Conditions Database for Run3

Status of present infrastructure and proposed architecture

A.Formica, E.Gallas, D.Barberis

Topics

- Present infrastructure: COOL
  - what have we learned during Run1 and Run2
- Proposal summary
  - architecture and DB organization
    - on going prototype, access only inside CERN GPN
- Short term plans
  - plans for the next year and man power
- Link to previous presentations….
  - Tim workshop in Genova, https://indico.cern.ch/event/342881/session/9/contribution/21/attachments/1159952/1669544/Report_from_DB_TIM.pdf
  - CHEP 2015, http://indico.cern.ch/event/304944/session/3/contribution/5/material/slides/0.pdf

present infrastructure

- COOL API
  - COOL API was developed in C++, and used as a client library to access tables inside Oracle, Sqlite (& MySQL?)
  - Successfully used during Run1 and Run2
- DB schemas today
  - One schema per system (~20x2 in total, online/offline)
  - N (~4) tables per payload type (the COOL node)
  - COOL DB instances for MC and Data (and monitoring)
  - Metadata(Nodes, Tags, Iovs) + Payloads
- Architecture
  - client-server model not scalable enough for large *grid* processing model
    - we have added an intermediate server (Frontier) to handle DB requests in distributed computing (caching of data is essential)

some caveat…

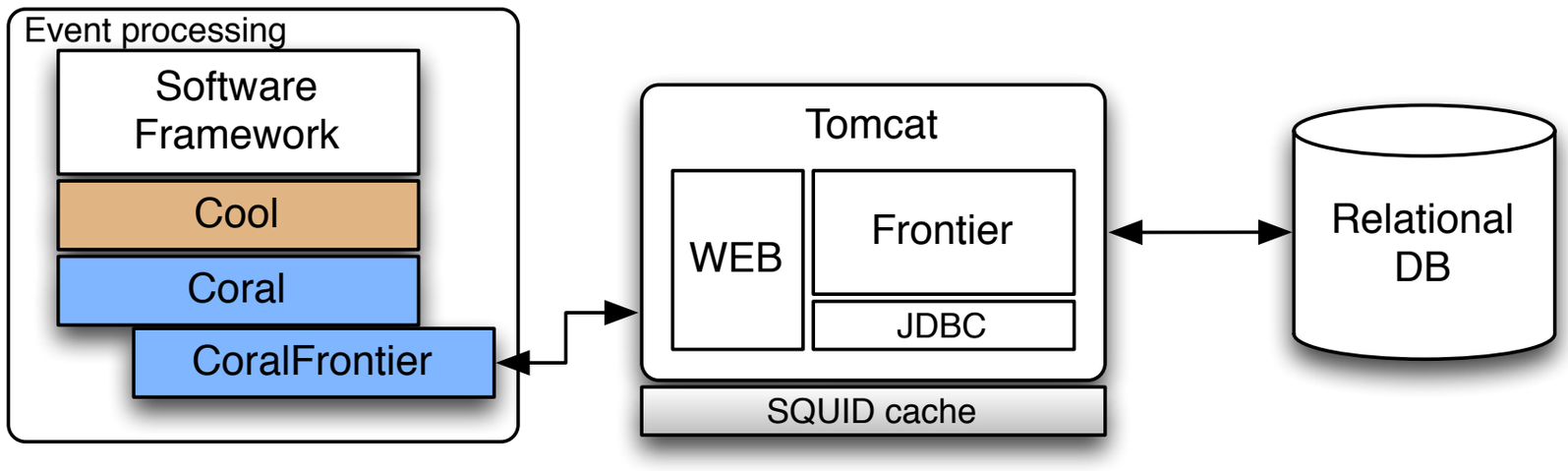
- Global Tag integration…
  - sort of “addons” to COOL, implemented today via cumbersome parent-child links
- …and protection mechanism (UPDx)
  - this is completely outside the DB, and is implemented via python scripts
- payload choice
  - lot of flexibility was asked to COOL API, and then used in our code, ending up with ad hoc solutions for every system (CLOBs, BLOBs, DB standard types, urls to POOL files,…)
  - in general, data formats have been never documented and are bound to one language
- Metadata and DB organization
  - we know pretty well today what are the main metadata needed for Conditions Data management: the concepts of Global Tags, Tags, and Iovs are important for our data flows.
  - DB is today pretty large: ~20x2 schemas, ~1000 Nodes => ~4000 tables for 1 DB instance (e.g. CONDBR2) and the total volume sums up to something like 1.5 TB for Run1
  - DB administration of large amount of schemas and tables can be a problem in the long term.

