

# PAT – PostDST Analysis Tool

A tool which brings you **closer** to your dedicated physical analysis making some intermediate analysis steps very easy but this is not the tool you draw your final histograms with

Each piece of software has some features and limitations!  
Very generic program leaves people **helpless** and the time necessary to "understand" and "configure" it is usually unacceptable

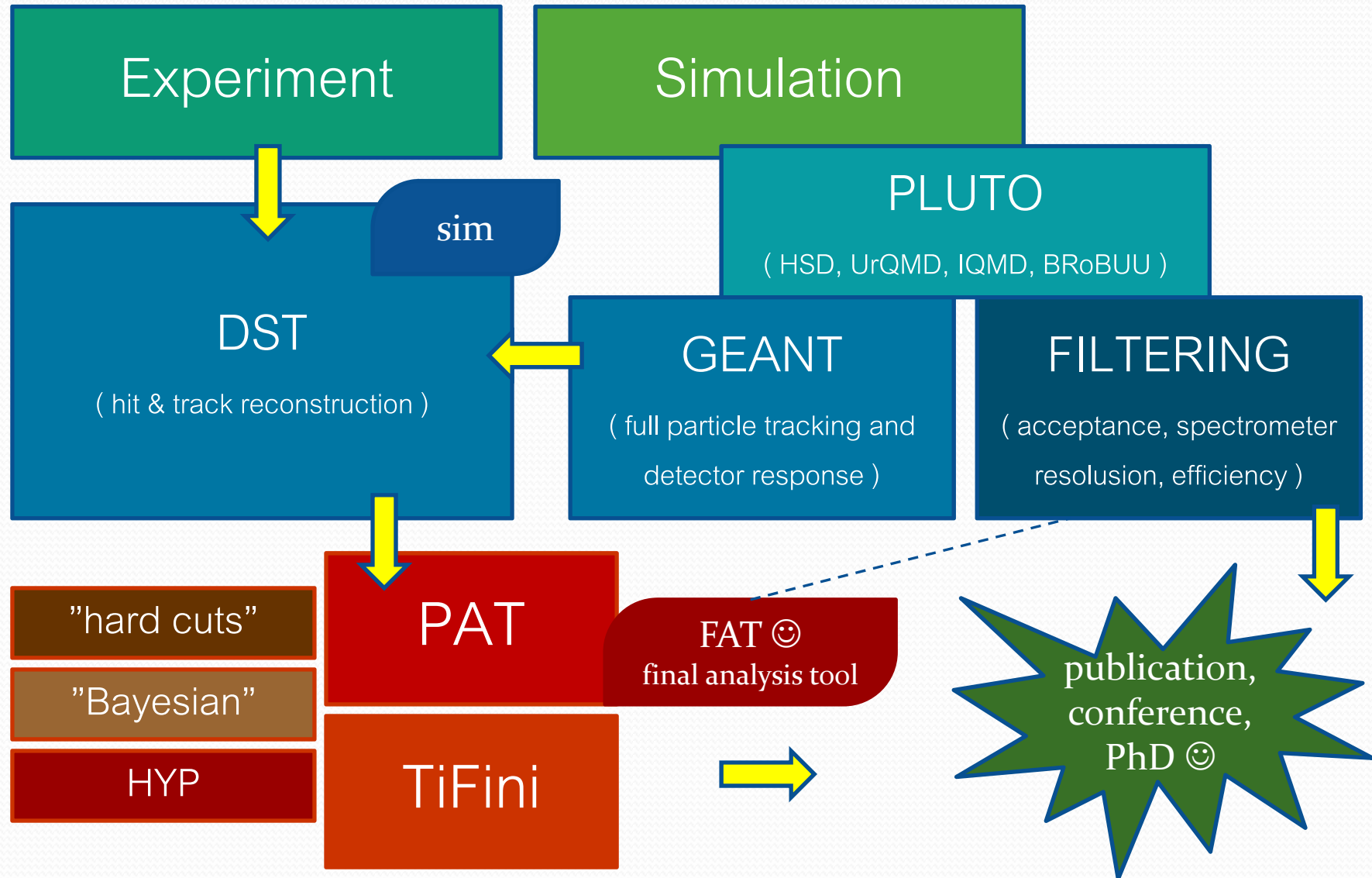
**Rapid analysis** development  
and immediate effects

**Simple** no deep programming knowledge  
(unless you really want to change the code)

In most cases you will need to change only a few lines of code (define particle combinations and provide your (i.e. graphical) cuts in ROOT files.

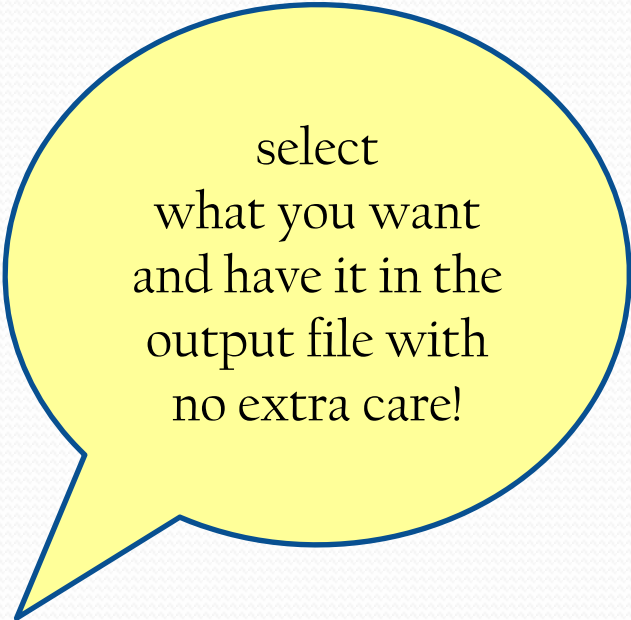
If your analysis is very specific and goes beyond the tool can provide, do not hesitate to ask how to alter the code

# How the data circulate in HADES



# PAT - general idea

- **Intermediate analysis tool**
  - From **HPidTrackCand(Sim)** – DST level
  - Until any combination of single tracks
    - Full combinatorics done automatically
  - **Information propagated**
    - Common event data
    - Full set of particle data
    - Automatic data flow from the beginning till the end
    - Minimum (or none) of code to be written by the user
- **The last part (very specific physical analysis) has to be written and understood by the user!**



select  
what you want  
and have it in the  
output file with  
no extra care!

