Conditions Database Working Group

Introduction

As the definition of “conditions data” itself has been problematic, the scope of a conditions database in HEP experiments is first defined, together with the principle workflows that need to be supported. The key principles of a well-designed conditions database system are given, including the details of a data model for conditions data management, a key point of convergence of this working group. Finally some of the experience of payload technology from the different experiments is documented.

1 - Conditions in HEP experiments

1.1 - Definition of conditions data

Broadly speaking, conditions data is defined as the non-event data required by data-processing software to correctly simulate, digitise or reconstruct the raw detector event data. The non-event data required to maintain, operate and optimise detectors can be broken down into the following categories:

a. Configuration parameters
b. Detector Control System (DCS) [copied from Process Visualization and Control System (PVSS)]
c. Detector and system monitoring information, higher level monitoring than (b) and often used as input to (d)
d. Detector calibration data, e.g. alignment values

Conditions data thus largely consists of (d), together with the subset of (a) and (b) that are required for data processing. Other non-event data may also be required for data processing, for example machine parameters, and thus this list only attempts to give an indication of the scope. In practice, any non-event data from any source that are required for optimal data processing can be considered as conditions data. In general, conditions data vary with time but with a granularity much coarser than the event, ranging from one year to one run, down to a granularity of the order of one minute for a small subset.

1.2 - Workflows

Given the definition of conditions data, the workflows that need to be supported are predominantly those of offline data processing. However, both CMS and ATLAS high level (software) triggers use the same software framework as offline data reconstruction and must also be supported, as must any workflow that requires conditions data.

A non-exhaustive list of important workflows follows:

1) Subsystem calibration: conditions determination and testing (including the ability to access a copy of the conditions stored locally); uploading conditions to the production database.
2) Prompt data processing, with conditions updates adhering to strict protocols:
    a) the software trigger: running on dedicated resources with a dedicated server.
    b) similarly for prompt offline data processing, including the calibration determination.
3) Offline data processing, including Monte Carlo simulation, using pre-determined conditions.

a) Importantly this generally involves distributed data processing on the grid where caching is required to be able to run at scale.
b) HPC and similar off-grid resources are also becoming viable data processing locations and will have their own special requirements

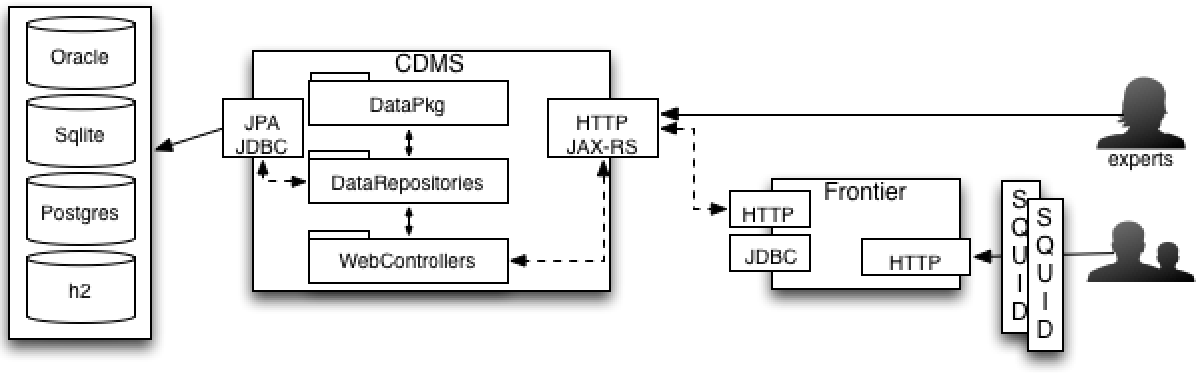
1.3 Data volumes, read and write rates

Conditions data tend to scale in a controlled way during the lifetime of an experiment, typically producing data volumes of Gigabyte to Terabyte scale. It is worth noting that on ATLAS, where the same database instance was used for conditions data as well as the DCS and trigger data, the DCS and trigger data dominated the offline and online database instances, respectively. It is therefore strongly recommended by this working group to factorise conditions data from other use cases, even where these are related.

Typically write-rates for conditions data must support of the order of 1 Hz (to ensure good support for several independent systems writing conditions data every minute), with the majority of conditions data being updated much less frequently than this. On the other hand, read-rates up to several kHz must be supported for distributed computing workflows, where thousands of jobs needing the same conditions data may start up at the same time. Conditions data are typically written once and read frequently.

