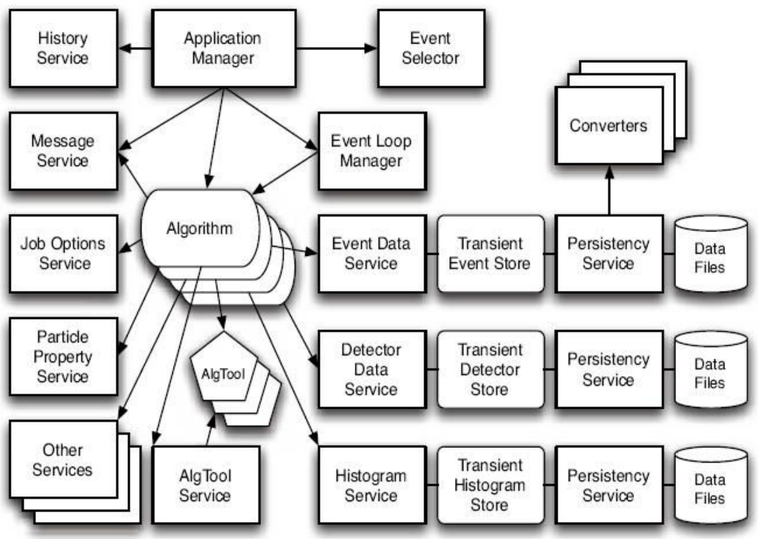
The ATLAS Data Model



Peter van Gemmeren (ANL): ANL Analysis Jamboree

Data Flow at ATLAS

RAW:

- Original data at **Tier-0**
- Complete replica distributed among all **Tier-1**

ESD:

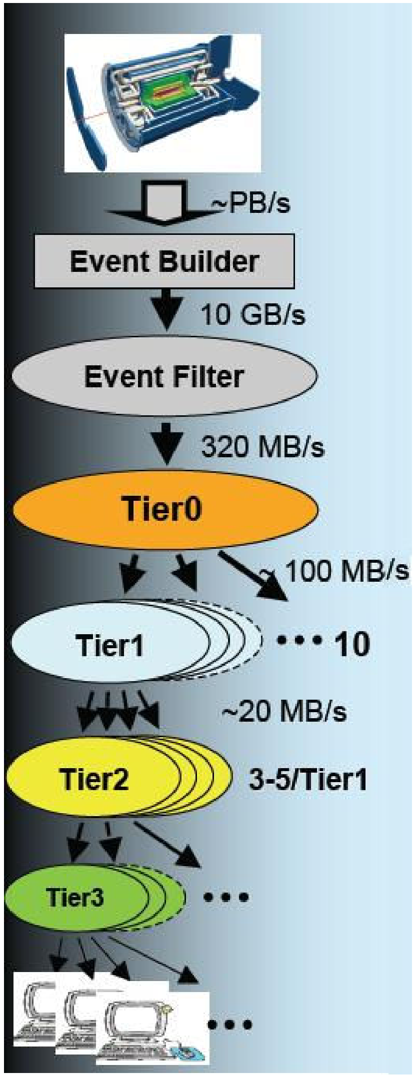
- ESDs produced by primary reconstruction reside at **Tier-0** and are exported to 2 **Tier-1s**
- Subsequent versions of ESDs, produced at **Tier-1s** (each one processing its own RAW), are stored locally and replicated to another **Tier-1**, to have globally 2 copies on disk

AOD:

- Completely replicated at each **Tier-1**
- Partially replicated to **Tier-2s** (depending on each **Tier-2** size) so as to have at least a complete set in the **Tier-2s** associated to each **Tier-1**
- Every **Tier-2** specifies which datasets are most interesting for their reference community; the rest are distributed according to capacity

TAG:

- TAG files or databases are replicated to all **Tier-1s** (Root/Oracle)
- Partial replicas of the TAG will be distributed to **Tier-2** as Root files
- Each **Tier-2** will have at least all Root files of the TAGs that correspond to the AODs stored there



