Desarrollo de API y rediseño de la base de datos asociada para CMS

API development and database redesign for CMS

Álvaro Rodríguez García

Dirigido por Fernando Sáenz Pérez

Curso académico 2019-2020

TRABAJO DE FIN DE GRADO

DOBLE GRADO DE INGENIERÍA INFORMÁTICA - MATEMÁTICAS

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Resumen:

CMS (Compact Muon Solenoid) es un detector de partículas situado en el LHC (Large Hadron Collider), el gran acelerador de partículas del CERN, en Suiza. Los metadatos recogidos durante el funcionamiento del detector se escriben a una base de datos relacional. El sistema web que muestra estos datos está siendo completamente reescrito y rediseñado, y el nuevo sistema utiliza como backend una API, RESTful y escrita en Java, que expone diversos datos de orígenes muy distintos. Para reunir estos datos, el antiguo sistema utilizaba diversas tablas que agregaban la información necesaria. Sin embargo, la mayor parte de estas tablas han quedado obsoletas, y deben ser rediseñadas, junto con toda la infraestructura necesaria para mantener la información actualizada y consistente. Por tanto, el objetivo de este proyecto es rediseñar la parte afectada de la base de datos y construir sobre esta una API que devuelva la información relevante.

Etiquetas: CERN, CMS, RESTful, SQL, PL/SQL, Java, API, backend.

Abstract:

CMS (Compact Muon Solenoid) is a particle detector part of the LHC (Large Hadron Collider), the particle accelerator of CERN in Switzerland. The metadata gathered during the operation of the detector are written to a relational database. The web system displaying these data is being fully rewritten and redesigned, and the new system uses a RESTful API written in Java, which exposes diverse data coming from different sources. To store these data, the old system used several database tables aggregating the necessary information. However, most of those tables are obsolete, and must be redesigned, along with the whole infrastructure used to update the information and keep it consistent. The objective of this project is, then, to redesign the database and build upon it an API that returns the relevant information.

Tags: CERN, CMS, RESTful, SQL, PL/SQL, Java, API, backend.

Table of contents

Acknowledgments

I would like to express my very great appreciation to my advisor, Professor Fernando Sáenz, for his careful reviews of my drafts and his very valuable suggestions to improve them.

I would also like to thank the whole OMS team at CERN: Remi Mommsen, Ulf Behrens, Mantas Stankevicius, Audrius Mecionis, Gilbert Badaro and André Govinda. Their assistance throughout my internship was invaluable. I am particularly grateful for the assistance given by Ulf Behrens, who was always there to answer my questions and to help me refine my ideas.

Furthermore, I would also like to commend CERN as an organization. Its Technical Student Programme and Admin Student Programme are extraordinary opportunities for undergraduate and master students. The chance to contribute to a project so big is truly astonishing, and an incredible learning experience for a student.

Lastly, I would like to thank my parents and sister, for their unwavering support.

1. Introduction

1.1 The Online Monitoring System

1.1.1 Context: CERN, LHC and CMS

The European Council of Nuclear Research, better known for its acronym CERN, is a European research organization which manages the largest physics laboratory worldwide. Founded in 1954 [26], CERN has played a key role in the advancement of the field of particle physics during its more than 60 years of history.[1]

CERN is widely known for hosting the largest particle accelerator on the planet: the Large Hadron Collider (LHC) [25]. The accelerator consists of a 27-kilometer ring made of superconducting magnets, which direct the particles along a tunnel. Two beams go in opposite directions in two separate pipes, kept at ultrahigh vacuum. After being accelerated, both beams collide in one of the four points of the ring circumference where the detectors are placed. ATLAS, CMS, ALICE and LHCb detect the results of these collisions and register data which are analyzed and used to test different theories of particle physics. LHC has been in operation since 2008, when the first tests were run [18], and it has been the source of several breakthroughs in particle physics. The most important of these is the experimental confirmation of the existence of the Higgs boson in 2012 [4].

I was hired by CERN as a Technical Student, with a 12-month contract (that was later extended to 14 months). I had the opportunity to collaborate with the CMS DAQ and Trigger group, which is part of the CMS experiment. Compact Muon Solenoid (CMS) is one of the four big detectors situated in the LHC tunnel. It is one of two general purpose detectors, together with ATLAS, studying a broad range of physics that includes the Standard Model but also other, more exotic, topics such as the search for dark matter candidates [5]. The DAQ and Trigger group (Data Acquisition and Trigger) is responsible of data acquisition and filtering of the collision data recorded by CMS.

