A scalable new mechanism to store, query and serve the ATLAS detector description through a REST web API

R M Bianchi[1,†] **and I Vukotic**[2,†]

[1] University of Pittsburgh, Department of Physics and Astronomy, 100 Allen Hall, 3941 O'Hara St, Pittsburgh PA 15260, US

[2] University of Chicago, High Energy Physics, 5640 S Ellis Ave, Chicago IL 60637, US

[†]on behalf of the ATLAS Collaboration

E-mail: riccardo.maria.bianchi@cern.ch

Abstract. Until now, geometry information for the detector description of the ATLAS experiment was only defined in C++ code, stored in online relational databases integrated into the experiment's frameworks or described in files with text-based markup languages. In all cases, to build and use the complete detector geometry, a full software stack was needed. In this paper, we present a new and scalable mechanism to store the geometry data and to query and serve the detector description data through a web interface and a REST API. This new approach decouples the geometry information from the experiment's framework. Moreover, it provides new functionalities to users, who can now search for specific volumes and get partial detector description, or filter geometry data based on custom criteria. We present two approaches to build a REST API to serve geometry data, based on two different technologies used in other fields and communities: The graph database Neo4j and the search engine ElasticSearch. We describe their characteristics, and we compare them in a HEP context.

1. Introduction

Every modern particle physics experiment is based on a sophisticated particle detector, usually composed of many different sub-detectors, each of them targeting a specific measurement. The so-called *Detector Description* is the piece of data containing the detailed description of all the sub-detectors, their position in the 3D space and their relationships with the other components of the whole detector.

The Detector Description of the ATLAS Experiment [1] is based on the GeoModel [2] library, which lets the developers define the geometry in the C++ code, using *nodes* (shapes, transformations, materials, and so forth) to describe the structure and the characteristics of the detector geometry.

Until now, the ATLAS geometry was stored in-memory only, built on the fly by the software framework when a client asked for it, taking the required information both from C++ code and from online databases. Hence, the geometry data of the experiment could only be accessed and interpreted through the experiment's software framework using scripts or compiled code, making it impossible to interactively query and explore the detector geometry and to use it in stand-alone or cross-platform applications.

Building upon the earlier work [3] on a persistent representation of the geometry data, this paper describes the efforts recently made to design and develop a new web-based REST API service to store, query and serve the Detector Description of the ATLAS Experiment, in order to make the experiment's geometry data accessible and usable outside the experiment's framework.

2. The ATLAS detector description and the GeoModel library

The ATLAS experiment uses the GeoModel library to describe the volumes and structure of the geometry tree in C++ code. Figure 1 shows two examples of basic GeoModel trees. A *logical volume* (Figure 1(a)) defines a fundamental entity within the tree: A logical volume has a name, and it references a *shape node* and a *material node*, to define a single abstract type of object. This abstract entity is then used to build a concrete *physical volume*, placed in the 3D space. The same logical volume can be used to define multiple physical volumes, and other kinds of nodes can be used to place volumes and to set attributes and properties (Figure 1(b)).

