HSF Conditions Database use cases and requirements

Version 1.0

Authors: Roland Sipos (CERN), Paul Laycock (BNL), Andrea Formica (Saclay), Giacomo Govi (INFN Padova)

1 Introduction

The aim of this paper is to write down the main use cases for Conditions Data in HEP, and derive requirements from those use cases. A working definition of Conditions Data is needed to proceed, and we give a summary of the one defined in the HSF white paper [1]. The paper then defines the major use cases for Conditions Data, the writing and reading workflows for those use cases and their constraints. A set of requirements follows, starting from general requirements and then defining those that arise from consideration of the major use cases and workflows.

1.1 Conditions Data and related concepts

The generic term "Conditions data" is used to cover any information relevant to experiment data-taking that is not the event data itself. Examples include data-taking conditions like e.g. the beam conditions, and environmental conditions (temperature). Such data is recorded in databases, those involving separate systems having dedicated databases e.g. beam conditions may come under the remit of the accelerator group who record many parameters to allow them to understand their machine. Equally, the detector conditions are written with high granularity to dedicated SCADA system databases so that detector experts can understand their devices.

When offline computing experts want to process the event data, a subset of all the conditions data is needed – this is our working definition of "conditions data". As this subset of all possible conditions data is read from a database, and it is conditions data, the corresponding database is our working definition of *the* conditions database, as it is the only one relevant to the offline computing experts. These working definitions of "conditions data" and "conditions database" are used throughout this document. Although this definition is common across the LHC experiments and e.g. Belle II, it is neither universal nor particularly intuitive. The reader has been warned!

Conditions data change over time. Understanding the relevant precision of a particular type of conditions data is very important, e.g. the precision of a high voltage reading needed for offline data processing will usually be less than the precision written out by a SCADA system. This allows data volume for conditions data to be reduced, although care is needed to ensure that any variations relevant for offline data processing are retained. Again using high voltage as an example, a simple averaging algorithm would integrate over a pre-defined period of time, but this could easily hide significant problems if the period of time is too large. A better algorithm could be to set a tolerance to define a significant change, and use arbitrary periods of time for averaging. The average value produced is only valid for that

period and is referred to as the *Interval of Validity (IOV)*. The length of the IOV depends on the conditions data and thus is not constrained to be uniform across conditions data types. Conditions data can be more complicated than a single high voltage value. Indeed, in addition to the conditions data that are collected at the time of data taking, there are also conditions data that need to be derived from other sources or even inferred from the event data themselves, e.g. detector calibrations. The name *Payload* is used to refer to the conditions data that is ingested by a data processing framework, whether that be a single floating-point number or a file. Finally, a *Tag* is used to define the chronological variation over time (*IOV*) of a particular type of *Payload*. These concepts are summarized in Figure 1.