<u>2 - Conditions Database Archetype</u>

The archetypal solution to a conditions database management system (CDMS) is shown below. The conditions data payloads are stored in a master database and are accessed using a client-server design through a REST interface. All of the experiments gave feedback that achieving a high degree of separation between client and server was very desirable. Due to the read-rate requirements, caching is extremely important and good experience was seen when using web-proxy caches, e.g. the Squid cache shown here. Some key design principles are detailed in the following.



2.1 - Payload technology

Experiments will inevitably choose their favourite payload technology. **This working group recommends placing most emphasis on homogeneity and long-term maintenance when making this choice.** Inhomogeneity and home-grown solutions all place additional burden on projects that typically lack the resources to support this after the initial build and commissioning phase of an experiment. CMS has very good experience of removing choice and only supporting boost-serialised C++ objects, with all classes belonging to one package in the CMSSW framework. Such a strategy lends itself more readily to long-term maintenance and minimises hurdles to data preservation. It is noted that there are payload formats used more in industry which would lend themselves more to higher-level functionality without the need of the software framework, but, while this is attractive, the choice of format tends to be driven by software framework developers.

2.2 - Database backend

One of the key features of the design is that it is agnostic to particular choices of database backend. This was also one of the key features of COOL, but due to the lack of caching in COOL, the queries themselves had to become more complicated. Thus on ATLAS, which uses an Oracle back-end, a significant amount of Oracle DBA effort was required to tune the queries and make them performant. It is therefore important to realise that real flexibility with respect to choices in database backends only comes when the system as a whole is simplified.

2.3 - Client-side requirements

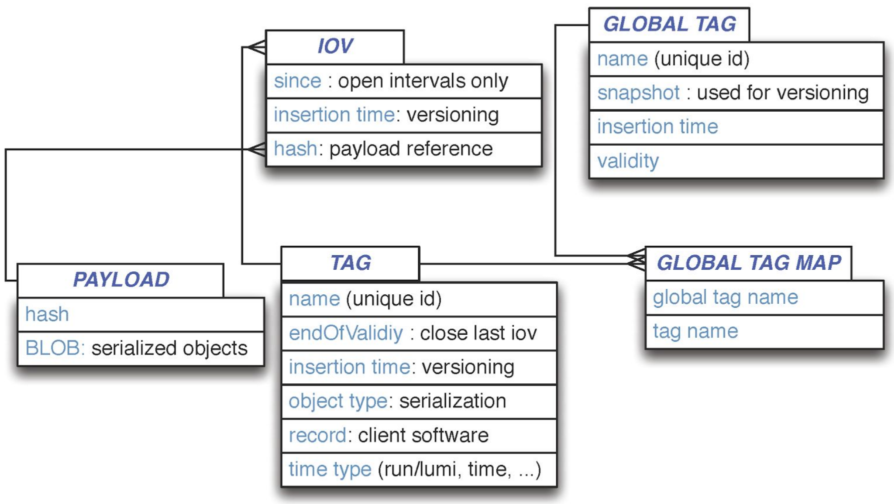
The client layer should be as simple as possible and should be as agnostic of the rest of the architecture as it is possible to be in order to improve maintainability. The database insertion tools in particular benefit from adopting a simple e.g. REST interface. The client layer needs to take care of payload deserialisation, as the remaining architectural components will deal with serialised objects. Experience also shows that clients should be able to manage multiple proxies and servers to provide robustness against server failures.

2.4 - Caching layer

CMS and ATLAS require an intermediate layer between the client and server to provide caching capabilities. Considering the distributed use case, where thousands of jobs start at the same time and require the same conditions data, this is a clear requirement and one that can be well met using web-proxy technologies. An alternative solution would be to use a distributed filesystem with good caching capabilities, and the ALICE experiment is gaining experience using cvmfs. The simplicity of this solution makes it very attractive and it has thus been adopted as a strategy by the NA62 experiment. This in turn suggests that, as a design requirement, it should be possible to represent a conditions database on a filesystem. The primary challenge is to make the filesystem mapping use the cvmfs caching layers efficiently. Several experiments also have experience using SQLite replicas which are attractive for workflows where the exact subset of conditions is known in advance (some MC workflows), but a performant caching layer is preferable for general use cases.