## Example 1

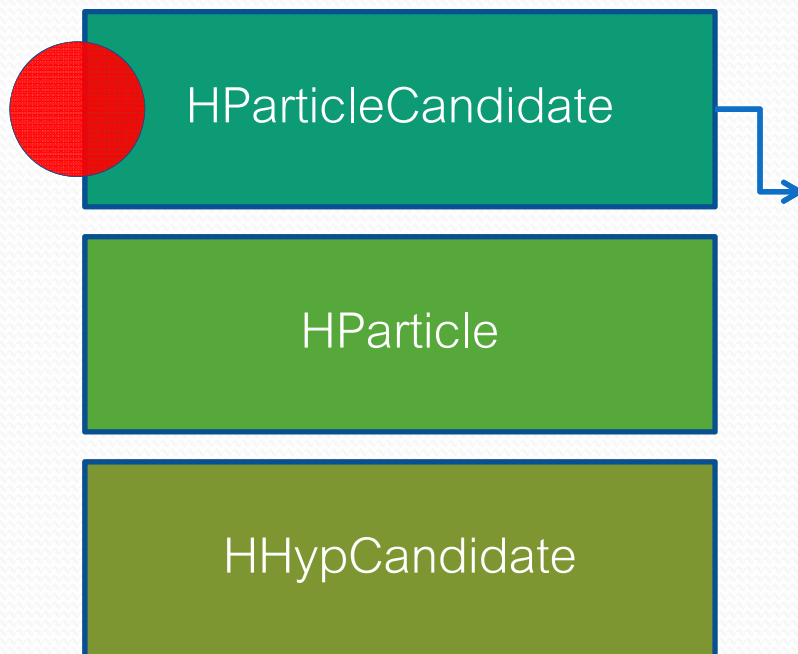
# HNtuple – simple data tree

- Location: **libHydra.so** (HYDRA), but for training now:
- Download source code ([HNtuple.tar.gz](http://hades-wiki.gsi.de/cgi-bin/view/Homepages/HNtuple)):  
<http://hades-wiki.gsi.de/cgi-bin/view/Homepages/HNtuple>
- In PAT you will define your particle set (a combination from 1 to 4 particles) with a label
- The label will become automatically an NTuple name
- You will select the variables to be stored always with the method `set("any_name", value)`
- Automatic data propagation till the output file
- No need to book explicitly any ntuple, no need to take care of data once they have been selected

# PAT – compile the source code

- Location: (not in HYDRA so far)
- Download source code ([PAT\\_5thSummerSchool.tar.gz](http://hades-wiki.gsi.de/cgi-bin/view/Homepages/PAT)):  
<http://hades-wiki.gsi.de/cgi-bin/view/Homepages/PAT>
- For this training only the core and the user parts are together (because the code is not in HYDRA yet)
- The core part of the source code (it will be a library) is supposed not to be changed unless necessary
- The user part is the place to define what to analyze and which selections, cuts etc. to apply
- User is responsible for cut definition (i.e. you have to draw your own graphical "banana" cuts)

# Basic data units



Container with a set of useful particle parameters, copied from **HPidTrackCand**, (or **HPidTrackCandSim**) selected by the user. Auto-recognition if data are simulation and copying additional set of data.

Storing a variable is as easy as:  
**set("label" , variable);**  
called in the constructor.

*For example:*

**set("rich\_amp", pHit->getRingAmplitude() );**  
and since then "rich\_amp" is propagated to all higher data levels.

## *Where and how can I add or remove data of my interest?*

- Open file: `hparticlecandidate.cc`

```
set("id", 0.);
set("sector", HitData->getSector());
set("system", HitData->getSystem());
set("p", TrackData->getMomenta(4));
// ... and so on ...
HPidTrackCandSim *PidCandSim = dynamic_cast<HPidTrackCandSim*>( ptr );
if ( PidCandSim != 0 ) {
    HPidGeantTrackSet* ptrSim = PidCandSim->getGeantTrackSet();
    set("sim_iscommon", ptrSim->getMostCommonCorrelation());
    set("sim_id", ptrSim->getGeantPID());
}
```

Here we recognize  
the simulation!

- Any data is just a matter of a line of code with "set" method.
- This is core code modification but very easy!

## FAQ

*Where is HParticleCandidate object added?*

*Can I get data also from the other categories?*

*(i.e. HWallHit or HMdcClusInf)...*

- Open file: `hparticlepool.cc` and `hparticledatapool.cc`

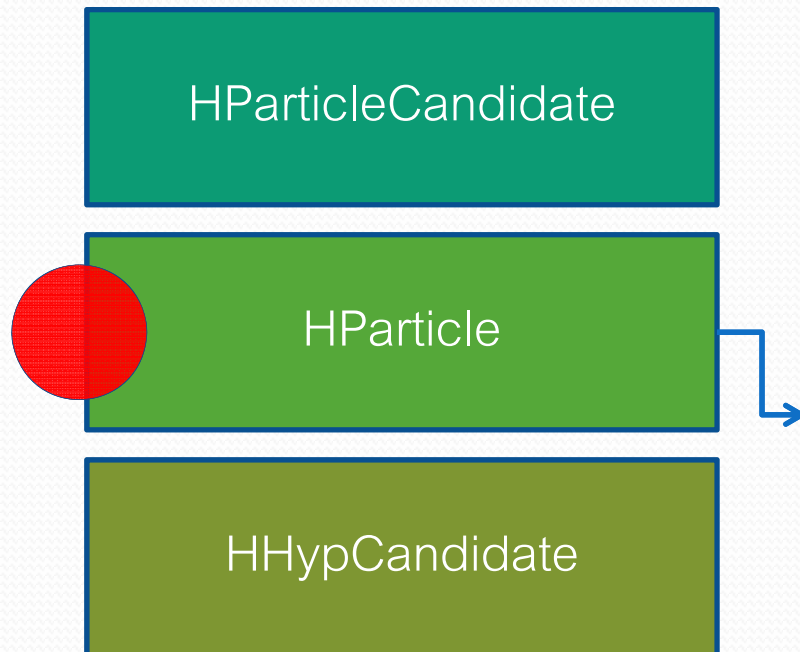
```
void HParticlePool::loop(HIterator* dataIt) {  
    // ...  
    if (PidCand->isFlagBit(HPidTrackCand::kIsUsed) == 1)  
        addPartCand(myEld, PidCand);
```

```
void HParticleDataPool::addPartCand  
(EParticle eId, HPidTrackCand *ptrC) {  
    HParticleCandidate *ptr =  
        new HParticleCandidate(ptrC);
```