1.1.2 History: the Online Monitoring System

Not all data registered by the detector are data coming directly from the collisions taking place in the accelerator. Non-event data, the "metadata" which describes the behaviour of the system, also have a big importance. This importance is twofold: on the one hand, engineers and technicians need to have information about the performance and problems' of the system to be able to fix its issues. On the other hand, physicists need to know when datataking happened, and under which conditions, to be able to correctly analyze the actual experimental data. To fulfill both needs, metadata include all the information related to the beam, to fills and runs, to the number

[1]This paragraph and other parts of the introduction were initially part of *Studentship at CERN*, my internship report written for the "Prácticas en empresa" course (February 2019).

of collisions, to the uptime and downtime of the detector, to its deadtimes, to event filtering, and much more.

To compile and manage all these data, Web Based Monitoring (WbM) was created more than a decade ago [20]. It consists mainly of a web interface that shows all relevant information in the form of tables and plots of different types. However, an uncontrolled addition of new pages, by different developers who did not follow a unified standard, led to a system which has been increasingly complicated to maintain and expand. Not only that, but the technology that WbM was built with quickly became outdated, and both the codebase and the external appearance of the system needed an upgrade.

It is for all these reasons that in late 2016 it was decided to move away from WbM in favour of a new system [20]: the Online Monitoring System (OMS). OMS is expected to replace WbM as the system to view CMS metadata, in order to be used by all scientific and technical personnel in the experiment. To achieve that, the system should not only be able to show new data, but it should also have backwards compatibility with pre-existent data.

The LHC was shut down for scheduled maintenance in December 2018 [24], marking the start of Long Shutdown 2. It will be restarted in May 2021 [1], to run for almost 4 years in what is called Run 3. The shutdown time will be used for maintenance and upgrading, since big-scale time-consuming changes can only be implemented during one of these shutdowns. In particular, although work on OMS started before the shutdown, development is on track to be completed during it, so WbM can be turned off and replaced without affecting the end users.

Summarizing, my studentship has taken place within the framework of OMS development with the final aim of having it online and available at some time before Run 3.

1.1.3 Project structure

To organize development and separate responsibilities in the code, the project is divided in three clearly-defined parts: database and data warehouse, Aggregation API, and web portal. The web portal queries data from the Aggregation API, which on turn queries the database.

Database and data warehouse

The database provides all the necessary information to the other layers. The CMS database is an Oracle database, which is used by hundreds of people and features a complex and interrelated system of tables, views, triggers and procedures. It is the common destination for the information obtained from countless hardware components, placed at different points of the system. The data controlled by the OMS team is only a small portion of the total stored data, but OMS displays certain data controlled by external groups too. This adds another layer of complexity to the system, because we need to query data from some tables that we cannot change nor optimize.

Aggregation API

The Aggregation API is the intermediate component between the database and the graphical interface. It consists of a Java-based server, which exposes the necessary data as an API which returns it in JSON format. These data sometimes come straight from the database, but more often it is necessary to aggregate them in different ways, giving the Aggregation API its name.

During development, aligning with existing API standards and keeping the API intuitive has been of great importance. Many WbM users regularly used scripts that scraped the data from the old pages for varied purpose. We expect all those users to migrate to our API at some point in the future. Thus, keeping the external interface of the API as clean and intuitive as possible is of the utmost importance. Furthermore, this has led to the API providing additional data that will not be displayed by the graphical interface. In this sense, the API is not just subordinated to the needs of the GUI but a project in itself.

Web portal

The last layer is the front-end of our application. It shows the data provided by the API, either via tables or various kinds of plots. As opposed to WbM, it is built to simplify to the fullest the process of creating new pages and editing old ones. In fact, adding new code is only necessary when adding a custom component, and tasks such as creating a new table or editing the data source of a plot can be done without modifying the code at all. This is a system where we expect fast-changing requirements, and WbM struggled to keep up with the requested changes and new features. OMS should simplify the process, allowing simple changes to be done without leaving the web interface, and providing a generic framework to help with more complex changes.

1.2 Project organization and my contribution

At the beginning of my internship, the OMS project was about one year old. The whole infrastructure was already established, and early versions of both the API and the GUI were already released to the end users for feedback purposes, although still considered in development.

1.2.1 Status on arrival

This early version of the system still missed many features and especially many pages. From the point of view of the API, the core functionality was present: the server was working and the most important endpoints were already created (eras, fills, runs, lumisections...). However, more than a dozen endpoints were missing. Besides, the API lacked a general framework: in these early stages, there was no custom layer between the libraries providing the server functionality and each endpoint, that is, each endpoint had to manually query the data and process them for displaying.

