A new mechanism to use the Conditions Database REST API to serve the ATLAS detector description

A. Formica (IRFU, Université Paris-Saclay), R.M. Bianchi (Pittsburgh), A. De Salvo (INFN - Roma “Sapienza”) on behalf of the ATLAS Collaboration

CHEP 2018 - Sofia, Bulgaria - 9–13.07.2018
Goals of the exercise

- **Profit of recent developments in detector description**
  - Standalone mode for geometry software library
  - Introduction of new serialization formats (SQLite/JSON)

- **...in order to:**
  - Provide a lighter software stack in Athena for geometry access via a simple REST service
  - Reduce the amount of network operations to serve the geometry to clients
Outline

- **Geometry DB:**
  - Oracle ATLASDD (Detector Description)
- **Loading and building the geometry today**
  - Athena “architecture” to gather parameters from Oracle
- **GeoModel : recent evolutions and alternative persistency formats (SQLite, JSON dump)**
- **Prototype for alternative storage and access**
  - JSON geometry dump as Conditions Data type
  - Access using HTTP REST API : covers both Athena and standalone use cases
ATLAS Detector Description today

- ATLAS experiment uses the GeoModel [1] library to describe the **geometry tree**: a set of nodes (shapes, containers, materials, ...) connected through relationships of different types.

- Parameters are used to further customize the base shapes/structure used in the GeoModel tree; they are stored in the Oracle-based **Geometry DB** [2].

- Different sets of parameters can be saved in the Geometry DB, to store different configurations of the detector geometry.

- The GeoModel tree is **computed on-the-fly** when requested through the experiment software framework **Athena**, and it is stored **in memory** only.
Oracle-based Geometry DB: ATLASDD

- This database [2] consists of a set of tables dedicated to parameters for individual detector elements (the “nodes”). Each set of parameters is identified via a dedicated tag.

- A set of “valid” tags at a given moment is identified via a unique parent tag ("geometry tag").

- Schema implementation is based on a hierarchical versioning.

- Geometry parameters are accessed via SQL queries, starting from a unique entry point which is the geometry tag.
Access to geometry parameters from Athena

- Today we do access geometry parameters from Oracle ATLASDD schema via a set of SQL queries: ~300 queries needed to retrieve the full set of parameters for a given geometry tag...so every job needs to run those queries.

- A dedicated service is capable of “interfacing” GeoModel to the Oracle DB. All queries are performed via Coral (C++ DB access layer) and executed via Frontier/SQUID (in the context of distributed computing) [3].
A standalone geometry description

- Being described in C++ code and built on-the-fly, the geometry was not queryable so far
  - the Geometry DB only stores shapes’ parameters; all relationships between nodes are stored inside the C++ description only.

- The tight integration of the geometry mechanism with the ATLAS software framework, prevented a lightweight access to the experiment’s geometry. One notable use-case is interactive data visualization.

- The recent decoupling of the GeoModel library from the framework, and the possibility of dumping the full geometry to flat-file DBs, made possible the exploration of alternative methods to access, read, explore, debug, and visualize the experiment’s geometry.

See CHEP-2018 talk from Sebastian Merkt [9] for details on recent GeoModel development
From Oracle to SQLite and JSON

- The possibility of dumping the geometry information into a persistent copy, opened up many possibilities to further development.

- **Two new exporters** have been implemented: **SQLite** and **JSON**
  - **SQLite**: compact, with a data model optimized for the smallest file size, easy to share - size: ~50 Mb
  - **JSON**: human-readable and easy to be split in multiple files (subsystems) - size: ~O(100 Mb), based on a draft version of the JSON format (*work in progress*)

- We can **dump the full detector** description from memory into new data structures for each “geometry tag” (e.g.: one JSON file per one geometry tag).

- Of course, saving a full geometry description for each tag leads to redundant information, but it gives us the possibility of having a **persistent copy of a common geometry version**, to be saved, shared, served, explored and debugged without accessing the experiment framework, **for read-only tasks**.
GeoModel <-> SQLite/JSON

ATLAS SW Framework

GeoModel

Geometry DB

application

~300 Oracle DB calls

TODAY

ATLAS SW Framework

GeoModel

Geometry DB

application

DumpGeo

JSON

SQLite

Geo2Neo

GeoRead (C++ lib)

CREST Cond DB

Neos graph DB

~O(1) HTTP call

PROTOTYPE

Standalone application
$ sqlite3 geometry_simplestToyDetectorFactory.db

SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> .headers ON
sqlite> .tables
AlignableTransforms  LogVols              SerialDenominators
ChildrenPositions    Materials            SerialTransformers
FullPhysVols         NameTags             Shapes
Functions            PhysVols             Transforms
GeoNodesTypes        RootVolume           dbversion
sqlite> select * from GeoNodesTypes;
id|nodeType|tableName
1|GeoPhysVol|PhysVols
2|GeoFullPhysVol|FullPhysVols
3|GeoLogVol|LogVols
4|GeoMaterial|Materials
5|GeoShape|Shapes
6|GeoSerialDenominator|SerialDenominators
7|Function|Functions
8|GeoSerialTransformer|SerialTransformers
9|GeoTransform|Transforms
10|GeoAlignableTransform|AlignableTransforms
11|GeoNameTag|NameTags
sqlite> select * from RootVolume;
1|1|1
sqlite> select * from PhysVols;
id|logvol|parent
1|NULL|1
2|2|2
3|3|2
4|4|3

List of GeoPhysVol nodes

GeoModel tree structure stored in SQLite ("id"s only)

