 'recursive'
      lfs: 'true'
```

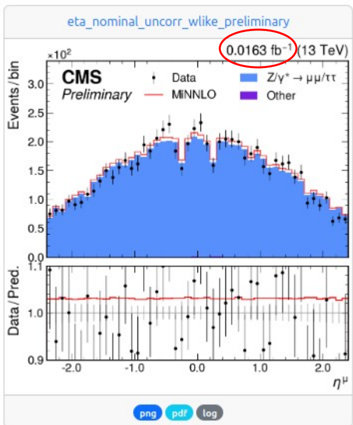
Network authentication via Kerberos, key stored in local keytab file

```
- name: setup kerberos
  run: |
    kinit -kt ~/private/.keytab cmsmwbot@CERN.CH
    klist -k -t -e ~/private/.keytab
    klist
    echo "xrdfs root://eosuser.cern.ch// ls $EOS_DIR"
    xrdfs root://eosuser.cern.ch// ls $EOS_DIR

- name: setup kerberos within singularity image
  run: |
    scripts/ci/run_with_singularity.sh kinit -kt ~/private/.keytab cmsmwbot@CERN.CH
    scripts/ci/run_with_singularity.sh klist -k -t -e ~/private/.keytab
    scripts/ci/run_with_singularity.sh klist
    echo "xrdfs root://eoscms.cern.ch// ls $EOS_DATA_DIR"
    scripts/ci/run_with_singularity.sh xrdfs root://eoscms.cern.ch// ls $EOS_DATA_DIR
```

Webpage support

Results initially created in local temporary folder

- Copied via xrdcp to CMS protected webpage
- CMS centrally maintained plot browser
- Automatic lumi scaling for using subset of data files



/ WMassAnalysis / PRValidation

Directories

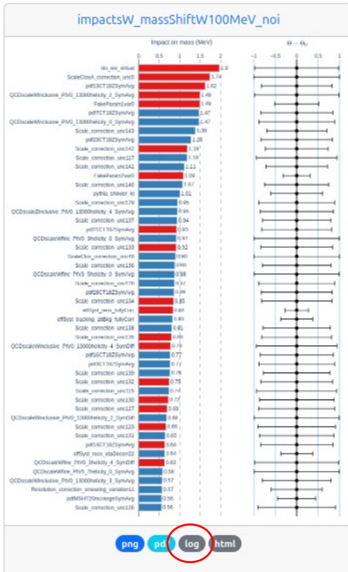
- Archive
- Dispatch
- PR2XX
- PR3XX
- PR4XX
- PR500
- PR501

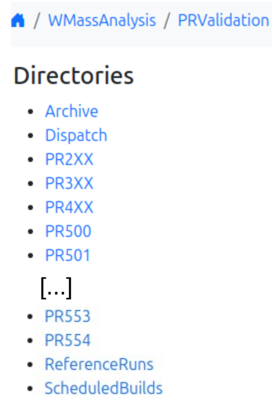
[…]

- PR553
- PR554
- ReferenceRuns
- ScheduledBuilds

32

Webpage support





Plots for nuisance parameter pulls, constraints, and impacts produce via plotly

- Available in interactive .html
- See all O(1000) nuisances with more digits

Each plot is produced with a .log file

33

.log files

```
--------------------------------------------------------------------------------
Script called at 2024-10-01 15:56:53.608388
The command was: scripts/plotting/postfitPlots.py
'/scratch/dwalter/CombineStudies/test/ZMassDilepton_ptll_yll/fitresults_123456789.root' --
legCols 1 --eoscp -f '241001_test' --yscale '1.25'
--------------------------------------------------------------------------------
```

Check exact event yields

```
Yield information for Stacked processes
--------------------------------------------------------------------------------
                        Process       Yield  Uncertainty
0               $\gamma$-induced     1796.97        94.01
1  Z/$\gamma^{\star}\to\tau\tau$     1582.57        77.61
2                          Other     1630.27        14.59
3    Z/$\gamma^{\star}\to\mu\mu$  1931915.83       336.74