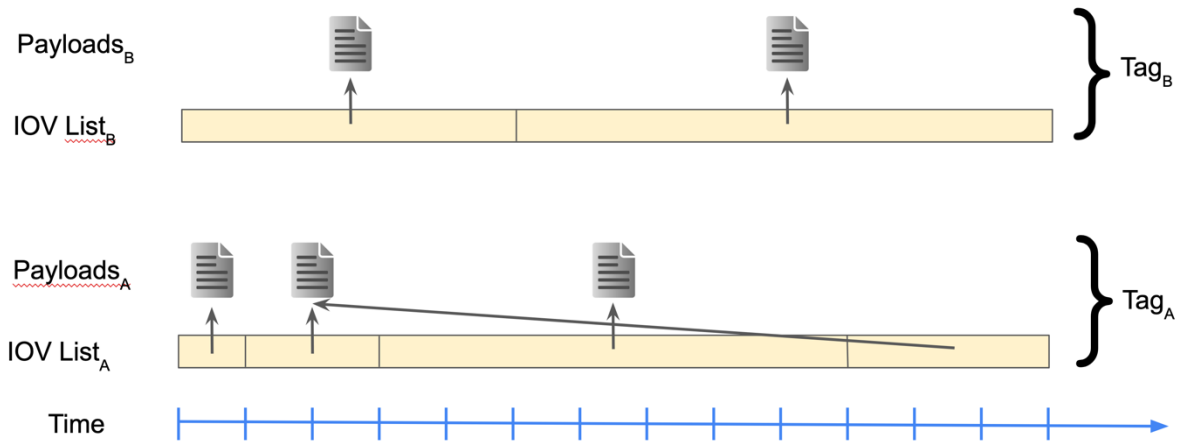

Figure 1: Schematic showing how Conditions Data vary over time for two types of *Conditions Data*, **A** and **B**. A list of *IOVs* conveniently defines the chronological structure, with each *IOV* pointing to a *Payload* file for its particular *Conditions Data* type. The combination of the list of *IOVs* and *Payloads* defines the *Tag*. Note that multiple *IOVs* can point to the same *Payload* file (of the correct type).

2 Conditions use cases

We consider three main use cases, two of which (2.1 and 2.2) are synchronized with respect to data taking, and a third (2.3) which is asynchronous. A fourth use case is a special case of the third use case (2.4, analysis). A final, fifth use case is needed to support software development and conditions data producers (2.5).

2.1 Online / real-time processing use cases

The distinguishing feature of this group of use cases is that generally (apart from for debugging purposes) the data stream is processed once in real or close-to-real time. The latency between data being recorded and processed is so small that there is no chance to have updated conditions prepared in time to process the data. High-level trigger is the most obvious use case, and online data quality is a second example.

Interestingly, the biggest challenge here is to integrate the exceptions to the assumption that no updates will happen. Examples include measurements of the beam position which is fundamental to calculating the physics quantities used to select events. [describe how this is done for ATLAS and CMS].

Aside from the exceptional conditions, the general workflow for writing is simple. Updates are made rarely as a stable benchmark is more important than having the best conditions, and conditions calculated for old data can only be used to process future data. These updates are usually carefully validated offline before carefully deployed as a well synchronized update, which are assumed to be valid until further notice. The reading use case simply uses the latest suitable set of conditions available, but crucially there must always be valid conditions data available for this use case. Figure 2 shows the online update use case, with Run Control being used to determine whether the conditions update is safe (in the future) or not.