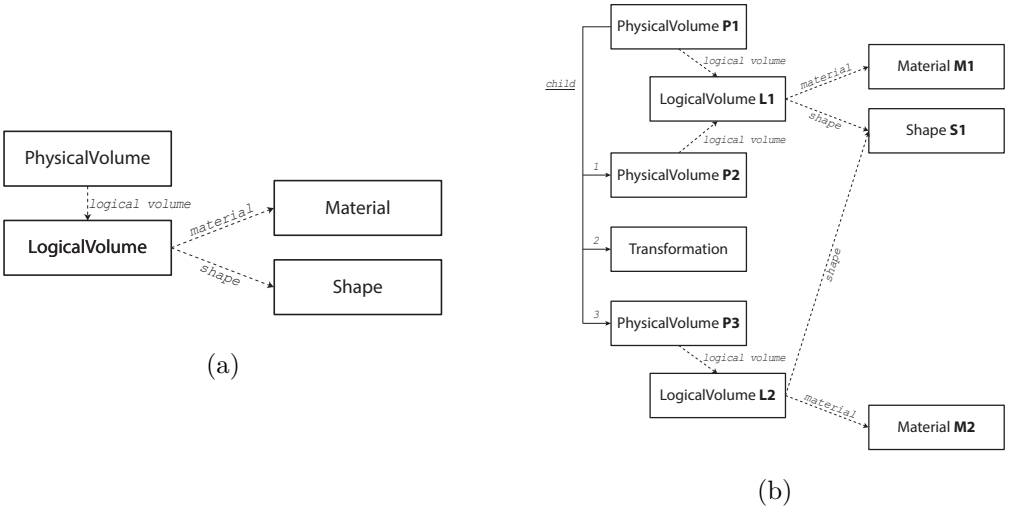

Figure 1: (a) the definition of a logical volume, and its usage in the definition of a physical volume, as described in Ref. [3]; (b) an example of a GeoModel tree, where both shared and nested nodes are used: The Physical Volume P2, the Transformation node, and the Physical Volume P3 are all children of P1; both P1 and P2 share the same Logical Volume L1 (a *shared node*), which means they use the same material M1 and the same shape S1; P3, on the contrary, references the logical volume L2, which uses the shape S1 (another *shared node*, because it is used by L1, too) and the material M2. The position of the nodes within the tree is important: In this example, the Transformation node (child 2) affects only the P3 node (child 3).

The hierarchy of the nodes and their attributes are defined in different C++ files, typically related to specific sub-detectors. In addition to that, parameters which can change over time are stored in a dedicated online database, and retrieved through the experiment's software framework. The resulting GeoModel tree is then built on-the-fly upon request, through the experiment's framework, and stored in memory. At that time the experiment's geometry is ready to be used by applications running inside the framework.

3. The GeoModel structure

The GeoModel library offers many types of nodes, used to build objects, place volumes, and set attributes; as seen above, nodes can be shared within the tree. Moreover, physical volumes

can be nested, through a *parent-child* relationship: The *parent* can act as an outer container, containing several *children volumes*. Figure 1(b) shows an example of a more complex tree, where both shared and nested nodes are used. In GeoModel the order of children and nodes is critical: Nodes like space transforms affect only the nodes after them, like the Transformation node in the example tree; moreover, transformations are accumulated: When traversing the sub-tree, if a transform node is found, its transformation matrix is combined with the transformation matrix of the other transform nodes before that in the sub-tree.

The detector description of a modern HEP experiment uses a very complex hierarchy of sub-trees to define every single piece of the geometry. In ATLAS roughly 80,000 physical volumes are used, built with nearly 50,000 logical volumes. The shape and material nodes used to define the logical volumes are about 100,000 and 380, respectively; and approximately 200,000 transform nodes are used to place all the pieces of the ATLAS detector correctly. The number of the transform nodes is higher than the number of the physical volumes because a same physical volume can be used to define different pieces of the detector (*shared volumes*).

4. Serving the detector description

The resulting ATLAS description is a huge and complex hierarchical structure of roughly 500,000 GeoModel nodes. In addition, another piece of data is part of the detector description: The relationships among all the nodes and the position of each node in each sub-tree. There are around 665,000 of these entries and they are part of a complete description of the ATLAS GeoModel tree.

In the earlier paper, which the work described in this article is built upon [3], the focus has been placed on the design of a new data model to efficiently dump and store this complex data structure independently of the experiment's software framework, and on the development of a new mechanism to make a persistent local copy of it. Using the server-less database SQLite [4] we could store a persistent copy of the full GeoModel structure into a small file[1], which can be easily distributed to end users. The new mechanism allows a straightforward restore of the GeoModel, but it has some limitations: The whole detector description tree is restored, without the possibility of building sub-trees only; there is no way to easily inspect and examine the detector geometry; and using the new SQLite database, asking for nodes based on properties and relationships requires very complicated SQL queries.

To efficiently query and retrieve our complex tree-structured geometrical data, two open-source architectures and their relative platforms have been identified: The graph-based database Neo4j [6, 7], and the Elasticsearch search engine and its graphical interface Kibana [5]. The two platforms have a different purpose, and they have been chosen to address two different use cases: The first as development and debug tool, mainly to examine the actual detector description; the latter for an easy and straightforward use of the detector geometry by third-party applications which are not interested in the description structure itself, but only in the final outcome, that is the *rendered geometry*[2].