Yield information for Unstacked processes
--------------------------------------------------------------------------------
     Process       Yield  Uncertainty
0       Data  1936925.64      2547.78
1  Inclusive  1936925.64       313.63

===> Sum unstacked to data is 100.00%
--------------------------------------------------------------------------------
```

Command chain used to produce the plot

→ Look up how to run specific scripts

```
Meta info from input file AnalysisOutput

--------------------------------------------------------------------------------
{
     "time": "2024-10-01 15:39:52.107792",
     "command": "scripts/combine/setupCombine.py -i
'/scratch/dwalter/results_histmaker/test/mz_dilepton.hdf5' --fitvar 'ptll-yll' --lumiScale
100 --realData -o '/scratch/dwalter/CombineStudies/test'",
     "args": {
          "outfolder": "/scratch/dwalter/CombineStudies/test",
          "inputFile": [
               "/scratch/dwalter/results_histmaker/test/mz_dilepton.hdf5"
          ],
          "postfix": null,
          "verbose": 3,
```

[...]

Git commit hash

```
     "git_hash": "\"4713c27278391e1df49f86834c6d122cff8beba5\"
",
```

Local untracked changes

→ Each plot is reproducible

```
     "git_diff": "diff --git a/scripts/combine/saturatedGOF.py
b/scripts/combine/saturatedGOF.py
index 1b9fb2e9..1df140a8 100644
--- a/scripts/combine/saturatedGOF.py
+++ b/scripts/combine/saturatedGOF.py
@@ -15,7 +15,7 @@ tree.GetEntry(0)