- Similarly it is possible to add HWallHit information (for example another version of HParticleCandidate constructor). To any HParticleCandidate object one can add more data (i.e. from HMdcClusInfo) with the "set" method.
- However, this requires core code modification, therefore please call me for help!



# Basic data units



Wrapper to **HParticleCandidate** class pointers. Because the combinatorics is done on pointers (different combinations with the same pointers set) this class allows to **overwrite** existing variables and **add new variables**.

*For example:*

Each **HParticleCandidate** object has a prefix defining the particle species, like **hpos** – positive hadron, **lneg** – negative lepton etc. Hadron can then be a proton or or a  $\pi^+$ , therefore in **HParticle** the new particle id shadows the "old" one.

## Example 3

# HParticle – a wrapper

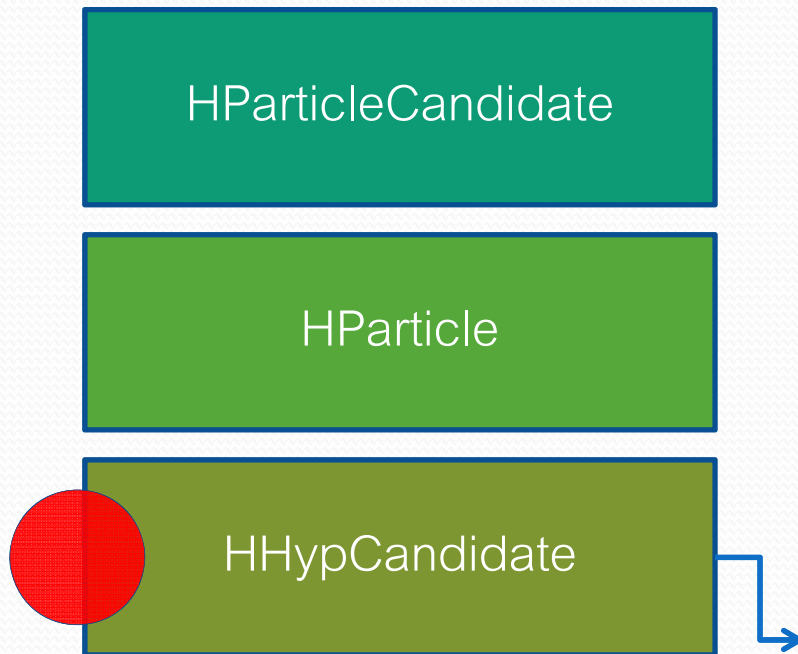
- Open file: [hparticle.cc](http://hparticle.cc)

```
HParticle::HParticle(HParticleCandidate *pC) : pCand(pC) {  
    set("id", pCand->getEId() );  
    set("track_length", -1. );  
    set("tof_mom", -1. );  
    set("tof_new", -1. );  
    set("beta_new", -1. );  
}
```

Here we overwrite the **id** value and add new variables not present in HParticleCandidate

- HParticle object contains (a pointer to) HParticleCandidate but also it has an independent list of data (name – value)
- User can add new data or overwrite "old" data (from HParticleCandidate) and the new value will be streamed to the output. **HParticleCandidate object is not changed because it contributes in many other combinations!**

# Basic data units



All basic units  
are kept in the  
higher data level:  
**pools.**

A combination of particles of a given pattern. Combination has a name (label) and a set (array) of pointers to **HParticles**.

Each "hypothesis" has a pointer to a pattern it represents. The container stores one set of combined particles and collection of such objects will keep all combinations.

## Example 4

# HHypCandidate – set of particles

- Open file: [hhypcandidate.h](#)

```
class HHypCandidate : public TObject { // ...
protected:
    ParticleCandSeq nPart;
    ParticleCandSeqIter partIter;
    HPattern *pPattern;
```

An array of particles (pointers)  
of a given combination based  
on a pattern

- There are many places in the code where I use **typedefs** of some types, i.e.: ParticleCandSeq. **All these (typedef) definitions you can find in the file hcommondef.h**  
`typedef std::vector< HParticle* > ParticleCandSeq;`
- If you want to read (understand, change...) the core code I recommend to print hcommondef.h file!

## *What is the "HPattern" to which HHypCandidate refers to?*

- Open file: `hpattern.h`

Really open it (there are too many lines to copy/paste here)

- HPattern object is very important (in the core code)!
- It holds the pattern of user defined particle combinations, i.e. pp, e+e-, pe+e-, pppi+pi-, K+K- etc.
- It holds the output (if defined), books the ntuple, passes the data to the ntuple, calls filling the ntuple
- User can also define the prefix or suffix to any data label, this way i.e. momentum (labeled "p") is automatically supplemented with the particle type, i.e. "ep\_p" for positron

# Data pools

Collections of  
particle  
candidates, sets  
of particles...