5. Graph-based data structure to store detector description data

The graph-based format used by Neo4j is ideal to store the GeoModel data: Neo4j has no concept of data tables, it only works with nodes, relationships, and properties. Therefore, the complex structure of the GeoModel tree is efficiently stored in full detail, without using auxiliary intermediate link tables like in SQL-based data formats. Also, all sorts of properties

[1] With the current implementation, the complete ATLAS geometry fits into a 46 MBytes SQLite file. For comparison, the corresponding in-memory footprint is about 130 MBytes [3].

[2] In this context, the expression *"rendered geometry"* refers to the geometry computed by traversing the GeoModel tree.

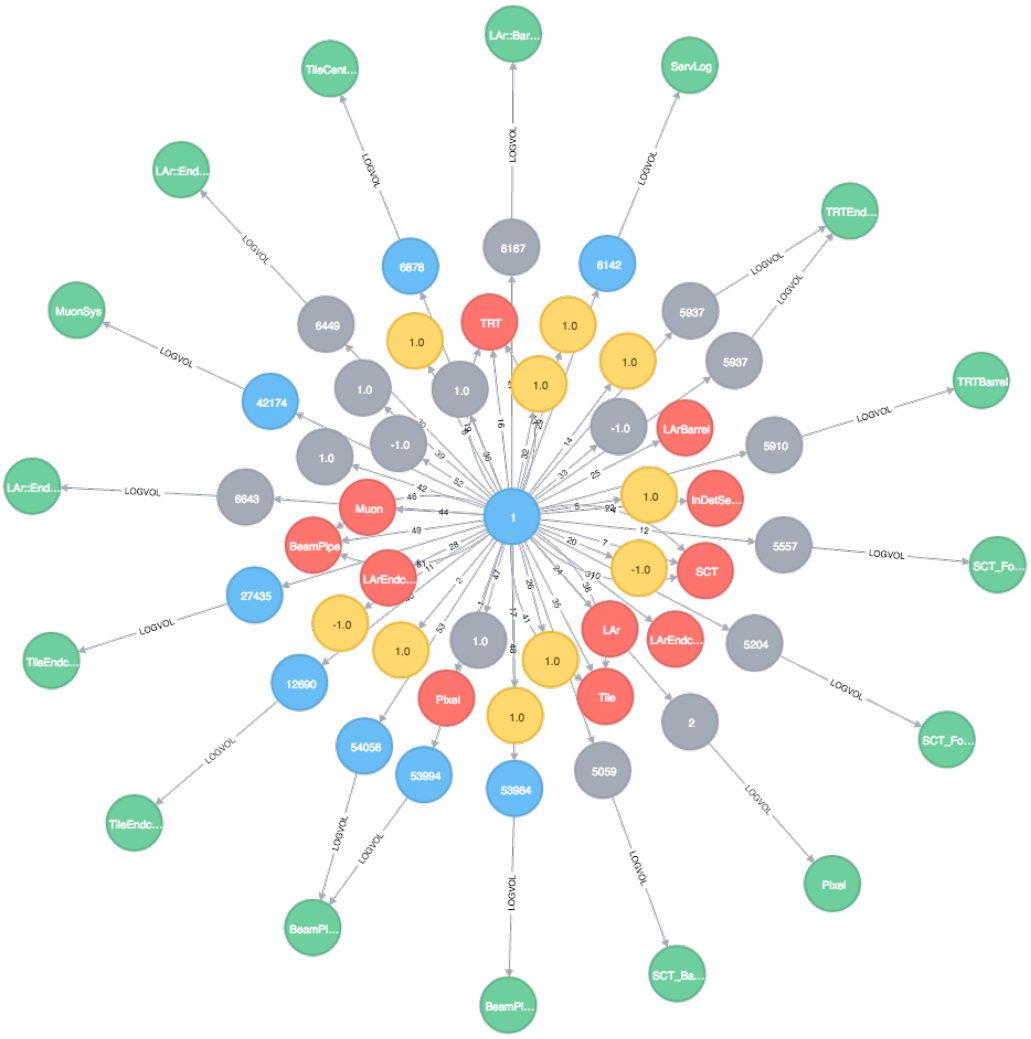

Figure 2: The graphical output of a Neo4j query retrieving the top logical volumes of the ATLAS geometry; Physical volumes are rendered in blue, Logical volumes in green, Transformations in grey and yellow. The node at the center is the World Volume. The numbers on the arrows state the children's position in the subtree. The red Name Tags define the global name of subtrees defining the geometry of the ATLAS sub-detectors: *Pixel* detector, *Muon* spectrometer, *LAr* and *Tile* calorimeters, and so forth. In this graph, only the nodes of the tree satisfying the example query are shown.

can be attached to each node or relationship in an independent way, without having to create additional tables for a single property or entity.

The first step required the export of the GeoModel data stored in SQLite tables (as described in Ref. [3]) into comma-separated files (CSV). Then, the CSV data was imported into Neo4j though a series of Cypher queries that create nodes and relationships. *Cypher* is the native query language of Neo4j, which is focused on graph-based data and optimized to perform queries and operations involving nodes, relationships, links, paths, and properties.

Cypher can be used to query the GeoModel graph as well, once the data are imported in the Neo4j database. Its syntax is very powerful, letting developers accurately inspect the detector description while developing or debugging it. Listing 1 shows an example of actual Cypher queries, used to inspect the structure of the ATLAS detector description. Figure 2 shows an

example of the graphical result of a Neo4j query.

```
/*retrieve a specific logical volume filtering on its name*/
match (n:LogVol) where n.name = "TileEndcapNeg" return n;
/*as above, but retrieve the GeoPhysLogVol which uses it, too*/
match (n:PhysVol)-[r:LOGVOL]->(m:LogVol) where m.name = "TileEndcapNeg" return m,
     labels(m),n,r;
/*get all "Box" Shape nodes used together with "Air" as Material*/
match (s:Shape {type: "Box"})<-[rs:SHAPE]-(l:LogVol)-[rm:MATERIAL]->(m:Material {
    name: "Air"}) return s,rs,l,rm,m;
/*get all \emph{root volume} children and sort them by the 'position' property of
     the 'CHILD' relationship*/
match (n:RootVolume)-[r:CHILD]->(m) with r,m order by r.position return r.
    position, m;
```