Analyzing the Data

- Inside **Athena (RAW, RDO, ESD, AOD, DPD, TAG)**
  - Interactive OR batch using C++, python code.
  - Provides **full access** to all tools and services.
  - Can submit to the **grid**.
- Outside **Athena (DPD, and to some degree ESD, AOD)**
  - using **ROOT** (to at least read)
  - **CINT**, or using python, or compiled C++ code.
  - Does **not need full Athena** installation (expected 1GB)
  - **Not all classes** are available (example, calo-Cells)
- **Important**: both methods use the **same files as input**.

Athena/Gaudi components

- All levels of processing of ATLAS data, from high-level trigger to event simulation, reconstruction and analysis, can take place within the Athena framework.
- The major components of Athena are:
  - **Services**. A `Service` provides services needed by the Algorithms. In general these are high-level, designed to support the needs of the physicist. Examples are the message-reporting system, different persistency services, random-number generators, etc.
  - **Algorithms**. Algorithms share a common interface and provide the basic per-event processing capability of the framework. Each `Algorithm` performs a well-defined but configurable operation on some input data, in many cases producing some output data.
  - **AlgTools**. An `AlgTool` is similar to an `Algorithm` in that it operates on input data and can generate output data, but differs in that it can be executed multiple times per event. Each instance of a `AlgTool` is owned, either by an `Algorithm`, a `Service`, or by default by the `ToolSvc`.

Common Services

- There are quite a few Services in Athena to help you:
  - **Job Option Service**. The **JobOptionSvc** is a catalogue of user-modifiable properties of Algorithms, AlgTools and Services. As an **example**, the value of a property called "`CutOff`" in the `JetMaker` can be set either from a job-option file or from the Athena interactive prompt by:

```
JetMaker.CutOff = 0.7
```

    Default values are set in the Algorithms, AlgTools or Services itself.
  - **Logging**. The **MessageSvc** controls the output of messages sent by the developers using a `MsgStream`. The developer specifies the source of the message (its name) and the message verbosity level. The MessageSvc can be configured to filter out messages coming from certain sources or having a high verbosity level.
  - **Performance Monitoring**. The **AuditorSvc** and the **ChronoStatSvc** manage and report the results of a number of Auditor objects, providing statistics on the CPU and memory usage (including potential memory leaks) of Algorithms and Services.

And of course, StoreGate

- **StoreGate** is the Athena implementation of the **blackboard**.
- StoreGate allows a module (such as an algorithm, service or tool) to use a **data object** (like for **example `Jet`**, `Track` or `Cell`) **created by an upstream module** or **read from disk** transparently.
  - ~~A proxy defines and hides the cache-fault mechanism: upon request, a missing data object instance can be transparently created and added to the transient data store, retrieving it from persistent storage on demand.~~
    - *On second thought I am sure you don't want to know this.*
- StoreGate allows object identification via data **type** and **key** string.
  - In ATLAS data objects like `Jet`, `Track` or `Cell` are stored in container (*think STL vector, or fancy array*) called `JetCollection` or `TrackCollection` .
- StoreGate supports **base-class and derived-class** retrieval, **key aliases**, and **inter-object references**.
    - *Just say "Wow!"*

StoreGate storing DataObjects: record()

- Object (**example**):

```
MissingET* met = new MissingET();
met->setEtSum(arg);
…
StatusCode status = m_storeGate->record(met, key
                                   /*, bool allowMods = true */);
// check status…
```

- Container (**example**):

```
MyJet* jet1 = new Jet();  // create new Jet objects
MyJet* jet2 = new Jet();
jet1->set4Mom(arg);       // setting the attributes of the Jets
jet2->set4Mom(arg);
…
JetCollection* jetColl = new JetCollection();
jetColl->push_back(jet1); // pushing Jets into a container
jetColl->push_back(jet2);
…
StatusCode status = m_storeGate->record(jetColl, key, false); // locked
// check status…
```