2.5 - Data Model

The data model for conditions data management is an area where the experiments have converged on something like a best practice. The model is shown in the figure below. A global tag is the top-level configuration of all conditions data. For a given system and a given interval of validity, a global tag will resolve to one, and only one, conditions data payload. The **Global Tag** resolves to a particular system **Tag** via the **Global Tag Map** table. A system **Tag** consists of many intervals of validity or entries in the **IOV** table. Finally, each entry in the **IOV** table maps to a payload via its unique hash key in the **Payload** table. A relational database is a good choice for this design.


IOV
since : open intervals only
insertion time: versioning
hash: payload reference
GLOBAL TAG
name (unique id)
snapshot : used for versioning
insertion time
validity
PAYLOAD
hash
BLOB: serialized objects
TAG
name (unique id)
endOfValidiy : close last iov
insertion time: versioning
object type: serialization
record: client software
time type (run/lumi, time, ...)
GLOBAL TAG MAP
global tag name
tag name

This design has several key features. Firstly, conditions data payloads are uniquely identified by a hash which is the sole reference to any given conditions data payload. The payload data has been separated from the data management metadata and could in principle be placed in a separate storage system. This could also be important for data preservation, as the entire metadata component will occupy a trivial data volume and could exist in e.g. an SQLite file, while the payload storage could be handled separately. Secondly, IOVs are resolved independently of payloads and are also cacheable. **Efficient caching is a key design requirement for any conditions database that must support high rate data access.**

2.6 - A git-based approach

For workflows which are completely offline and asynchronous with respect to data-taking, LHCb has adopted a different approach. Using git as the versioning system, conditions are placed in a directory structure, one for each type of condition. A file is used to map timestamps to payload files, and a simple format is used to allow a level of indirection to improve performance. With the algorithm used, payload look-up on timestamp is linear within the file, so large numbers of IOVs cause performance issues; the file format allows a timestamp to point either directly to a payload file or to a directory, thus allowing partitioning of the lookup. Versioning is then taken care of by creating a git tag, equivalent to a global tag.

3 - Future and roadmap

The Conditions Data Management System described here is expected to meet the needs of CMS and ATLAS into the HL-LHC era [1]. Data volumes are not expected to exceed a Terabyte per year, and the rate of requests (determined by the computing resources of the experiments) are expected to peak at tens of kHz. In other words, assuming the solutions presented here work in Run 3, they are also expected to work beyond that. The most important issue for those experiments then will be maintenance and operation in the face of evolving hardware and infrastructure, which makes consolidation to a simple and modular design, such as that presented here, crucial for the experiments' continued success. Full scale tests of this design are expected in the coming years, but based on experience with similar systems (the current CMS approach) the outlook is positive.

Nevertheless, there are still open questions that need to be addressed. The functionality to produce a filesystem-based replica of a conditions database, that can be accessed transparently to the client, needs to be prototyped. The NA62 experiment will look at this problem in reverse, moving from a filesystem-based approach to a conditions database. This requires first modifying the filesystem-based infrastructure so that it can later be migrated to a database.

Meanwhile, LHCb and ALICE will move to only applying calibrations promptly on a time-scale of Run 3, effectively making them conditions-free for offline workflows. This is completely driven by the data rates, the data-volume output to offline workflows must be significantly reduced and that can only be achieved if the only data output is already fully reconstructed. While this clearly leaves those experiments exposed to potential data quality losses, the benefit in terms of data processing is equally obvious.

3.1 - Conditions for analysis

Despite the plans for LHCb and ALICE to be effectively conditions-free, there will nevertheless be higher-level calibrations applied to the physics data, and this will require some management. Belle II expects that this will be part of the analysis model from the outset, and there is wider interest in the analysis community in using systems like those described here to deliver "analysis conditions data". Given the plans of Belle II, and the growing interest in the analysis community, feedback on the suitability of these solutions is expected within the next few years.