HParticleDataPool

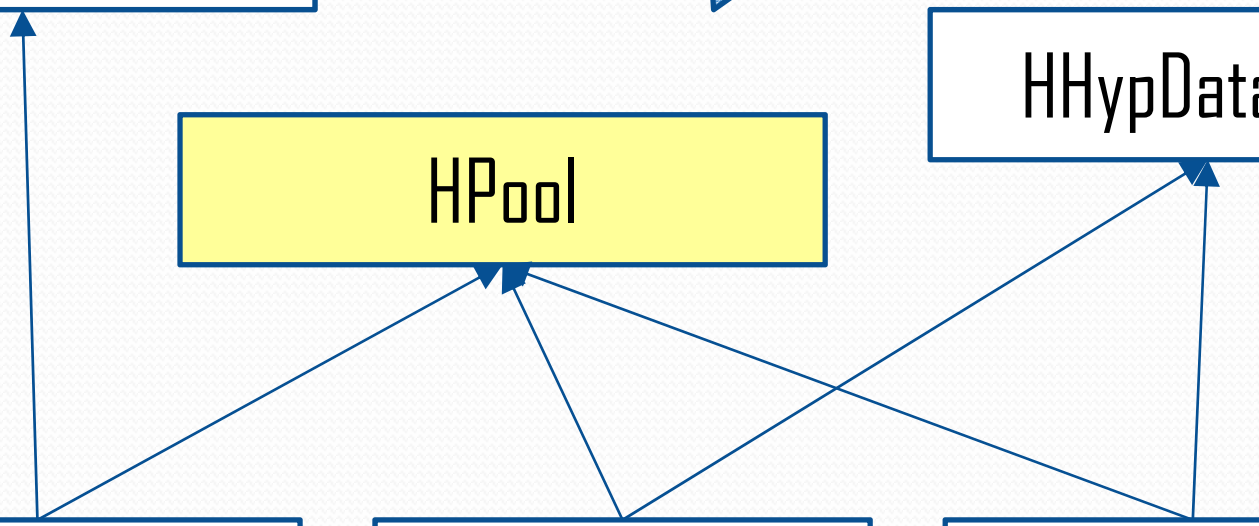
HHypDataPool

HPool

HParticlePool

HHypPool

HPidPool



## Example 5

# HPool – define your pattern

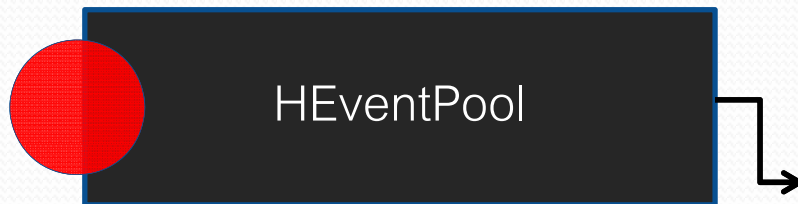
- Open file: `hpool.h`

```
class HPool { public:  
    HPool(HOutputFile *ptr = 0);  
    virtual ~HPool() = 0; // ...  
    bool add(const char* name, EParticle p1);  
    bool add(const char* name, EParticle p1, EParticle p2); // ... and so on  
protected:  
    HEventPool eventData;  
    std::list<HPattern*> objectList;
```

Set of common (event) data,  
i.e. event number,  
multiplicity, event vertex...

- Any "pool" is a data structure streamed to the output file (if defined). User defines here **patterns**: all particle combinations (method "add") – name (label) and particle species (number of particles from 1 to 4).

# Data pools



Container for keeping the common event information (event number, particle multiplicity, event vertex information *etc.*). Object of this type is present in every data pool and propagated to the next one.

At each higher data level (data pool) you can **add more information** to the HEventPool. It will be then automatically streamed to the output file.



# HParticleDataPool

```
MultiParticle partCand;
```

```
typedef std::multimap< EParticle, HParticleCandidate* > MultiParticle;
```

```
ParticleNum partNum;
```

```
typedef std::map< EParticle, int > ParticleNum;
```

# HHypDataPool

```
MultiHyp hypCand;
```

```
typedef std::multimap< std::string, HHypCandidate* > MultiHyp;
```

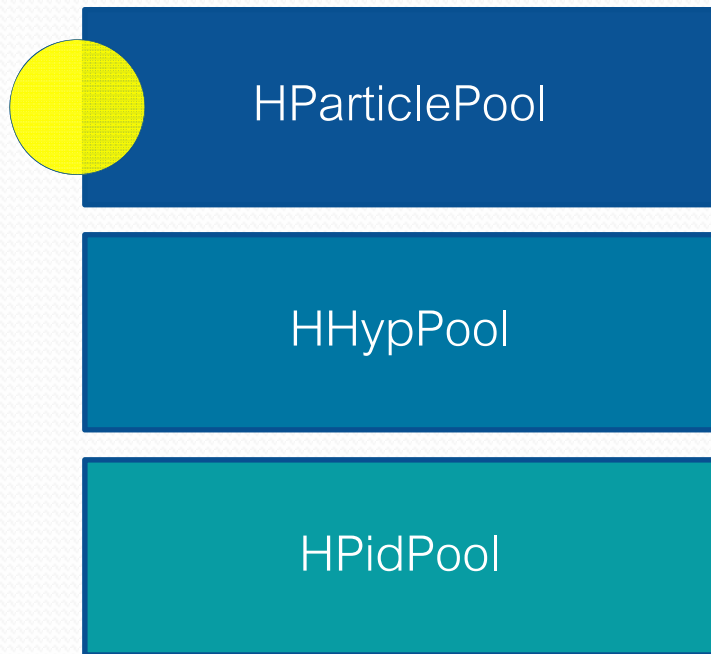
```
HypNum hypNum;
```

```
typedef std::map< std::string, int > HypNum;
```



Part of the code

# Data pools



Container for various particle candidates like +/- hadrons, +/- leptons first, then particles: p,  $\pi^+$ ,  $\pi^-$ ,  $e^+$ ,  $e^-$  and so on.