..and other uncertainties

- COOL maintenance
  - rely essentially on one person (A.Valassi) from IT
  - this is true for Coral as well
- DB platforms
  - even if abandoning Oracle is not an option for the moment we have to be careful in relying too much on it
  - adding additional support for new DB platforms (e.g. Postgresql) in Coral is a new development

present architecture

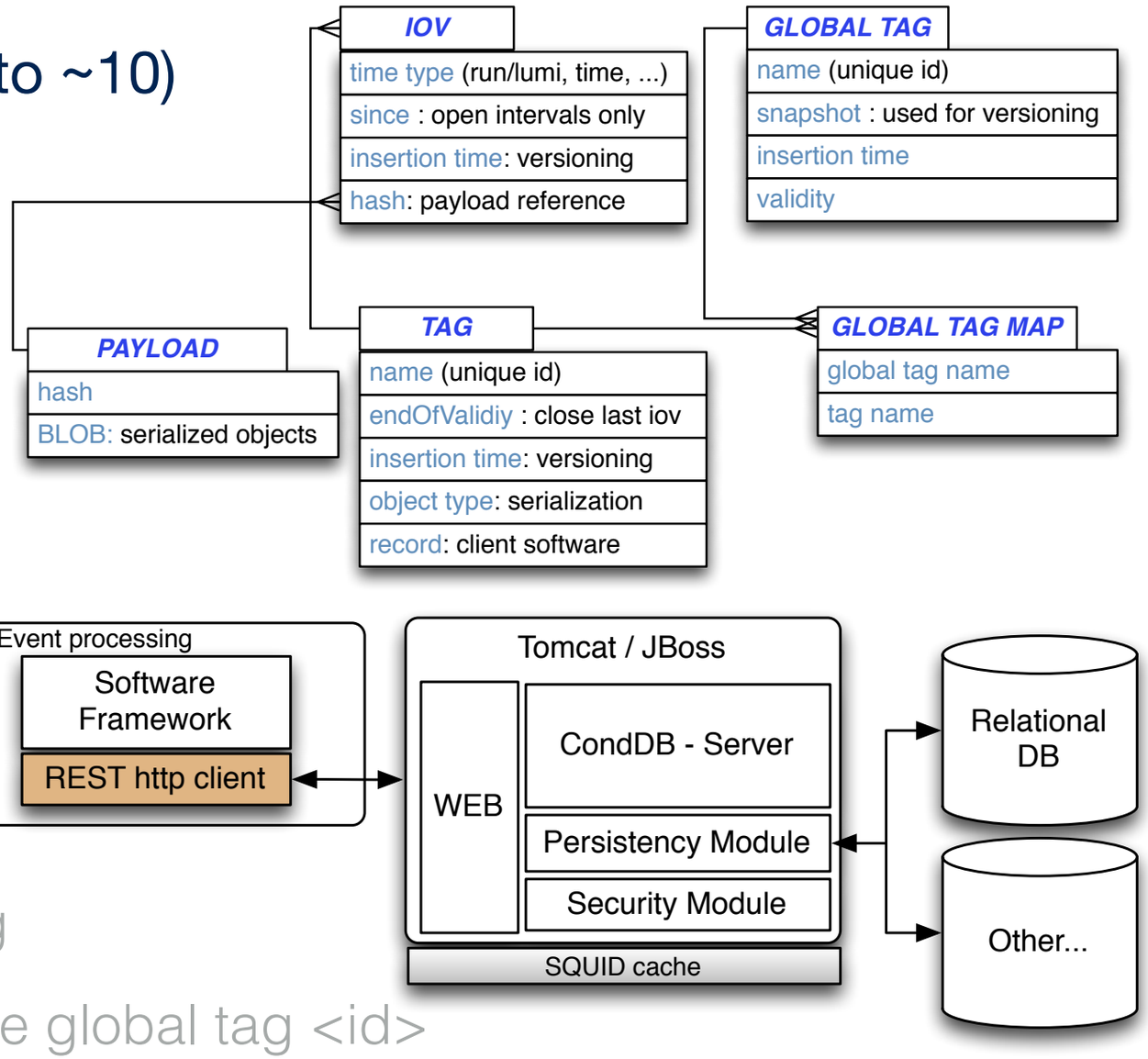
- Architecture
  - client-server model being not suited for grid processing we have added an intermediate server, providing a REST access to DB content



- Frontier model (READ-only, based on HTTP, allows Squid caching)
  - the middle-tier server sends generic (*select*) queries via JDBC
  - the output is packed in XML blobs, containing the whole ResultSet from the query
  - client libraries (COOL+Coral+CoralFrontierPlugin) are used to generate the queries and parse the output

proposal in short

- Extend the role of Frontier server for conditions data access
  - multi-tier model architecture using a full REST interface to the DB storage system (GET, PUT, POST, DELETE)
- Reduce the number of tables (from ~1K to ~10)
  - highly simplify the DB administration tasks
  - make payload storage opaque to the conditions server (a BLOB in the DB)
- Simplify the client side code
  - no relational DB libraries: only HTTP
  - messages from server to client are in JSON