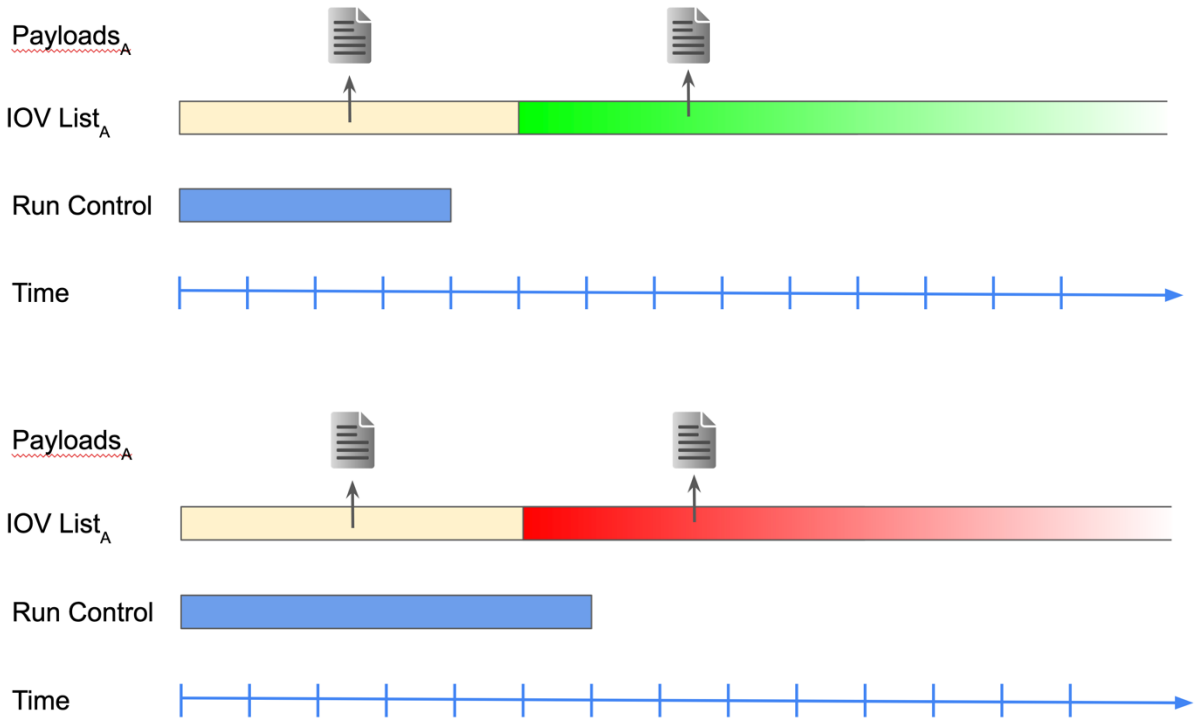

Figure 2: Schematic showing an online update use case for one type of *Conditions Data,* **A**. In the first scenario (top), the update has a start time in the future wrt Run Control and is accepted – there is no end of validity. In the second scenario (bottom), the update is attempted for data that has already been taken and therefore the update is rejected.

2.2 Fast-processing use cases

The physics performance that can be extracted from processed data tends to improve with both increasing data volume (statistics), and time (needed to understand and correct for subtle detector effects). Probably the most challenging conditions data use case is the fast-

processing workflow, which attempts to balance the competing demands of making processed data available as fast as possible for analysis, with the physics performance of that processed data. Several strategies are used depending on the physics performance requirements of the specific use case. One strategy can be to process a fraction of data very quickly with existing conditions data to derive improved conditions (detector calibrations and alignment). This can be used to give better data quality information than possible in the control room, and those improved conditions can be used to process all data.

In this set of use cases, the updated conditions are generally calculated using the same data, or a subset thereof, that they will then be used to process. **Orchestration** is needed to ensure that new conditions have been calculated before the data is processed in bulk. Given that orchestration is needed, there is no requirement that there should be valid conditions data available for all IOVs. In the most challenging case, conditions for data periods can be updated out of chronological order and efficient orchestration is needed to ensure data processing resources are completely saturated to keep up with data taking.

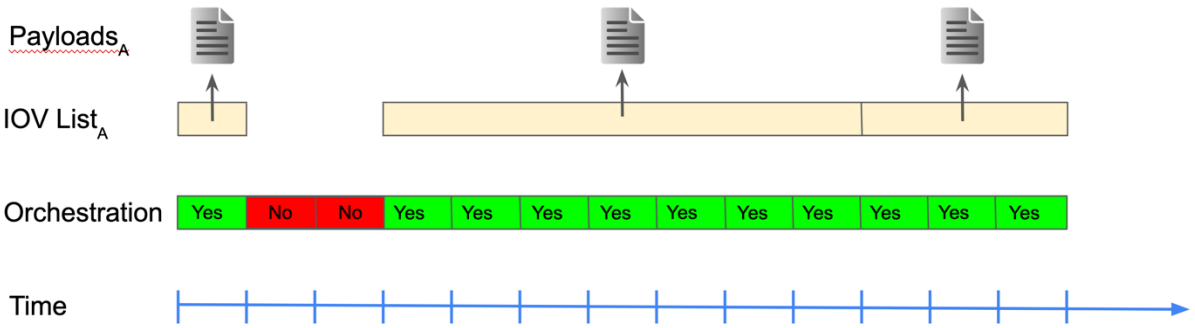

Figure 3: Schematic showing out of order *Conditions Data* updates for the prompt processing use case. *IOVs* do not overlap and **Orchestration** ensures that only data that have valid *Conditions Data* are processed. The *IOVs* with missing conditions data can be processed once those IOVs have been filled.