 fitresult_h5py = combinetf_input.get_fitresult(args.infile.replace(\".root\",\".hdf5\"))
 meta = ioutils.pickle_load_h5py(fitresult_h5py[\"meta\"])
-nbins = sum([np.product([len(a) for a in info[\"axes\"]]) for info in
meta[\"channel_info\"].values()])
+nbins = sum([np.prod([len(a) for a in info[\"axes\"]]) for info in
meta[\"channel_info\"].values()])
 ndf = nbins - tree.ndofpartial
```

34

Many interesting features not discussed today

Other analysis ingredients

- Efficiencies
  - Using tag and probe fits, smoothing of scale factors in 1D/2D
- Helicity cross section corrections & uncertainties
  - Based on Eigen
- Muon calibration
  - Object to event weight variations via CDF transform
- Recoil calibration
  - Functional fit based on JAX, evaluation with tensorflow lite c++
- ...

Summary

Increasing amount of data opens new opportunities

- Software developments must be ahead to fully exploit potential

Fast analysis turnaround was essential for this complex measurement

- RDF provides a convenient and efficient library
  - Initially showstoppers observed in scaling
  - Extensive work on critical parts to improve RDF and histogram implementation
- Full analysis runs in ~hours

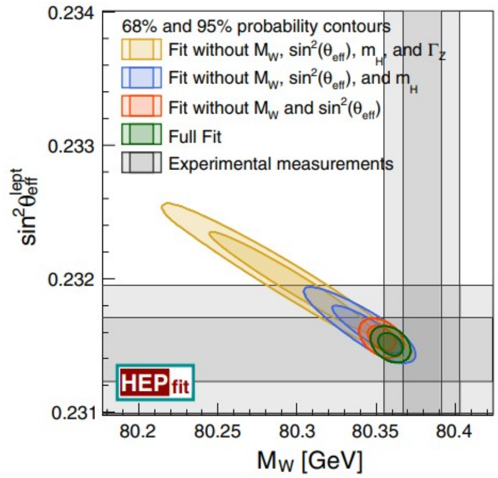
Challenging collaborative work with increasing number of contributors

- Github CI/CD pipeline has turned out to be extremely useful
- Time savings in PR reviews, spot/avoid bugs, backtrack changes
- Always ensure working implementation for different analyses/ configurations

Many areas identified for further improvements

Backup

Precision measurements of standard model parameters provide opportunity to over constrain the theory and pose stringent tests



Indirect prediction (~6MeV) more precise than direct measurement (~10MeV) and in tension (CDF)

→ Call for more precise measurements

Lumitools

Automatic computation of integrated luminosity of processed data

- CMS data is organized by fill, run, luminosity block (~24s)
  - Use .csv file containing integrated luminosity information
  - Provided by the CMS BRIL group
- Processed with RDataFrame, read non-ROOT data
- Guarantees consistent luminosity calculation
- Convenient for running on subset of data

Implemented in lumitools

- Could be used standalone

Histogram benchmark

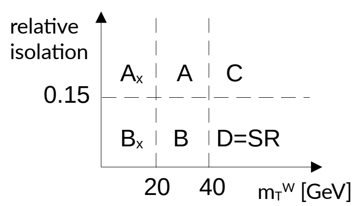
Hist Type	Hist Config	Evt. Loop	Total	CPUEff	RSS
ROOT THnD	10 x 103 x 5D	59m39s	74m05s	0.74	400GB
ROOT THnD	10 × 6D back	7m54s	25m09s	0.27	405GB
ROOT THnD	10 × 6D front	13m52s	30m27s	0.42	406GB
Boost ("sta")	10 × 6D back	7m07s	7m17s	0.90	9GB
Boost ("sta")	10 × 6D front	3m22s	3m33s	0.86	9GB
Boost ("sta")	10 x (5D + 1-tensor)	1m54s	2m04s	0.81	9GB
Boost ("sta")	1 x (5D + 2-tensor)	1m32s	1m42s	0.77	9GB

- In the tensor/array weight-case the weights for the different systematic idxs are contiguous in memory by construction
- In the N+1-d histogram case it depends on the array ordering
- TH1/2/3 and boost-histograms have fortran array ordering → systematic idx axis is best at the front
- THn has C array ordering → systematic idx axis is best at the back
- The difference is about a factor of 2 for both root and boost hists (but still > 50% additional gain from tensor filling)
- Largely accounted simply by skipping the extra FDIVs needed for redundant value-to-index conversion for the 5 axes

QCD multijet background estimation

Estimated from data using extended ABCD method

- Prediction from yields in sideband regions in bins of high relative isolation and low m_T^W
- Prompt background in sideband region subtracted from simulation
  - Repeated for each systematic variation ~O(1000) times
- Evaluated in fine bins in p_T^μ, η^μ, q^μ

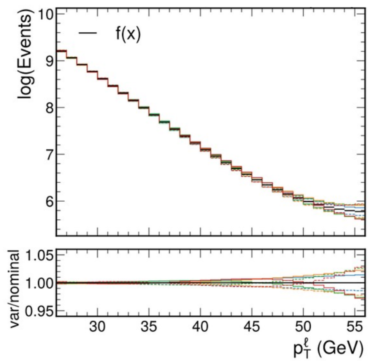


$$D = C \cdot \underbrace{\frac{A_x B^2}{B_x A^2}}_{\text{fakerate}}$$

QCD multijet background estimation

Smoothing each sideband region in p_T^μ with exponential "on-the-fly"

- Maintain good statistical properties
- Smoothing in 5 regions, 96 bins for η^μ, q^μ
- Repeated for O(1000) systematic variations
- Robust and efficient calculation required
  - Use polynomial in log space
  - Analytic solutions using least squares



$$f_i(p_\mathrm{T}) = \mathrm{e}^{P_i(p_\mathrm{T})}$$

$$f_\mathrm{D}(p_\mathrm{T}) = \mathrm{e}^{\sum_i w_i P_i(p_\mathrm{T})}$$

  - Systematic uncertainties from eigenvector decomposition
  - Everything done in ~seconds

More complex procedures tested

  - E.g. using integrated Bernstein polynomials with nnls to enforce monotonicity