StoreGate storing DataObjects: retrieve()

- Object (**example**):

```
// Most objects are recorded as const
/*const*/ MissingET* met;
StatusCode status = m_storeGate->retrieve(met, key);
// check status…
met->setEtSum(arg);       // works only if not const
val = met->getEx();       // should always be OK
…
```

- Container (**example**):

```
const TrackCollection* trackColl;
StatusCode status = m_storeGate->retrieve(trackColl, key);
// check status…
for (it = trackColl->begin(), itEnd = trackColl->end();
                                                    it != itEnd; it++) {
   // do something with (*it), which is a Track
   …
}
```

Skip

StoreGate: SymLinks and Aliases

- StoreGate supports base-class and derived-class retrieval via **symLinks**.
  - e.g.: `CaloCell` is base class to `TileCell`:

```
status = m_storeGate->symLink(tCell, cCell);
status = m_storeGate->symLink(ClassID_traits<TileCell>::ID(), key,
                              ClassID_traits<CaloCell>::ID());
```

  - Creates symlink from `TileCell` to its base class and allows:

```
const CaloCell* bCell = new CaloCell(); // works for LAr and Tile
StatusCode status = m_storeGate->retrieve(bCell, key);
// check status…
cellE = bCell->energy();
```

- StoreGate supports **key aliases**:

```
status = m_storeGate->setAlias(tCell, "PetersFavorite");
```

Persistency: From StoreGate to Eternity… (and back)

- **The only thing more exciting than finding the Higgs is writing it to disk!**
  - Ok maybe not. Anyway, it still needs to be done.
- Items in **StoreGate** can be written to POOL/ROOT file using the **Athena/Pool I/O infrastructure** (*my day job*).
- Existing types (like for **example** `Jet`, `Track` or `Cell`) can be written to disk by adding

```
OutputStream.ItemList += [ "JetCollection#PetersFavorite" ].
```

  to the jobOptions file.

- New types need **converter** and **persistent state representation** (*somewhat harder, did I mention my email?*).
- Check: `Database/AthenaPOOL/AthenaPoolExample`

Athena Algorithms (1): Interface

- If you want to do a more complex analysis, you will want to use Athena and need to provide an algorithm.

- Algorithms perform a **well-defined** but **configurable** operation on some input data and may produce output data.
- **Common interface** provided by Gaudi: **`IAlgorithm`**
- Implemented in Gaudi/Athena as **`Algorithm`**, the common base class for Algorithms.
- Can use **Services** (e.g., **`StoreGateSvc`**) and **AlgTools** via 'Handles'.

- Next slide **example**: **`JetMaker`** ->

Athena Algorithms (2): Implementation header (in src)

```
#include "GaudiKernel/Algorithm.h"
#include "GaudiKernel/ServiceHandle.h"

class StoreGateSvc;       // Forward declaration

class JetMaker : public Algorithm {

public: /// Gaudi boilerplate
   /// Constructor with parameters:
   JetMaker(const std::string& name, ISvcLocator* pSvcLocator);
   /// Destructor:
   virtual ~JetMaker();
   virtual StatusCode initialize();
   virtual StatusCode finalize();
   virtual StatusCode execute();
…
private: /// Handle to use services e.g., StoreGate
   ServiceHandle<StoreGateSvc> m_storeGate;
   /// cutOff (e.g.) property, configurable by jobOptions
   DoubleProperty m_cutOff;
};
```

Athena Algorithms (3): Implementation source

```
#include "JetMaker.h"

JetMaker::JetMaker(const std::string& name, ISvcLocator* pSvcLocator) :
   Algorithm(name, pSvcLocator), m_storeGate("StoreGateSvc", name) {
    // Property declaration (label, variable, description)
   declareProperty("CutOff", m_cutOff, "KT Jet cut off parameter");}
JetMaker::~JetMaker() {}
StatusCode JetMaker::initialize() {
    // Get handle for StoreGateSvc and cache it:
    StatusCode status = m_storeGate.retrieve();
   // check status
    if (!status.isSuccess()) {
       // get message service
       MsgStream log(msgSvc(), name());
       // log error message
       log << MSG::ERROR << "Unable to retrieve StoreGateSvc" << endreq;
       return(StatusCode::FAILURE);
    }
…
    return(status);
}
```