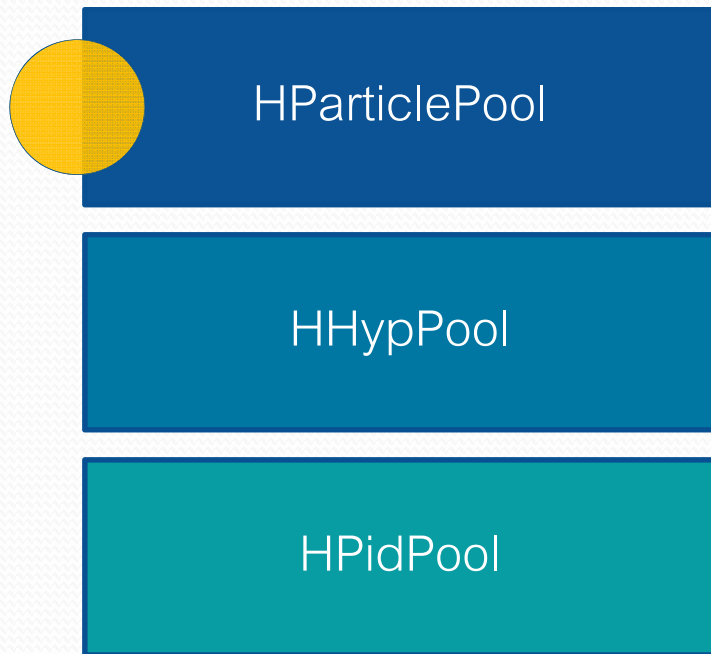
Defining your combination pattern:

```
HParticlePool myParticles( &outputFile );  
myParticles.add(„all“, eHadronPos, eHadronNeg,  
                eLeptonPos, eLeptonNeg);
```

*For example:*

You can also define a smaller subset, i.e. only lepton candidates. Particle species are decided based on tracking (Runge Kutta) information. Positive/negative polarity means +/- particle, correlation with a RICH ring means lepton.

# Data pools



**Important:**  
each data pool  
can be stored in  
the output file.

During copying particle (track) information from **HPidTrackCand** a track cleaner decision is taken into account in order to reduce garbage tracks (killing the performance and the signal). You can change it, i.e. to take all particle candidates.

Later, a cut (window between RICH / MDC) for leptons additionally checks correlation. If a ring and a track are outside the cut, the track returns to hadrons.

## Example 6

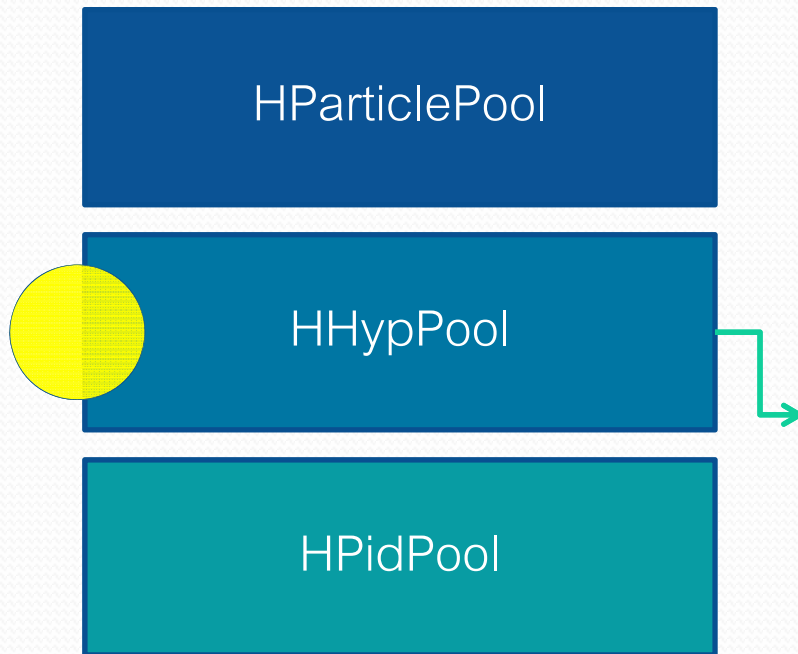
# HParticlePool – first step in data flow

- Open again file: [hparticlepool.cc](http://hparticlepool.cc)

```
data1->Reset();
while (((PidCand = (HPidTrackCand *) data1->Next()) != 0)) {
    isPositive = ( PidCand->getTrackData()->getPolarity(4) > 0 ) ? kTRUE : kFALSE;
    isRing = PidCand->getHitData()->getRingCorrelation(4);
    EParticle myEld;
    if (isPositive && isRing) myEld = eLeptonPos;
    else if (isPositive && !isRing) myEld = eHadronPos;
    else if (!isPositive && isRing) myEld = eLeptonNeg;
    else if (!isPositive && !isRing) myEld = eHadronNeg;
```

- Preliminary particle selection (positive/negative, hadron/lepton)
- Limit on the maximum multiplicity of a given particle species
- Only good tracks (based on track cleaning procedure) taken

# Data pools



Container for **HHypCandidates**, each of them has a name (label) and a set (array) of pointers to **HHypCandidate** object.