Early efforts in this direction had already been made, and the best example of these is the query builder: a class that built a query from a list of URL parameters. However, most endpoints still had duplicated functionality, which did not only affect the code quality, but also made much more difficult to create a new endpoint, since there were almost no helper classes providing common functionality. Even worse, most endpoints had a very similar structure and even shared big parts of code, with the maintenance and bug fixing difficulties that this entails.

From the GUI point of view, an early version of the OMS webpage was already running, and displayed almost everything that the API offered. The backbone of the page was already in place, and it was possible to create new pages, portlets, etc. with relative ease.

The weakest side of the project at this point in time was the database, since the views and tables of WbM were still being used. The consequences could be felt throughout the system, with slow and often difficult to write queries, that could not be improved without modifying the underlying tables. This problem was exacerbated by the usage of the old WbM queries in several endpoints directly, without trying to refactor the SQL code at all.

1.2.2 Objectives

The aim of the project can be easily stated: to have a fully functioning solution to replace WbM from the start of Run 3 (2021), and that can last for another 10 years with minimal maintenance. In this context, the main objective of my internship was to improve the API to reach a production-ready state, and to upgrade if necessary other systems the API depends on, such as the database.

1.2.3 My contribution

Aggregation API

Initially, I was hired to collaborate with another colleague on the Aggregation API. However, in a couple of months I became the main maintainer of the Aggregation API, while my colleague started devoting most of his time to other issues. While, of course, we discussed the direction of the project among ourselves and often with the whole team, I was the person tasked with implementing most of the features and bugfixes that were needed.

This means that, during the year, I wrote more than a dozen new endpoints: conditions, deadtimes, deadtimes per range, deadtimes per scaler, diplogger, HLT config data, HLT path info, HLT path rates, HLT prescale sets, lumisummaries, run parameter, runtime analysis, stream summaries, trigger modes... Some of these were not straightforward and required changes on the general structure to support them. In addition, I fixed more than a hundred big and small bugs in old and new endpoints alike.

However, I was also given much freedom to improve the API base classes and to automatize as much code as possible. From the beginning of my internship I worked on these topics, and they took a very big part of my time. Among these changes, the most important are listed here:

- New base superclass hierarchy to abstract away most code from the concrete endpoint files.
- New query builder to automatically build queries for a given URL request. The previous query builder was too limited and lacked support for several different use cases, so a new one was built from scratch.
- Support of tuple comparisons on filters (using lexicographical order).
- Support for new URL parameters to return a grouped result (every row returned is an aggregation of rows) instead of the raw database output. This feature is very useful for plotting.
- Support for new URL parameters to specify the granularity of the results, which is translated to an SQL `GROUP BY` clause. Filters are also translated automatically to be compatible with the grouping. The translation is customized via annotations.
- Support for advanced timestamp and time interval URL filters.
- Annotations allowing heavy endpoint customization without extra code (for example, to throw an error if certain filters are specified).
- Automatic verification of URL parameters for a given pattern, to catch wrong requests early on the deserialization process.
- Automatic CSV serialization of most endpoints.
- Support for unit conversion while mapping columns from the database.
- Out-of-the-box support for complex queries where the resulting rows need to be reduced instead of mapped.

The entries on the list will be explained with detail later in this document, in Chapter 2 or, for the least important of them, in appendix A.

The improvement of the common framework of the Aggregation API led to very big stylistic and behavioral differences between the endpoints created before the changes and the ones created afterwards, which are much simpler, with most of their functionality hidden in the framework.

Thus, during my last working months I upgraded almost every old endpoint to use the new options available, often deleting hundreds of lines of code whose functionality could be replaced by the framework, and merging several endpoints that could be queried as one thanks to the new options.

Database and queries

Around the middle of my stay, we decided to move away from the database tables of WbM, creating a new and modern schema with the data that we needed. I was tasked with designing and implementing the new schema, although most of the triggers and procedures to insert the data into the tables were written by one of my colleagues.

Furthermore, I rewrote a lot of the API queries to use the new tables, and improved those that were using external tables. In some cases, I created views to hide away some of the complexity of the queries and to simplify the Java code dealing with these tables. Both the new schema and the new queries are described in Chapter 3.