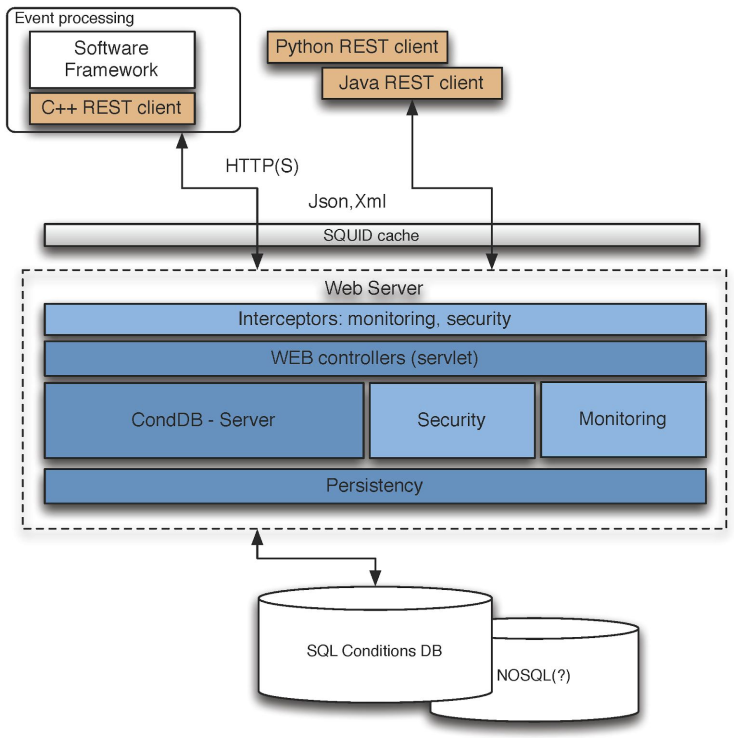
4 - Conclusion

Following a decade of experience with the LHC experiments, conditions data management is an important component of HEP software and one that is often considered relatively late in the software design cycle. While the data volumes are easily accommodated by several database technologies, the workflows can be demanding. In particular, access rates at the level of tens of kHz is critical (and non-trivial) to support, and the importance of a caching system like Frontier [2] or CVMFS [3] for CMS and ATLAS cannot be overstated.

Several experiments have converged on similar solutions for their plans for solving conditions data management problems. All of the experiments want a high degree of separation between the client and server side of the problem, with a client that is relatively simple (in contrast to the original COOL solution). An intermediate layer with caching capability is needed to support high rate requests and web-proxies have performed very well. REST interfaces and industry standard components, together with a relational data model to represent the global tag concept, can support the wide variety of workflows used in HEP while being sufficiently modular to evolve easily. For purely offline workflows, cvmfs is attractive and could be produced from the master database. This would resemble the git-based approach used by LHCb, where the IOV structure is encapsulated by a simple file. Analysis conditions data management could also benefit from using one of these solutions.

[1] DB Futures Workshop: https://indico.cern.ch/event/615499/timetable/?view=standard
[2] Frontier homepage: http://frontier.cern.ch/
[3] CVMFS homepage: https://cernvm.cern.ch/portal/filesystem

Backup figures



The live notes from the workshop follow, this was the basis for the writeup

Conditions DB WG CWP live draft

(draft, starting point is the ATLAS answers so beware bias!)
Present: Paul Laycock, Andrea Formica, Dave Dykstra, Giacomo Govi, Marko Bracko, Lynn Wood

Key: Important answers/points are underlined
Context - to set the scope of the document
Recommendation - something we would like to state as a recommendation in the CWP
Observation - considered to be useful, this will appear in the CWP
***Require feedback** - feedback required from the analysis framework WG*

1) Define what we want to consider as conditions data (based on past experience).

Context - In general we have:

a) Configuration
b) DCS (copied from PVSS)
c) Monitoring
d) Calibrations, alignments

When we talk about "conditions data", we think we want to constrain this to largely mean (d), with the subset of (a) and (b) that are needed for offline data processing. For ATLAS, the HLT also uses the same software as offline and is considered to be the same use case in this context.

Consider carefully how to word this - there are things like machine parameters that aren't in this list but that fit very well into the model. There are probably other examples.
Another aspect is that conditions (usually) vary with time.
This has been a hot topic on ATLAS! The common (mis)understanding was "anything that goes into the database that we use to store conditions (COOL)". This led to a lot of problems in defining scope, etc.

2) Study of workflows and access patterns based on LHC experiment past experience. This is important for the data model which determines how the conditions data are managed.

Context - As for (1) we mainly consider offline data processing requirements. A non-exhaustive list of important workflows:

a) Subsystem expert calibration determination, conditions testing (includes ability to access local copy of conditions stored in e.g. sqlite), uploading to production DB

b) HLT workflow, dedicated resources/server, dynamic* conditions with strict requirements for updates - similarly for prompt data processing/ calibration determination

c) Offline data processing, including MC, using (usually) static conditions including distributed data processing on the grid where caching is required to run at scale

d) Offline data processing on HPC and similar resources

Need feedback on write-rate frequency supported (give examples) and read-rate support (give numbers again).