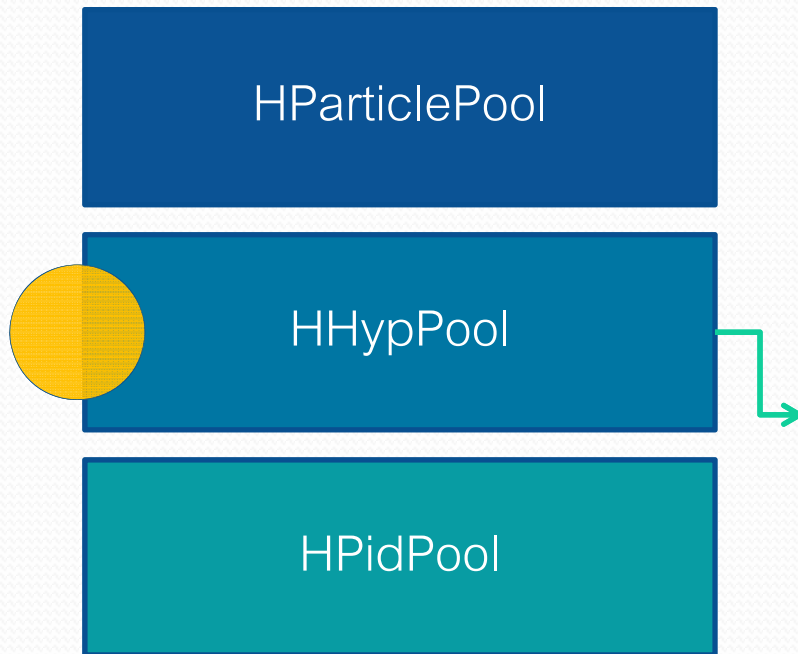
Defining your combination pattern:

```
HHypPool myHyps( &outputFile );  
myHyps.add("LpLm", eLeptonPos,eLeptonNeg);  
myHyps.add(„HpHp", eHadronPos,eHadronPos);
```

*For example:*

You give a label and list of particle species (from 1 up to 4). Here (**HHypPool**) we do not know particle id yet, only if it is positive or negative, lepton or hadron.

# Data pools



Full combinatorics is done automatically. In each event you have  $N$  particles and following the user pattern all combinations  $(1,N)$ ,  $(2,N)$ ,  $(3,N)$  or  $(4,N)$  are done. The limit ( $N < 30$  of a given particle type) is for performance reasons only.

You can define in parallel as many combination patterns as you want and each pattern will have the separate ntuple output to the file.

## Example 7

# HHypPool – particle (but no id) combination

- Open again file: `hhypool.h`

```
private:
```

```
int count(HPattern *ptrC, HParticlePool& refPPool);  
void combine(HPattern *ptrC, HParticlePool& refPPool);  
std::vector<HHypCandidate*> hypSet;
```

- Full combinatorics (N choose k) is done here
- Method "count" calculates how many combinations of a given pattern are possible in a given event (taking particles from HParticlePool)
- Method "combine" does all combinations and stores them in the array of HHypCandidates (here hypSet)

*How the combinatorics is done? Is it  $k$ -element subset of  $N$ -element set with the order irrelevant?*

- The order **is** relevant if you have more than one particle of a given species. Keep in mind that you do your particle combination first on a level of generic particle selection, i.e.  $H^+$ ,  $H^-$ ,  $L^+$ ,  $L^-$  ( $H$  – hadron,  $L$  – lepton).
- For example,  $H^+H^+$  (two positive hadrons) can become i.e. proton-proton (and you need to avoid double counting by selection of one combination) or proton- $\pi^+$  (then it is important which hadron is really proton/pion, 1st or 2nd).
- Open file: [hhyppool.h](#) and see

```
#define COMB_REPETITION 1
```

what means that choosing particles with repetition is active. If you want combinations with no repetition, change 1 to 0



# Data pools

HParticlePool

HHypPool

HPidPool

Container for **HHypCandidates**, but now all particle candidates are assigned to be real particles.

Defining your combination pattern:

```
HPidPool myPids( &outputFile );  
myPids.add("LpLm", "EpEm", ePositron, eElectron);  
myPids.add("HpHp", "PP", eProton, eProton);  
myPids.add("HpHp", "PPip", eProton, ePiPlus);
```

*For example:*

The action between **HHypPool** and **HPidPool** is usually applying user cuts making time (beta) recalculation, PID selection *etc.* You can define as many **HPidPool** objects with various cutting scenarios as you want.

## Example 8

### HPidPool – various id scenarios

- Suppose we have defined combination of 2 positive hadrons.

```
HHypPool myHyps;  
myHyps.add("HH", eHadronPos, eHadronPos);
```

Feeding from the same combination we can define which particles we actually want to have in the combination:

```
HPidPool myPids( &outputFile );  
myPids.add("HH", "PP", eProton, eProton);  
myPids.add("HH", "PPip", eProton, ePiPlus);
```

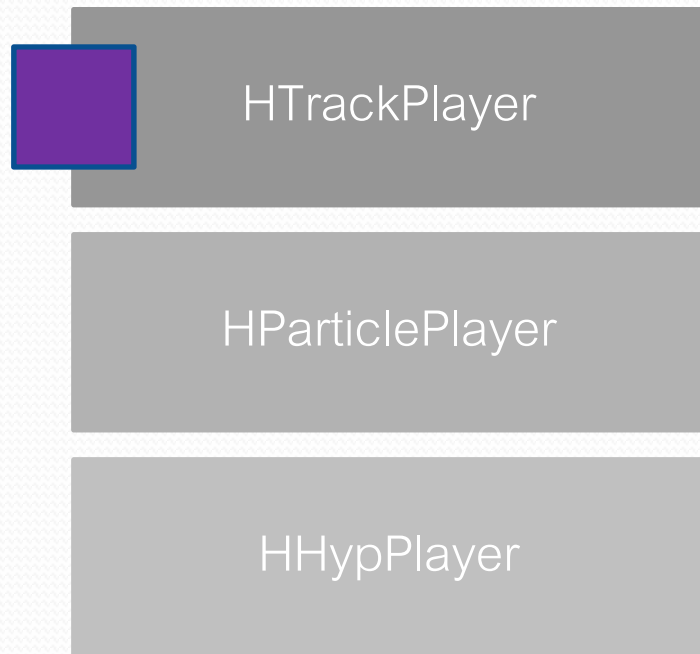
- These "eSomething" names are just enumeration type labels to be found in `hcommondef.h` file: `eUnknown=0`, `ePositron=2`, `eElectron=3`, `ePiPlus=8`, `ePiMinus=9`, `eProton=14`, `eLeptonPos=102`, `eLeptonNeg=103`, `eHadronPos=104`, `eHadronNeg=105` – you can extend the list!