Subsystems

Lastly, I collaborated to bring OMS to subsystems. This was a central focus of our efforts from the API side, and also a very important source of feedback. The main aim was for subsystems to have a workspace on the main OMS website where they could display their own custom data. This means that subsystems had to be able to extend our project to create and deploy their own API server, but also that the process of endpoint creation should be as straightforward and understandable as possible.

We tried to cater to both the "basic" user, who only wants to display the information of a simple table and does not want to write Java code for that, and the advanced user, who wants to edit their endpoints to allow custom behaviour. To achieve this goal, we provided the aforementioned framework with all our endpoints as examples, but also a tool that is able to convert a given database table into a working Java endpoint. Moreover, subsystems were a big source of changes and suggestions for the API, which had to adapt to meet their needs.

The communication with the subsystems was the main task of one of my colleagues, but I implemented some of the technical requests by myself and met with several representatives of the subsystems to teach them how to create new endpoints and to gather feedback. In particular, I designed the scripts to migrate the last value pages of WbM (which will be shown later), and collaborated on the design of the portlet to display them (which was my one and only contribution to the GUI). Chapter 4 explains my contributions to subsystems in detail.

1.3 A word from Dr. Ulf Behrens

To summarize the impact of OMS and my contribution to the project, we can include the following quote from Dr. Behrens, who was my supervisor at CERN:

> OMS is a mission critical tool and is crucial to operate efficiently CMS, one of largest physics experiment in the world. During the data taking periods (run 3 will start end of next year and is going to last until 2025), OMS has to be up and running 24/7.
>
> OMS is built from scratch and Álvaro was member of the team when essential decisions on database structures, tools and methods to access the data had to be taken. The whole project profited a lot from his brilliant ideas, his profound knowledge and his systematic and future-proof approach to solve problems. As a clear sign of the high

level of appreciation for his work he was hired as an external consultant after his time at CERN. For the databases and the Aggregation API, everything is carrying his signature.

2. Aggregation API

The Aggregation API was the main focus of my work at CERN. This chapter will describe the API and list the most important of my contributions to its base framework. By *framework* we understand all the classes that are shared between endpoints and that include common functionality. Most of these classes either extend a class of one of the libraries used or provide some functionality missing in those libraries. Note that all the features described here were added by me during my studentship but are not necessarily listed in chronological order. Several of them were added early on and then iteratively improved.

A general "user guide" of the API, which includes syntax and examples for some interesting features, can be found on appendix A. Some of the features described there are also discussed in depth in this chapter, and, for those, the appendix can provide interesting syntax and code examples that were not included in here. Other features are only mentioned in passing in this chapter, and the appendix can be useful to discover how those features work.

2.1 APIs and RESTful architectures

As an introduction to the Aggregation API, we will briefly discuss APIs and their architecture, and in particular we will introduce RESTful architectures and their advantages.

An Application Programming Interface (API) is an interface to a software that defines how other systems can use it. In this text, we will use the term API to designate Web APIs, which are interfaces to a web server. The API defines both a set of specifications to generate valid requests and the structure of the response messages.

A RESTful API is an API that implements the REST architectural constraints. Those were defined by R. Fielding in his PhD thesis [9], where he first introduced REST, and are:

Client-server The typical requirements of a client-server architecture shall be fulfilled: a server provides a service that can be requested by a client.

Statelessness Each request from the client must include all information necessary to understand the request. This implies that the server cannot store any context, and all state of the session is kept on the client.

Cache Data returned from the server should be implicitly or explicitly marked as cacheable or non-cacheable. Cacheable data can, then, be reused by the client for an equivalent request, instead of querying again. Caching can eliminate some client-server interactions, improving scalability and performance.

Uniform interface This feature separates REST from other network architectures. It is based on four constraints:

- Identification of resources: resources are identified in requests. Resources are decoupled from their representations, and the same resource can be returned with different representations (for example, the Aggregation API can return some resources both as JSON or CSV).
- Manipulation of resources through representations: the representation of a resource includes enough information to modify or delete that same resource.
- Self-descriptive messages: information about how to process a message is included in the messages themselves.
- Hypermedia as the engine of the application state (HATEOAS): the server should provide links dynamically to allow a client can discover all the available actions and resources starting from a single initial URI.

Layered system The architecture could be composed with hierarchical layers, and the client should not be able to determine whether the information is returned directly from the immediate layer or whether it is returned from elsewhere. This enables, for example, a proxy or load balancer to be placed between the client and the server, without changing the code of either of them.

Code on demand Optionally, REST allows clients to download and execute code in the form of applets or scripts, to extend its functionality.