Listing 1: Example of Cypher queries which can be used with the Neo4j interface

6. A search engine to serve the rendered geometry

The data returned by Neo4j reflects the raw structure of the GeoModel tree, and that is useful for developing and debugging the detector description. But, as briefly introduced in Section 4, other kinds of applications are interested in the final geometry volumes instead—for instance, all tools that are interested in the visual representation of the ATLAS detector, such as event displays and applications for outreach and education. To be able to provide that kind of data to clients, we chose to use the search engine Elasticsearch. The goal is to let applications quickly retrieve geometry objects without having to traverse the geometry tree.

Data from the SQLite file are converted to JSON, then inserted into Elasticsearch, traversing the tree data and pre-computing the geometry data at index time. Also, geometry data are *flattened*: All the attributes are collected along the way, and all the space transforms are computed and accumulated, in order to get all the attributes and the absolute position of each volume directly associated with the object itself. Ultimately, Elasticsearch does not store the actual GeoModel structure anymore; it stores *flattened objects* instead, which represent final usable objects of the geometry. The loss of information about the GeoModel structure is compensated by an increased speed in getting the final geometry volumes and in a greater convenience in querying them. The data can be interactively inspected using Kibana front-end, or queried and retrieved using Elasticsearch REST API. Listing 2 shows an example of Lucene queries (Lucene is the query language used by Elasticsearch); while Figure 3 shows flattened geometry data retrieved through the Kibana web-based interface.

```
/*retrieve a specific volume*/
name:TileEndcapNeg
/*retrieve all Pixel volumes in the EndCap side C*/
tags:Pixel AND tags:EndcapC
/*retrieve all box-shaped volumes whose material is ""Air*/
shape:box AND material:Air
```

Listing 2: Example of Lucene queries which can be used with the Ealsticsearch/Kibana interface

7. Comparison and conclusions

The main issues with the current implementation with the ATLAS Geometry system are the inability to access the detector geometry outside the experiment's framework, the lack of a way to query and inspect the detector description, and the lack of a tool to easily retrieve the final

Figure 3: *Flattened* geometry data retrieved through the Kibana web-based interface; each line gives the information of a single volume, collected traversing the geometry tree: Name, space position, dimensions, material and so forth.

rendered geometry or usable partial data. The first issue has been solved in the earlier work [3], defining a new model to store the complete geometry data in a SQLite database in an storage-wise efficient way. But the SQL-based data model could not solve the remaining two issues: Easily querying and serving the geometry. Those were addressed in the work presented in this paper.