2.3 Reprocessing use cases

A final set of use cases is by far the simplest to handle as there is no synchronization required between data taking and the conditions being consumed. The chronological structure and content of the conditions are fully defined by experts, with no constraints on the writing use case.

A caveat here is that the "best guess" for future data taking is usually the last best set of conditions data. Equally, avoiding artificial boundaries between the IOVs for reprocessed data and for future data allows knowledge to be consolidated. For example, the "best knowledge" gained in Run 1 can be captured in the reprocessing IOVs that cover that period, while the "best knowledge" for Run 2 at the time will be captured in the "fast-processing"

IOVs for that period (later to be superseded by Run 2 reprocessing IOVs. This is particularly important when considering Data Preservation.

2.4 Analysis use cases

A special case of the Reprocessing use case is analysis. Here the main issue is the variety of analysis use cases. Data processed using a fast-processing or reprocessing workflow is computing resource intensive and is usually performed centrally with one set of conditions data common to the whole experiment. Generally, the output is a dataset that can be analyzed with much lower computing resource consumption. It is typically analyzed multiple times and analysis-specific conditions data produced and consumed at the analysis stage are often managed by individual physicists. An HEP analysis is very often a collaborative effort, and although there are some analysis conditions that are unique to one analysis, there are also a lot of analysis conditions that are common across many analyses. The arguments to make these analysis conditions first class citizens of the conditions data ecosystem include reproducibility, collaboration, and data and analysis preservation. As the timescale of modern HEP experimental analyses can now stretch beyond the term of a post-doctoral contract, the risk of work being effectively lost increases.

Management of analysis conditions is primarily a logistical problem rather than a technical one, although there are technical challenges. For each centrally produced dataset, made with one set of conditions data, there are multiple sets of analysis conditions needed to capture the full picture of any given analysis with different degrees of commonality. Thus, for an individual analysis, the problem of collecting all the conditions data needed for that workflow is exactly the same as for the reprocessing use case. It would clearly be desirable to capture the common elements and reduce entropy across the whole physics program, but as there is no agreed upon solution to this aspect, we leave this as a problem for the interested reader.

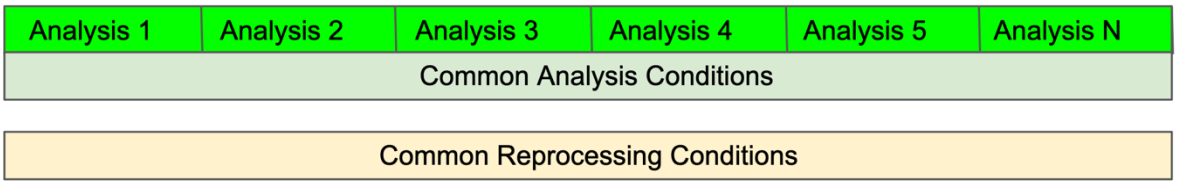

Figure 4: Conditions Data for the analysis use case shown as a hierarchy of conditions. For example, Analysis 1 requires all three of “Common Reprocessing Conditions” (needed to produce the baseline data), the “Common Analysis Conditions” (common to all analyses for that experiment using that dataset) and “Analysis 1” (conditions data specific to that analysis). In practice there may be several more groups and/or hierarchies.

2.5 Development use cases

A final use case is critical for conditions data producers who need to validate new conditions data payloads, and software developers creating new types of conditions data. Here the main challenge is to validate that the newly produced conditions data behaves as expected.