Athena Algorithms (4): Implementation source

```
StatusCode JetMaker::finalize() {
    StatusCode status = m_storeGate.release();
   // check status…
…
    return(status);
}

StatusCode JetMaker::execute() {
    // Do the real work once for each event
   const TrackCollection* trackColl;
   StatusCode status = m_storeGate->retrieve(trackColl, key);
   // Let's use those tracks to make our very own jets
   …
   JetCollection* jetColl = new JetCollection();
   // pushing Jets into a container
   StatusCode status = m_storeGate->record(jetColl, "PetersFavorite");
   // check status…
…
    return(status);
}
```

Athena AlgTools (1): Interface

- AlgTools operate on **input data** and can generate **output data**, it can be **executed multiple times per event**.
- Can be called by an **`Algorithm`** using an interface **`I<AlgToolName>`**
- There can be **multiple implementations** of the same interface.
  - E.g.: an IJetMakerTool could have two concrete implementation as KTJetMakerTool and ConeJetMakerTool.
  - Using the interface will allow the Algorithm to be configured to use either KT or Cone.

Athena AlgTools (2): Implementation header (in src)

```
#include "GaudiKernel/AlgTool.h"
#include "<dir>/IJetHelper.h"
class StoreGateSvc;

class MyJetHelper : virtual public IJetHelper, public AlgTool {
public: /// Gaudi boilerplate
   /// Constructor with parameters:
   MyJetHelper(const std::string& type, const std::string& name,
               const IInterface* parent);
   virtual ~MyJetHelper();
   StatusCode initialize();        // called once, at start of job
   StatusCode finalize();          // called once, at end of job

public: // AlgTool functionality to be implemented by all IJetHelper
   virtual double helpWork(double arg) const;
…
private: /// Handle to use services e.g., StoreGate
   ServiceHandle<StoreGateSvc> m_storeGate;
…
};
```

Athena AlgTools (3): Implementation source

```
#include "MyJetHelper.h"
#include "GaudiKernel/IToolSvc.h"

MyJetHelper::MyJetHelper(const std::string& type, const std::string& name,
   const IInterface* parent) : AlgTool(type, name, parent),
   m_storeGate("StoreGateSvc", name) {
    // Property declaration
    // Declare IJetHelper interface
    declareInterface<IJetHelper>(this);
}
MyJetHelper::~MyJetHelper() {}
    StatusCode MyJetHelper::initialize() {
    StatusCode status = ::AlgTool::initialize();
    // check status…
    // Get handle for StoreGateSvc and cache it:
    status = m_storeGate.retrieve();
    // check status…
…
    return(status);
}
```

Athena Algorithms (4): Implementation source

```
StatusCode MyJetHelper::finalize() {
   StatusCode status = m_storeGate.release();
   // check status…
…
   return(::AlgTool::finalize());
}
double MyJetHelper::helpWork(double arg) {
   // Do the real work each time called
   // Use m_storeGate to retrieve/record data objects to EventStore
…
   return(status);
}
```

- Using AlgTools in Algorithms is similar to using Services:

```
.h:           ToolHandle<IJetHelper> m_helper; // Hold ToolHandle
.cxx, c'tor:  m_helper("MyJetHelper"),  // Init to default AlgTool
              // Allow jobOption to configure any IJetHelper
              declareProperty("HelperTool", m_helper);
```