As said, GeoModel is composed by nodes interconnected by many types of relationships; however, relationships are not natively stored as objects in SQL-based databases; link tables and external refs are used to connect two items in two different tables. Therefore, SQL queries aimed at retrieve nodes and the relationships between them get very complicated very quickly. Here is where a graph-DB comes to rescue, by natively storing data as nodes and relationships. That lets users easily interact with the geometry data, examining and visualizing all nodes and relationships with convenient graph-aware queries. In addition to that, a search engine like Elasticsearch, let us easily store the rendered geometry into a searchable data structure, to easily retrieve specific final volumes ready to be used in external applications.

In conclusion, while GeoModel has been designed to minimise the memory footprint and to ease the definition of complex geometries by using efficient C++ code, the new mechanism using Neo4j and Elasticsearch has been designed and optimised to easily and interactively query the geometry data and to get the final volumes by just using simple text-based online queries.

Table 1 summarizes all the features offered by the new systems, compared to the current implementation. The original *in-framework* mechanism is designed to store all the latest and versioned data describing the ATLAS geometry. The SQLite-based persistification mechanism is designed to create a persistent copy of specific versions of the ATLAS geometry. On the other hand, the newest mechanisms presented here are designed to easily store and query the persistified geometry data. As one can infer from the table, the two new systems are complementary, since they address different targets and use-cases. Neo4j stores and exposes the full GeoModel structure, which can be queried, inspected and visualized. While Elasticsearch/Kibana serves only the final, computed geometrical volumes, ready to be queried and used by client applications interested in the visualization of the final *rendered* geometry.

	Current implementation (*in-framework*)	Persistent SQLite DB	Neo4j	ElasticSearch Kibana
Out-of-framework access to complete geometry data	No	**Yes**	**Yes**	**Yes**
Local copy of complete geometry data	No	**Yes**	**Yes**	No
Interactive query of GeoModel nodes and relationships	No	*Yes*[‡]	**Yes**	No
Visualization of GeoModel nodes and relationships	No	No	**Yes**	No
Partial retrieval of geometry data (subtrees)	No	*Yes*[‡]	**Yes**	**Yes**
Fast retrieval of final physical volumes with global positions	No	No	No	**Yes**

[‡] *with complex SQL queries*

Table 1: Comparison of different tools used in ATLAS to store, query and retrieve the geometry and data description data. The first column shows the current *in-framework* system; the second column refers to the mechanism described and presented in Ref. [3]; the third and the fourth columns refer to the newest mechanism which is the object of this paper.

Together, they offer a complete suite of tools to store, handle, inspect, retrieve detector description data for a large HEP experiment like ATLAS.

References

[1] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST 3 (2008) S08003.

[2] Boudreau J and Tsulaia V 2004 *The GeoModel Toolkit for Detector Description*, CHEP '04, Book of Abstracts, `https://indico.cern.ch/event/0/contributions/1294152/`

[3] Bianchi R M, Boudreau J and Vukotic V, *A new experiment-agnostic mechanism to persistify and serve the detector geometry of ATLAS* (2017), CHEP 2016, `https://cds.cern.ch/record/2243141/`

[4] Hipp D R *et al.*, "*SQLite*" [software], `https://www.sqlite.org`

[5] Elasticsearch BV, "*Kibana*" [software], `https://github.com/elastic/kibana`

[6] Neo Database AB, *The Neo Database—A Technology Introduction* [White paper], (2006), `http://dist.neo4j.org/neo-technology-introduction.pdf`

[7] "*Neo4j*" [software], `https://github.com/neo4j/neo4j`