*dynamic wrt time as data is being processed while new data is coming in and calibrations are being determined.
For (b) would be nice to use simpler terminology, but need to avoid “online” as that implies TDAQ, which we’re not talking about.

3) Evaluation of data volumes: we can foresee that the volume of conditions data involved in a large HEP experiment can reach several Terabytes over a period of data taking (conditions data do not scale in the same manner as physics event data with the luminosity, they tend to scale with time).

Observation - The main scaling issue we saw was in non-conditions data (DCS for offline, trigger for online) - it is necessary to take care and only use what’s required for offline processing.

4) Distributed computing environment: conditions data are used by many jobs processing physics event data. Standard client-server (job being the client and a relational DB being the server) architectures have shown their limitations and forced LHC experiments to adopt well conceived caching solutions.

Recommendation: It should be possible to cache the payload, this should be a design requirement and we can give example(s) in the CWP. A unique key to access the payload (instead of a huge number of ways to access the same payload) is critical for this.

5) Related to the last point, we have also to carefully study here the impact on the architecture coming from the usage of HPC centers (without external connectivity) and in particular event-based workflows, ensuring that conditions access is not a bottleneck.

Recommendation: For the distributed environment, it makes a lot of sense to think about cvmfs, the ability to extract a copy to cvmfs (complete with metadata) from a database master is very attractive.

The ability to create a snapshot of conditions for a particular version (global tag) - adding a requirement on the time interval reduces data volume.

6) End-user analysis also uses conditions data, should this data also be covered by a conditions architecture or should much simpler designs be preferred? End-users have a clear preference for understanding all data used in their analysis jobs (c.f. text files containing calibration data), how should conditions data storage/access be simplified to satisfy the end-user community?

***Require feedback** - interaction with analysis framework experts*

My hypothesis is that users don't care where their data is, they just want to be able to access it without needing any special software. Simple python/C++ tools that fetch data from any source and then cache the result locally would satisfy ATLAS requirements. The tools should have a similar look and feel to the home-grown approach which often tends to look like a tool reading a local textfile. It would be good to get the analysis framework perspective on this point, especially as to whether it could be useful to make recommendations from our side, but I think the client-side recommendations are in the analysis framework domain. For storage, analysis users like to be able to use "less" as an "interface", do we recommend cvmfs for remote storage (local caches satisfy that requirement but you shouldn't have to use an entire analysis framework to look at the data?)?

7) Back end solutions: relational databases, NoSQL databases, file-system
   The choice depends both on data volumes and data model, as well as on the kind of informations we want to extract from the stored conditions data.

Recommendation = it should be possible to store the payloads on a file-system (see Q5) - not necessarily the master

Observation = Relational DB appears to be a good solution for the metadata part

The high-level tools for managing conditions data should be backend-agnostic, allowing an implementation of the data model in whichever technology is most appropriate or favoured by the experiment for whatever reasons.
File system based storage: even though we have different solutions adopted, we see a general interest in having some sort of file system based storage for conditions. This is obviously
feasible only if at the end the conditions data (payload data) are represented by sort of "files".
Here I list some solutions I am aware about:
- LHCb : XML files for payload, investigation on git based storage
- Belle II: Root files
- CMS: Boost blobs (recent proposal by Dave to store in CVMFS)
- user analysis: are frequently poorly formatted text files ?
- NA62 : text files for the moment

Pros: it seems to me that storing in some easy file-like representation allows flexibility for future changes and also for "data preservation" issues. HPC use case (no network connectivity) can benefit a lot from this solution as well.

Contra: cannot query on payload values easily, …

Backend solutions part II: essentially relational DBs, as far as metadata are concerned. In some cases we try to use standard technologies to access relational backend, in order to focus our
developments on functionalities, and not on low layers (usage of JDBC/ODBC is preferable for long term maintenance). Interest probably on other technologies (non sql) but still very preliminary.
The backend choice should not block investigations on file-system storage; the possibility for example to "dump" a tag from a DB to the file system seems to me very interesting.

8) Data model: study the options and the effects on performance, maintainability, etc. How far can we / should we standardise choices and what are the pros and cons.