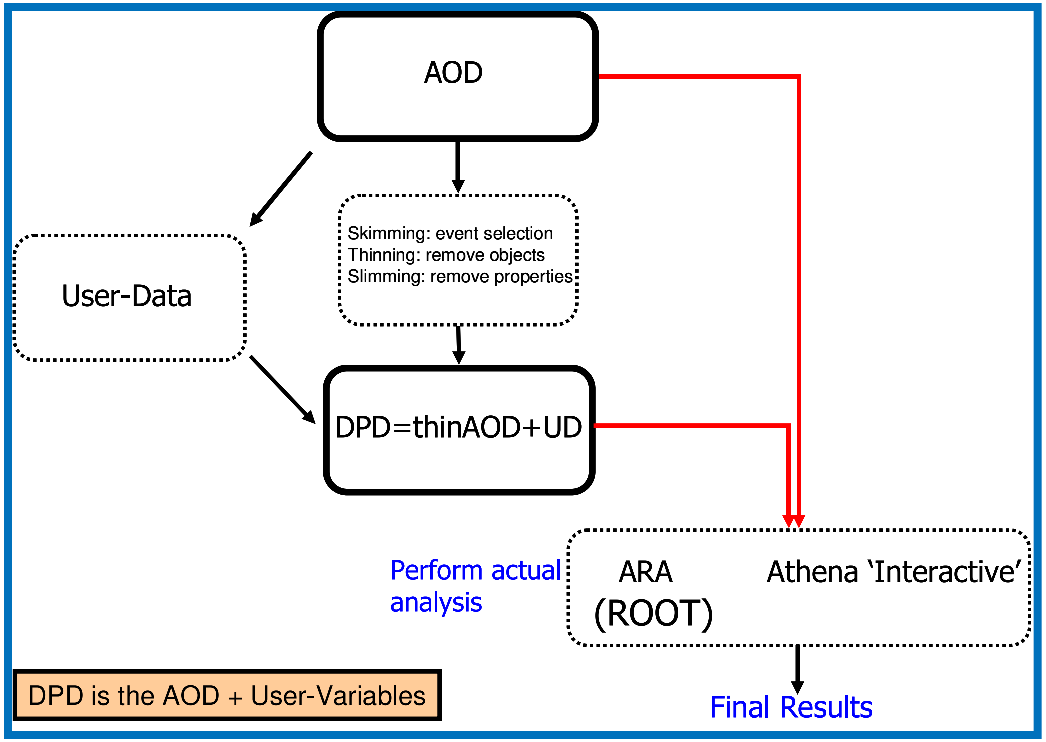
DPD Making

OK, shifting gears

Skimming, Thinning, Slimming… :

Skimming is writing a sub-set of events

- e.g., all **events** containing 1 or 2 `electrons` within a certain eta and with a minimum pT.
- Done using **TAGs**.



Thinning[1] (aka "poor mans' Thinning") is removing collections

- e.g., keep only `electron` **container** but not `muons`.
- Here one would modify the **ItemList** (in the jobOptions).

Thinning is removing objects from a container

- e.g., keep only good `electron` **objects**.

Slimming is removing quantities or sub-objects from an object

- e.g., keep only eta and pT.

All kinds of $D^N PD$…

- **Primary $D^1 PD$:** POOL-based DPD produced by the **GRID production** system. There are expected to be O(10) primary DPDs, so the contents will not be very specific to an analysis. It is expected to be **skimmed**, **slimmed**, and **thinned** compared to the AOD.
  - An Example Job Options file AODtoDPD.py (see CVS)
  - TauDPDMaker
  - BPhysicsDPDMaker
- **Secondary $D^2 PD$:** POOL-based DPD with **more analysis-specific** information. Typically, this is produced from Primary DPD and may be created using an Athena tool like **EventView**.
  - SimpleThinningExample
  - HighPtViewDPDThinningTutorial
- **Tertiary $D^3 PD$:** Does not need to be POOL-based, it includes **flat ntuples**.