List of GeoShape nodes
Geometry <-> JSON

- The JSON file is organized into two parts:
  - A part listing all the nodes which are part of the persistified tree.
    - Shared nodes are listed once, only
  - A part storing the geometry “tree”

```json
{
  "nodes": {
    "GeoShape": {
      "type": "Box",
      "id": 1,
      "parameters": "XHalfLength=12000;YHalfLength=12000;ZHalfLength=12000"
    },
    ...}
  "tree": {
    "children": {
      "1": {
        "logvol": 2,
        "children": {
          "1": {
            "logvol": 3,
            "children": {
              "1": {
                "type": "GeoPhysVol",
                "logvol": 4,
                "logname": "InnerPassive",
                "id": 4
              },
              ...}
          "logname": "Passive",
          "type": "GeoPhysVol",
          "id": 3
        },
        "logname": "ToyLog",
        "type": "GeoPhysVol",
        "id": 2
      },
      "type": "GeoNameTag",
      "id": 1,
      "name": "Toy"
    },
    ...}
}
```
New ways of building the detector geometry

- Goal of the experiment presented here is the use of the ConditionsDB REST API to efficiently serve the full detector geometry to clients (READ access), without the needs of extra layers.

- This has two main advantages:
  - a reduced number of DB queries for read-only operations (from $O(\sim 300)$ to $O(\sim 1)$)
  - the possibility of accessing the geometry through a REST API, removing the need to include specific SQL knowledge inside the client
REST access to geometry parameters

- While Oracle geometry DB has a table structure well suited for managing the informations, we do not need the same kind of complexity in a **READ-ONLY** use case of a typical *Athena* job.

- **Access simplification** could be beneficial for the core software components that are managing the geometry, and could also simplify the development for standalone applications (outside *Athena*).

- Today we have the possibility to store the full geometry in a SQLite and in a JSON file and read it back: we just need to find the correct place and way to distribute this file!

- **Conditions DB** seems to be a good **place**, and the **JSON** structure is ideal for this.
Storing geometry in Conditions DB

For this prototype we have been using the proposed Crest DB [5] infrastructure that ATLAS is evaluating for the future (beyond “Run3”), but every “conclusion” can be applied to the existing COOL [4] data model.

1. **Geometry JSON** file is represented in the Conditions DB as a BLOB:
   - associated to only 1 IOV (Interval Of Validity spanning 0 to Infinity) ...
   - ... and to a single TAG (which could correspond to the geometry tag).

2. In order to store and retrieve a sample of the geometry JSON we have been using the Crest prototype (deployed in Openshift [6] at CERN). For convenience we have also used existing python client libraries in order to interact with the Crest server via REST.

Accessing geometry architecture overview
A REST client for GeoModel

- Crest REST API has been specified via Swagger ([7] OpenApi spec, a sort of IDL for REST services): this allow us to profit from code generation tools.

- We have generated a C++ Qt based client library that allow us to interact with the Crest server via HTTP (REST API). This is useful as a client for standalone applications using GeoModel.

- API Examples from swagger documentation and CLI client ([8]):
Possible gains in Athena

- **Reduce** the amount of **HTTP traffic** from clients: from $O(100)$ queries/job to $O(1)$

- **Simplify** Athena service: read geometry using something very close to the service reading conditions data. This will obviously decrease the **maintenance** effort.

- **Reduce** dependency on SQL libraries: the typical job should not even be aware that is accessing a DB or a filesystem. All SQL would exists only for the management of the geometry parameters in Oracle (or other relational DB), which is a completely different use-case from loading a geometry in memory for reconstruction jobs.

- Knowing the geometry tag we can **access the full geometry with one HTTP call**: e.g. [http://crest-undertow.web.cern.ch/crestapi/payloads/17c2b6846852c503a4d653d9cec8ec6ac475f50744545020a571c80c5d3d71b9](http://crest-undertow.web.cern.ch/crestapi/payloads/17c2b6846852c503a4d653d9cec8ec6ac475f50744545020a571c80c5d3d71b9)

- **Side remark**: all **conclusions are valid** also if the backend is the existing **COOL DB**
Plans

- Measure **performance**: access geometry in Athena in both ways (as we do today or via an HTTP call from conditions DB)

- Needs a **full JSON dump** (with all GeoModel shapes JSON-persistified)

- **Evaluate**, based on the performance results, how to proceed for core software development

- **Explore REST** approach for **other future geometry backends** (like Neo4j), in order to reduce software dependencies in GeoModel standalone library.
References /1


References /2


- [6]: Openshift @ CERN: https://cern.service-now.com/service-portal/article.do?n=KB0004358

- [7]: Swagger and Swagger-codegen: https://swagger.io

- [8]: Crest server and client on gitlab.cern.ch: https://gitlab.cern.ch/formica/swagger_crestdb

- [9]: S. Merkt @ CHEP2018: https://indico.cern.ch/event/587955/contributions/2937522/

- [10]: L. Rinaldi @ CHEP2018: https://indico.cern.ch/event/587955/contributions/2936820/
The ATLAS GeoModel

GeoModel in a nutshell:

- the GeoModel is a collection of classes which represent “nodes”: shapes, materials, volumes, space transformations, ...
- There are nodes which are simple objects, with only parameters: GeoShape, GeoMaterial, ...
- Some nodes reference other nodes as part of their variables: GeoLogVol, ...
- A class of node can have children: GeoVPhysVol
- Some nodes are used as children: GeoPhysVol, GeoTransform, GeoSerialTransformer, ...

A detector description based on GeoModel has a tree structure, with branches made of such nodes (like a “scene-graph” in 3D graphics)
SQLite storage of the full ATLAS geo

- All GeoModel nodes are stored in a compact way in a SQLite file.
- All relationships are stored as well:
  - nodes using other nodes
  - mother-daughter and the position in the sub-tree, in a link table
- Full ATLAS geometry is stored in ~50 Mb only!
- JSON follows the same schema, but more verbose (human-friendly)

Number of table rows