Recommendation - One area of convergence for the WG is in the data model, we would like to draft recommendations (see other questions) on

a) Cacheable payloads (question 4)
b) Cacheable metadata as well

We should include a diagram of the CMS data model in the CWP as a reference
(Need to see how much space this really takes in the document)

Metadata definition: for different reasons it seems that we have quite a lot of convergence in the definition of the metadata (tags, global tags, iovs, …), even though we do not have exactly the same. Simplification of metadata can have a good impact on file-system based storage. We may define similar ways to map these metadata on a file system.

9) Data access layer: the application layer devoted to data access, which in a first approximation should be capable of supporting multiple backend solutions. Several standards are available with different degrees of maturity. Difficult to estimate where we will go in 5/10 years from now.

Recommendation = one approach is to achieve client / (master)backend-storage separation via an intermediate server

Alternative is to use distributed filesystem (cvmfs) for this (text needs some work!)

Can we give a brief poll of the choices available, any pros and cons, any clues as to the future?

Client / Server separation : in several experiments we profit a lot of separation between the client layer (at the level of a single job) capable of requesting the data and the server layer which is devoted to retrieve data from a backend system. One of the biggest impact is caching capabilities here. I think we should promote as much as possible usage of standard technologies to interact with a backend system, and simplify as much as possible the client layer to something “as stupid as possible”, promoting at the same time multi language capabilities.

- CMS/ATLAS : Frontier + new prototype for Run3 (REST)
- LHCb : git
- Belle II : REST on Payara+Hazelcast (micro-services)

10) Caching layer: caching is a key element for the present conditions data access. Again caching solutions are available in the open source industry and should be evaluated. There is somehow a very tight link between the data model, data access layer and caching. This should be studied as well.

Recommendation: Very good experience using web-proxy caches locally at grid sites. Experience has also shown that it is important to have a client that can manage multiple proxies and servers, to be able to robustly continue operating in the presence of failures.

11) Client layer (interface to production framework): an effort to keep client disentangled from all previous architectural elements is important to improve maintainability.

Recommendation: The client layer needs to take care of the deserialisation of payloads

Would it be useful for us to contribute some recommendations to the software framework WG?

12) Client layer (payload management / upload): emphasis again on disentangling from other architectural elements, but also simplicity here is key - we should not require super experts to perform basic conditions DB operations.

(We can talk here about the workflows that we support and the set of DB transactions required to be supported by the client-side tools, covered in other questions)

13) Configuration layer (c.f. Global tags) of clients vis a vis the framework should be studied, e.g. the management view (global tags) may not be the same as a job-view where emphasis should be on efficiency.

(Nothing to add to the document here, the new data model is better !)

Can compare the COOL global tag implementation with the proposed implementation. Configuration using COOL, which had independent schema for each subsystem, required a large number of queries (across all of those schema) to resolve a global tag. In the new design, this is done by querying the global tagmap table. Hard to see how this wouldn't be efficient?

14) Multi-language support (access to conditions may occur from different clients and not only from production jobs).

Recommendation - the conditions data management system should be language agnostic, acknowledging that experiments will make their own choices about payload serialisation

How high a priority should this be ? Language neutrality sounds good but in the end this is not what ATLAS and CMS experiments have done, enforcing a choice dictated by framework (CMS, serialised-C++ objects) makes sense and if you're accessing experiments conditions you should have access to the experiment s/w. Agreed that it should be possible to read a conditions object if I have its library, I shouldn't need to run the whole reconstruction program.

15) Data model and data preservation: study the evolution of payload technology choices particularly wrt backend storage evolution - how robust can we expect to be? Are some choices better than others in this regard, and can we standardise payload format (c.f. CMS example), what are the pros and cons?

Recommendation - choice of payload technology should take into account data preservation issues (long-term support, etc.)

Similarly to q14, language-neutral is in principle good so long as you make a good (lucky?) choice. Otherwise tying the format to the s/w framework at least means that you evolve both.

16) Payload formats (was in Q8) - and format evolution (maintenance matters more than pure I/O, formats should be easy to evolve because they will evolve - should still study potential for this to be a bottleneck consideration).

Recommendation - maintenance has the highest priority in making a choice of payload technology, consider homogeneity (over heterogenous solutions) more important than ultimate performance.

Serialization solutions (organise your conditions): we have different serialisation but in almost all experiments the idea is to have data which I can "dump" on a file.

- XML, Boost, Root, …

Should we try out something together here ?
Language independence for me is essential, I may want to write conditions without installing complicated libraries, and may be even being capable to "look with my own eyes" to the data.
Obviously every choice has a cost… Any thoughts ?