RESTful APIs, such as our Aggregation API, are typically HTTP-based. Resources are exposed via unique URIs, called endpoints, which are queried using the standard HTTP methods (GET, POST, PUT...). Since the Aggregation API is defined as read-only, it exclusively uses GET requests. Data is returned with a media type that specifies the type of content retrieved, which is JSON for most endpoints of the Aggregation API.

2.2 How does it work?

This section describes the base which the API is built upon, with emphasis on the libraries used: Katharsis, Dropwizard, Jersey... This will be useful afterwards to analyze how does our custom framework extend this base.

2.2.1 Server libraries

The following libraries handle the server setup and configuration.

- **Dropwizard**: Dropwizard [8] eases the creation of a RESTful web service. It includes other stable, widely-used libraries to handle each different part of the process and connects all of them in one framework. It is easy to set up but highly configurable.
- **Jetty**: Jetty [15] is used to create the HTTP server. However, we are not interested in configuring anything at this level, and the server will be managed by Dropwizard.
- **Jersey**: Jersey [14] is used as the JAX-RS implementation. This means that it controls the request deserialization and serialization.
- **Jackson**: Jackson [11] is used as the JSON serialization library. We use Jackson all throughout the project, both explicitly (in our classes) and implicitly (as part of Katharsis).

2.2.2 Katharsis

Katharsis [17] is a library that manages a RESTful API conforming to the JSON API specification [16]. It tries to implement all the RESTful architectural constraints, with big emphasis in HATEOAS. It does not create the server itself, the user does it, usually with another library (Dropwizard in our case). However, it provides optional libraries that allow Katharsis to be easily hooked up with the most common server creation frameworks.

We are using two of the sub-libraries provided by Katharsis: `katharsis-core`, the core implementation that includes the functionality described above; and `katharsis-rs`, that enables compatibility with JAX-RS implementations, Jersey in our case.

When Katharsis was chosen for this project, it was a framework under active development. However, there has been no work done on it since mid-2017. The latest release is very stable and we have created our own fork to add some very small features that are missing and that cannot be achieved by extension.

2.2.3 Data Source: Oracle Database

All the data for the core API come from the CMS online database, an Oracle 11g database hosted by CERN. To handle our database connections, we use Jdbi [13], a JDBC wrapper that refines its rough interface. JDBC is the standard Java API to access SQL databases [12], and it is supported by Oracle databases.

2.2.4 How do they all work together?

Initialization process

The application is initialized on the `AggregationAPIService` class. This is a subclass of a Dropwizard `Application`. It has the `main` method of our application, which creates an instance of this class and calls the `run(String...)` method in the superclass. This method initializes Dropwizard and calls our overridden `run(AggregationAPIConfiguration, Environment)` method, which handles our customization. As part of this method, we set the values of several Jersey environment variables, we instantiate several classes used for serialization and deserialization (the Jackson mapper, the query spec deserializer, the custom type parsers...), and we add to Jersey the extensions provided by Katharsis.

Request lifecycle

We are only interested in GET requests, since the Aggregation API is read-only. When a request is received by the server, it goes through the phases listed in Figure 2.1.

First, the request is received by the Jetty server and sent to Jersey. However, Katharsis adds to Jersey a pre-matching filter with maximum priority, which means that this is executed before all usual Jersey processing. This filter deserializes the request path, and it is the one that decides how to get the data. There are several path types:

- Resource path: this is the normal access path to a resource, when the user queries an endpoint with `/endpoint` or `/endpoint/id`. It will also distinguish between the first case (resource list access) or the second case (single resource access).
- Action path: when a repository has methods annotated with the `@Path` annotation, Katharsis will try to create an action path. Action paths indicate that one of the `@Path` annotations has been matched. Katharsis has strong restrictions for these action paths: they must have