(did I tell you to print that file?) ;-)

*Ok, we have got a bunch of particle combinations, possibly more than one per event. Which is the best?*

- The best combination needs to have a parameter distinguishing it from the other combinations. Usually this is  $\chi^2$  calculated elsewhere (i.e. in a user cut or selection).
- In elementary collisions I reconstruct the time of flight and calculate  $\chi^2$  (provided also detector time resolution). Then in `HPidPool::fill()` I loop over all combinations of a given pattern and select that one with the lowest  $\chi^2$
- Then I add a new data field:  
`eventData.set( "isBest", 1 );` // or 0 if not the best
- Sometimes we want more than one "the best" combination – this happens in the case of acceptance/efficiency matrix production with many white (good) tracks per event. *In this case call me for help (in order to change slightly the code).*

# Reconstructos (players)



Reconstructor acts between DST data (`HPidTrackCand`, `HPidTrackCandSim`) and `HParticlePool`.

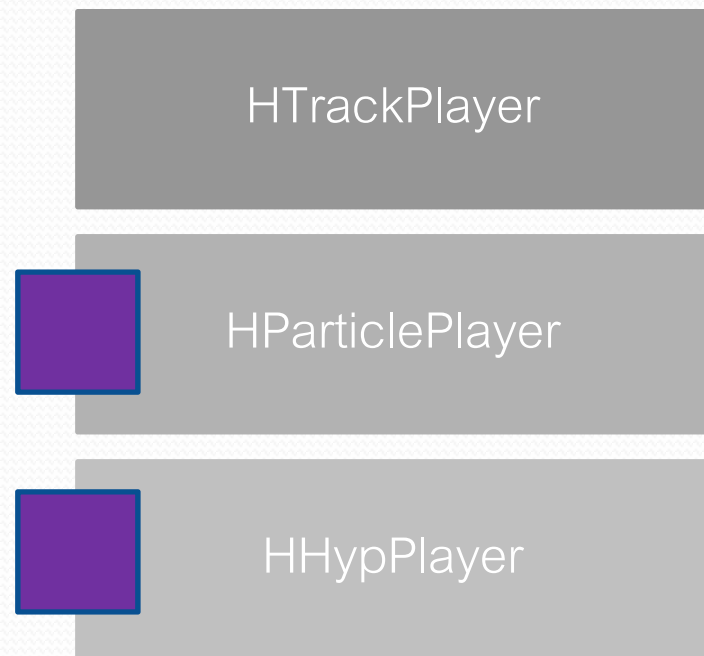
Defining reconstructor:

```
HTrackPlayer * hyp = new HTrackPlayer( myParticles );
```

*For example:*

Usually here you want to define your fine particle (track) candidate selection, i.e. apply some narrower correlation window for your leptons or do some track cleaning.

# Reconstructos (players)



At each reconstructor user can add a list of cuts to be done between data levels.

Reconstructors act between **HParticlePool** and **HHypPool**, and between **HHypPool** and **HPidPool**, respectively.

Defining reconstructors:

```
HParticlePlayer * hyp2 =  
    new HParticlePlayer(myParticles, myHyps);  
HHypPlayer * hyp3 = new HHypPlayer(myHyps, myPids);
```

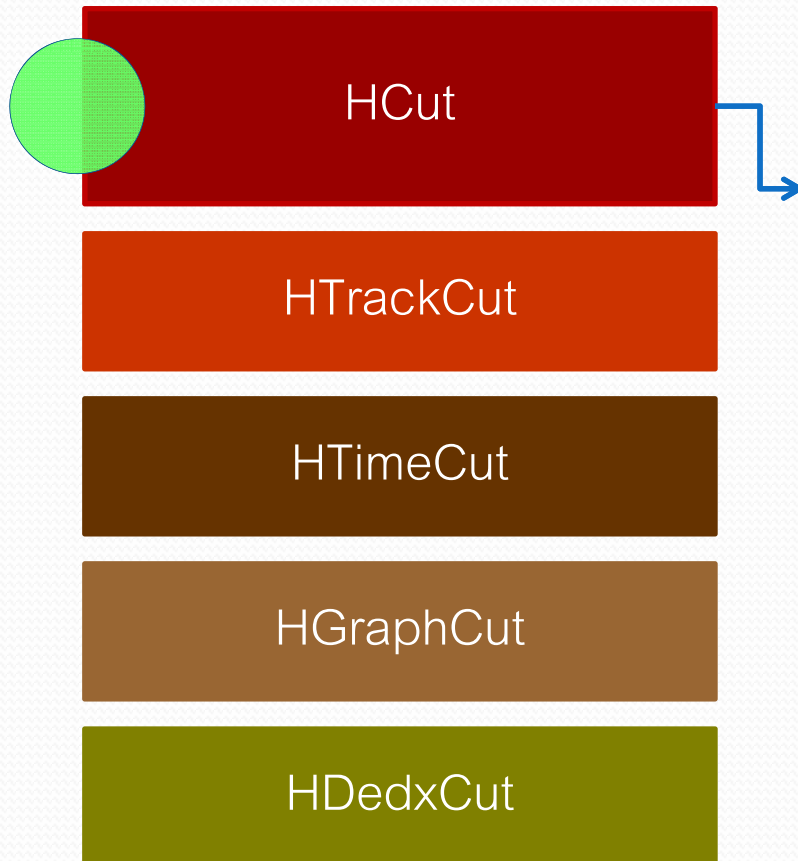
## Example 9

### HHypPlayers – various scenarios

- It is important to understand that a player (reconstructor) is a place where you add your cuts (as many as you want and in any order). It is convenient then to investigate the influence of a given cut just by creation of different final HHypPlayer reconstructors (which target different final HPidPool objects)

- ```
HPidPool myPids_A( &outputFile );  
myPids_A.add("HLpLm", "PEpEm_ID",eProton,ePositron,eElectron);  
HPidPool myPids_B( &outputFile );  
myPids_B.add("HLpLm", "PEpEm_DEDX",eProton,ePositron,eElectron);  
HHypPlayer * hyp3_A = new HHypPlayer(myHyps, myPids_A);  
HHypPlayer * hyp3_B = new HHypPlayer(myHyps, myPids_B);  
hyp3_A->add( tCut1 ); // tCut1 will be a graphical PID cut  
hyp3_B->add( tCut2 ); // tCut2 will be a graphical DEDX cut  
gHades->getTaskSet(context)->add(hyp3_A);  
gHades->getTaskSet(context)->add(hyp3_B);
```

# Cuts



**User is responsible to write his/her own cuts (algorithms how to deal with data).**

Base (interface) class. It can open and retrieve as many files and cuts (TH1\* and TCutG\* objects) as you may need.