DELETE: expert/tags/{id} : delete tag <id>
POST: expert/tags {tag-obj} : create a tag

GET: rest/globaltags/{id}?trace=on : trace global tag <id>
rest/tags/{id} : list info on tag <id>

prototyping

POST /expert/tags/{id} — Update an existing Tag.

globaltags — Show/Hide | List Operations | Expand Operations

GET /globaltags — Finds all GlobalTags

GET /globaltags/{gtagname} — Finds GlobalTags by name

Implementation Notes

Usage of % allows to select based on patterns

Response Class (Status 200)

Model | Model Schema


```
[
  {
    "name": "string",
    "validity": 0,
    "description": "string",
    "release": "string",
    "lockstatus": "string",
    "insertionTime": "2015-11-04T21:13:47.046Z",
    "snapshotTime": "2015-11-04T21:13:47.046Z",
    "globalTagMaps": [
```


Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
gtagname	(required)	**name pattern for the search**	path	string
trace	off	trace {off|on} allows to retrieve associated global tags	query	string

- doc http://aiatlas062.cern.ch:8080/swagger/
- code https://gitlab.cern.ch/formica/PhysCondDB

- couple of machines inside GPN
  - aiatlas: 063 and 137
  - DB: INTR (loaded with metadata from COOL)
  - caveat
    - can be removed at any moment !!
- Initial documentation for testing
  - swagger for (almost) automatic generation
- python and C++ clients
  - clients development is on going

main differences

- Global Tags
  - speed up resolution of associated tags from O(100) to 1 query
- IOVs: from since/until to since only (open ended)
  - COOL: allow to “close” an IOV
  - CondDB: always open-ended (only *last* IOV can be closed)

queries more simple
no holes in the time line

- Channels
  - No channels: handled inside the payload

cannot update only 1 channel

- Payload (from *free* to BLOB)
  - Format to be defined, CondDB server and DB are not aware of payload content
  - impact at client level

opportunity to define data formats
payload are loaded only on demand
cannot perform payload queries

core software

- IOVDbSvc
  - having open-ended IOVs implies that we should query a “large” number of IOVs (similar to the present caching mechanism in any case…) at job initialization
  - CMS uses the concept of “pages” to minimize the amount of data loaded in the system
    - only IOV *pages* are loaded at init
    - one *page* is used then to access the full list of IOVs
- AttributeLists and similar
  - we should simplify this part: today we have many *attributelists* - like objects (one should be enough !)

development plans

now

2016 | 2017 | 2018…. | R3

WP1-3: dev. prototype
Daq Dcs: storage studies
Trigger: Frontier testbed
Monitoring: dev prototype

Complete development
Start data migration

Validation
Tests

- Focus is on prototype:
  - we need to push the exercise far enough before taking a final decision on how to proceed at the end of 2016
- DAQ and DCS (today the main user of single version folders in COOL, and also main contributor to our conditions data volume inside Oracle)
  - start exploring alternative solutions for the DB part while also testing the proposed schema
  - may be have a look to what CMS is doing for DCS ?
- Trigger
  - we should start studying the existing Frontier+Squid solution and see where are the limitations
- Monitoring
  - start developing something already useful in today's infrastructure

summary

- Prototype development for new conditions data architecture (for R3) is on going together with CMS
  - collaboration with CMS database coordination for sharing informations and gather feedback is essential since they are already using in production the new DB schema
- Validation of the prototype is foreseen in one year from now
  - this implies that we also need core software components to be ready in order to test the full chain for few systems
    - IOVDbSvc (and related components) should be re-written : major development to perform in coordination with the activities on the new Athena Hive framework (understand multi-threading issues)
    - Payload serialisation needs to be explored : this is also a major aspect of the future system
- Other studies
  - we need in parallel to follow up on all usage of the conditions, e.g. DCS, Trigger, …to guarantee that the DB schema and the architecture do fit all our needs

use cases

- Validate the coverage for existing use cases
  - support existing data flows: **online / offline**
  - **montecarlo** productions
  - local mode : allow users to deploy a conditions server for their own testing (the substitute of the present *sqlite* usage)
- Additional use cases
  - **user analysis calibrations** (today handled separately from Conditions database)
    - we already started to explore needed functionalities together with Will and Attila
- Special cases (to be studied carefully !)
  - DAQ and DCS: single version conditions
    How do they fit in the present and proposed architecture ?
  - TRIGGER: special requirements in data access and caching
    Can we replace the present CoralServer caching architecture ?