The producer needs a mechanism by which they can override a subset of data processing conditions and then compare the results against a control sample. In practice the subset of conditions data usually consists of only one conditions data type, though there are cases where conditions data types are highly correlated and require coherent sets to be validated at once. Similarly, the developer needs a mechanism by which they can add a new conditions data type to the conditions used for data processing, such that they can compare the results with the corresponding control sample. Ideally in both cases the same software version should be used to produce the control sample, and the size of the control sample will depend on the conditions data being tested.

3 General Requirements

We start with the fundamental requirements which are independent of the individual use cases and arise from the context of data processing for big science.

3.1 Versioning – ARE IOVS AND TAGS ACTUALLY REQUIREMENTS THAT SHOULD BE DEFINED ALREADY IN SECTION 1 WHEN THEY FIRST APPEAR? BREAKS UP THE REQUIREMENTS BUT MAYBE BETTER LIKE THAT.

Already while introducing general conditions data terminology, the fact that HEP data are generally analyzed as large datasets that have time-varying conditions required us to track those changes in time using *IOVs*. Similarly, versioning of each type of conditions data across multiple *IOVs* is required and we use the term *Tag* to indicate this. One *Tag* should be valid for many *IOVs*, i.e. a large dataset.

3.2 Coherence – THIS AND 3.3 COULD ALSO BE MOVED TO SECTION 1 AS GENERAL REQUIREMENTS

As coherent data processing results require several different types of conditions data to be used in parallel by data processing frameworks, versioning across different types of conditions data and time is required. This global versioning is referred to as a *GlobalTag* which again should be valid for many *IOVs*, i.e. a large dataset.

3.3 Reproducibility

As conditions data need to be updated, and yet we require reproducibility, some minimal finite state behavior for a *GlobalTag* is needed which we define to as the *GlobalTagState*. The simplest set of states could comprise e.g. *LOCKED* and *UNLOCKED*, with only the *LOCKED* state guaranteeing coherent results.

Running the same data processing software on the same computing hardware over the same input data with the same version of conditions data configuration should produce the same output. Data processing should therefore in general be configured to use a *LOCKED GlobalTag*.

4 Use case requirements

From the data processing use cases considered, several requirements follow.

4.1 Open-ended *IOVs* – DO WE NEED TO RECORD THE START TIMES FOR THE ONLINE DATA PROCESSING USE CASE? I WOULD ARGUE YES, AND THEREFORE IT'S CORRECT TO REFER TO THIS AS AN IOV WITH A START TIME AND NO END TIME. A SYSTEM THAT ONLY ALLOWED UPDATES WITHOUT RECORDING WHEN THEY HAPPENED WOULD MAKE IT IMPOSSIBLE TO DEBUG PROBLEMS AFTER THE FACT.

It must be possible to define an "infinite" or *Open-ended IOV* such that for the ***Online data processing*** use case there is always valid conditions data. Once an update is made it is valid until further notice (i.e. there is no end validity time). This is shown in Figure 2.

4.3 Atomic updates of *LOCKED GlobalTags* – THIS WAS THE MOST GENERAL PHRASING I COULD THINK OF THAT DOESN'T TIE ITSELF TO AN IMPLEMENTATION, IS IT CORRECT? DOES IT NEED MORE CLARIFICATION?

GlobalTags should be in a *LOCKED* state for data processing, and yet data processing workflows require conditions data updates. Atomic transactions that allow conditional updates to support both the ***Online*** and ***fast-processing*** workflows are required.

4.4 Closed *IOVs* – THIS IS DIFFICULT TO UNDERSTAND WITHOUT REFERENCE TO FIGURE 3, BUT THIS IS A REAL USE CASE FOR SPHENIX !

To satisfy the worst case scenario of the ***fast-processing*** use case, atomic *LOCKED GlobalTag* transactions must allow out-of-chronological-order updates. To simultaneously guarantee reproducibility for any existing *IOVs*, that implies that the end of validity must be specified when writing ***fast-processing*** conditions data, i.e. *Closed IOVs* must be supported.