Cuts can be applied in any order and at any reconstructor (that is, between two consecutive data levels).

Some standard cuts are provided.

# Cuts

HCut

HTrackCut

HTimeCut

HGraphCut

HDedxCut

Cuts have labels! If you choose "all" the cut will be applied to all patterns (combinations) of a given data pool. If you name it i.e. "EpEm" it will be applied only to a pattern with the same label!

Defining and adding cuts:

```
HTrackCut tCut("all");  
HGraphCut tCut2("all", "M3_PIDCUTS.root");  
hyp->add( tCut );  
hyp3->add( tCut2 );
```

*For example:*

Cuts are written in the user directory, all have to be derived from **HCut** class. The user links with the **libPat.so** library. No changing the base code! (future: not during this training)



## Example 10

# HTrackCut – fine hadron/lepton selection

- The cut is meant to act on `HParticlePool`
- `HParticlePool` objects are filled from `HPidTrackCand` but the lepton selection is based on quite large correlation window (RICH–MDC).
- It is necessary to make a finer correlation. User has to draw  $\Delta\theta$  versus  $\sin\theta \cdot \Delta\varphi$  (where  $\Delta\theta = \theta_{\text{RICH}} - \theta_{\text{MDC}}$ ,  $\Delta\varphi = \varphi_{\text{RICH}} - \varphi_{\text{MDC}}$ ) and prepare parameterization of an elliptic correlation window (as a function of momentum)
- `HTrackCut` checks the correlation for lepton candidates and if there is no track correlation with RICH ring the candidate becomes again a hadron candidate.
- If you open `hhypdata.h` file you will see a lot of numbers – this is some parameterization for `pp@1.25 GeV`, prepare your own!

## Example 10

# HTimeCut – time reconstruction

- The cut is meant to act on `HPidPool`
- In elementary reactions measured by means of a HADES spectrometer we often ;- ) had no START detector therefore no absolute time calibration
- Only relative time between particles is available therefore at least two particles are necessary (in experiment this is the case but in simulation with "single tracks" – not)
- All particles can be taken as "reference" particle but with some priorities (hit in TOF better than in TOFino, lepton better than hadron)
- **Important:** in this cut I calculate  $\chi^2$  (chiz) which is a parameter of further the best combination selection. If you do not reconstruct the time you have to calculate  $\chi^2$  anyway.

## Example 11

# HGraphCut – graphical cut for PID

- The cut is meant to act on HPidPool
- Definition of the object requires also a file containing the cuts  
`HGraphCut tCut("all","MY_PIDCUTS.root");`
- HGraphCut has usually pointers i.e. `TCutG *p_cut`; which are assigned to cuts read from file in the constructor, i.e. `p_cut = getCut("p_cut");`
- Selection is just based on a check  
`if (p_cut) return p_cut->IsInside( beta, mom );`  
for all particles in a combination.
- If any of these checks fails, the HHypCandidate object is deactivated: `pHyp->setActive( false );`
- Exactly the same for **HDedxCut**
- Hint: name your cuts with `p_cut`, `ep_cut`, `em_cut` and so on, and you even do not need to change the code as it is now available.

# Life example

This is only part  
of the code  
relevant to  
the PAT  
configuration.

The rest see  
**main.cc**

```
HOutputFile outputFile( "test.root" , "recreate" );
```

```
HParticlePool myParticles; //HParticlePool myParticles( &outputFile );  
myParticles.add("all",eHadronPos,eHadronNeg,eLeptonPos,eLeptonNeg);
```

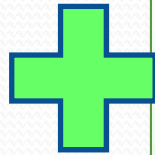
```
HHypPool myHyps; //HHypPool myHyps( &outputFile );  
myHyps.add("HLpLm",eHadronPos,eLeptonPos,eLeptonNeg);  
myHyps.add("HLpLp",eHadronPos,eLeptonPos,eLeptonPos);  
myHyps.add("HLmLm",eHadronPos,eLeptonNeg,eLeptonNeg);
```

```
HPidPool myPids( &outputFile );  
myPids.add("HLpLm", "PEpEm",eProton,ePositron,eElectron);  
myPids.add("HLpLp", "PEpEp",eProton,ePositron,ePositron);  
myPids.add("HLmLm", "PEmEm",eProton,eElectron,eElectron);
```

```
HTrackPlayer * hyp = new HTrackPlayer( myParticles );  
HParticlePlayer * hyp2 = new HParticlePlayer(myParticles, myHyps);  
HHypPlayer * hyp3 = new HHypPlayer(myHyps, myPids);
```

```
HTrackCut tCut("all");  
HGraphCut tCut2("all","M3_PIDCUTS.root");  
hyp->add( tCut );  
hyp3->add( tCut2 );
```

# S u m m a r y



- PAT can be used in any analysis where you have a certain particle (1-4) combination
- There is full event information available you can reconstruct your data on event basis
- Easy to adjust any kind of reaction to analyze
- All reactions processed in one run
- You can write the output to more than one file
- Full combinatorics done automatically
- Easy to add any variables to be stored
- Time recalibration option, PID cuts, dEdx cuts...
- Output: optional on any level: ntuples

- If you want to investigate more than one combination in parallel (i.e. make them dependent event by event) it is possible but you have to write your piece of software (FAT – final analysis tool 😊) which reads ntuples entry by entry, checks event number and does the correlation (selection).