AthenaROOTAccess

- Allows to read an **AOD in ROOT** like you would read a normal ntuple (without using Athena).
  - However it uses the **transient classes** and **converters** of the ATLAS software so a portion of the offline is needed.
  - A ~1GB distribution including Athena libraries .
  - **Not all Athena classes** can be called from ROOT: jobOptions, configurables, databases, geometry etc. are not reachable from ROOT - so athena code access has to be limited to all those classes not requiring configuration, **Detector Description etc**.
  - The user can also write **Athena tools**, applications that read the AOD which appears now as a **ROOT tree**.
- One can use **identical code/tools** to run on ESDs, AODs, DPDs.
- One can use any Analysis Framework to access the DPDs (ROOT, Athena batch, Athena interactive)
- The **names of the variables** in the AOD ROOT tree are the same as in the AOD.

AthenaROOTAccess Examples

- CINT macros
  - **Easy development** (change code and run),
  - Run time is **slow** ~x10 C++ compiled code
- C++ compiled code
  - **Slower development** (change code, recompile, cannot reload libs)
  - **Fastest runtime**
  - **Integrates easily** back into Athena
- Python scripts
  - **Easy development** (change code, reload and run)
    - *But no help from the compiler to find bugs either!*
  - Simple example shows runtime ~x3 C++ compiled code
    - *May be able to compile Python*
  - **Integration** of developed code into Athena?
- Examples on **TWiKi** and in **Release**:
  - https://twiki.cern.ch/twiki/bin/view/AtlasProtected/AthenaROOTAccess
  - PhysicsAnalysis/AthenaROOTAccessExamples

PhysicsAnalysis/AthenaROOTAccessExamples

- Need **python** script to **open file** and **setup transient tree**:

```
lxplus:~> get_files AthenaROOTAccess/test.py
```

- **Compiled C++ Example:**

```
lxplus:~> root
root [0] TPython::Exec("execfile('test.py')");
root [1] CollectionTree_trans = (TTree*)gROOT->Get
                                              ("CollectionTree_trans");
root [2] ClusterExample ce; // Example class in AthenaROOTAccessExamples
root [3] ce.plot(CollectionTree_trans);
root [4] TruthInfo ti;
root [5] ti.truth_info(CollectionTree_trans);
```

  - The `test.py` script takes about ~20 seconds to load necessary dictionaries
  - One can recompile and then restart from the beginning

PhysicsAnalysis/AthenaROOTAccessExamples

CINT Example:

```
lxplus:~> root
root [0] TPython::Exec("execfile('test.py')");
root [1] CollectionTree_trans = (TTree*)gROOT->Get
                                        ("CollectionTree_trans");
root [2] gROOT->LoadMacro
                 ("AthenaROOTAccessExamples/macros/cluster_example.C");
root [3] plot(CollectionTree_trans);
```

- One can now edit **`cluster_example.C`** and re-run LoadMacro

Python Example:

```
lxplus:~> python -i test.py
>>> import AthenaROOTAccessExamples.cluster_example
>>> AthenaROOTAccessExamples.cluster_example.plot(tt)
```

- One can now edit **`cluster_example.py`** and re-run:

```
>>> reload(AthenaROOTAccessExamples.cluster_example)
>>> AthenaROOTAccessExamples.cluster_example.plot(tt)
```

Conclusion

- Choose the right tool for the job (*Can't fix TileCal power supplies with a chain saw or install an endcap using a microscope*).
- **Athena** is very well suited complex analyses:
  - Provides common Services and Tools:
    - ***StoreGate*** *helps you exchanging data.*
    - ***Persistency*** *allows you to easily store complex data objects (and read them back even after a possible change of the class).*
    - ***MessageSvc***, ***Auditors***, ***ChronoStatSvc***, *etc. help you to design efficient, robust and well performing* ***Algorithm*** *to do your analysis task.*
  - Establishes Event Data Model:
    - *Many classes for physics objects are defined for you.*
      - Including **Dictionary**, **Converter** and **persistent state representation**.
  - Lots of functionality to help physicists develop their analysis
    - *Can be overwhelming, so start out with the basics only.*
- **AthenaROOTAccess** implements parts of athena's analysis support
  - More light weight framework
  - Fast turn around