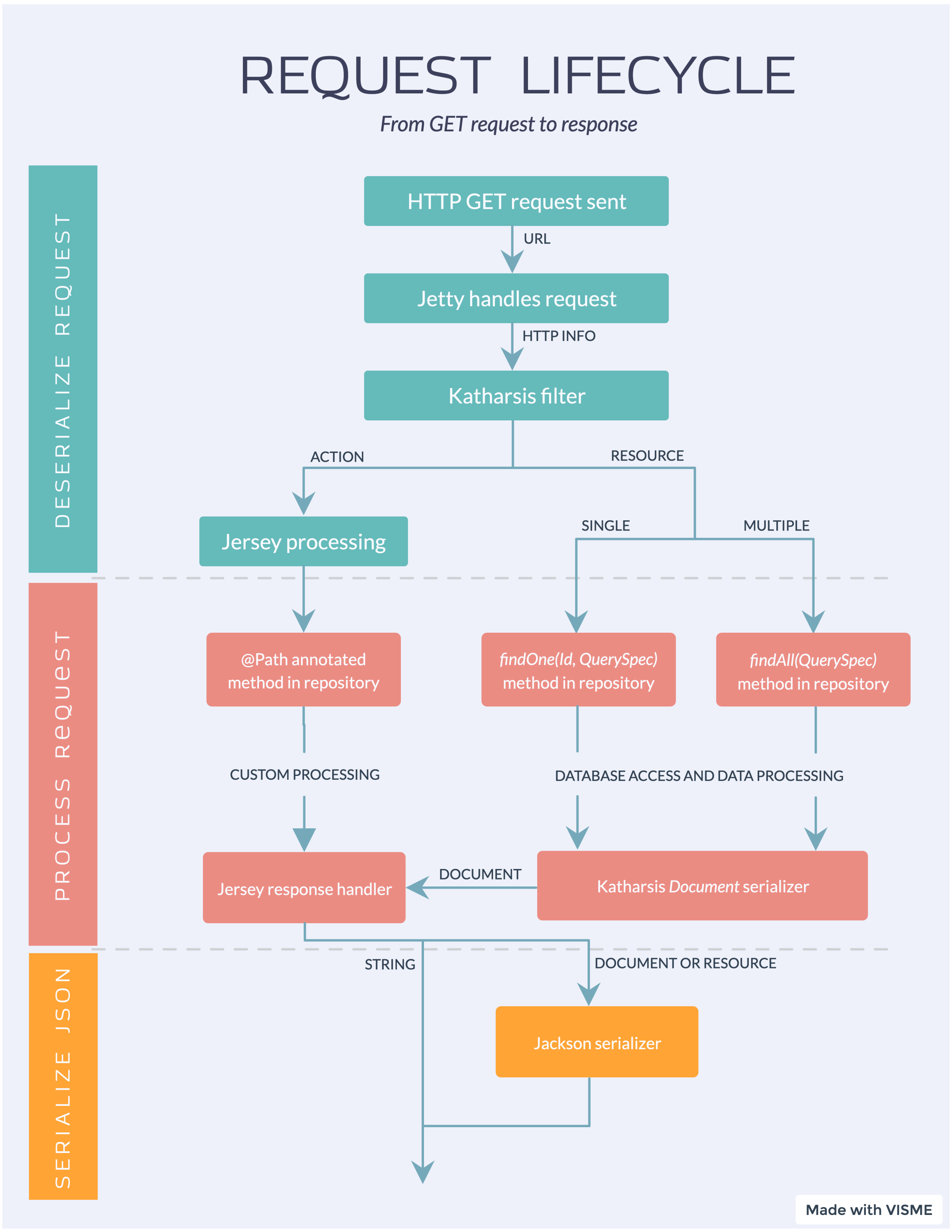


Figure 2.1: Request lifecycle flowchart. (Source: own elaboration)

a single path component and can only have two if the first one is an {`id`} variable. Besides, regex for variable matching is not supported, despite it being supported by Jersey. Our version of Katharsis, however, enables regex in a path annotation. Paths must still have one single path component *formally* (i.e., the path "template" shall not have a path separator `/`), but in practice paths like {`path: .*`} that match the full remaining path are accepted (and matched greedily).

- Other paths, in particular relationship paths. Katharsis supports relationships as another way of implementing HATEOAS, by allowing to query resources related to another resource from the URL of the latter. We rarely use relationships because they are not supported by the GUI. However, they are very similar to resource paths in how the data are obtained, the only different being that a relationship repository generates the data, instead of a resource repository.

After the path is deserialized, a decision is made. For resource paths (and relationship paths), Katharsis will handle the rest of the processing. This means that the internals of the framework will decide what repository and what method on it are the ones to be queried, and Katharsis will generate a `Document` object with the resources that were returned by the repository and, if present, the metadata and the links too. The document is built to be serialized straight away to a JSON response that conforms to the format specified in the JSON API specification [16]. The generated response is then delegated to the normal Jersey post-request handling, skipping possible processing by Jersey.

However, action paths are not handled directly by Katharsis, and processing is delegated to Jersey instead. It takes care of finding the adequate `@Path`-annotated method. The result of this method, which can be of any type, goes through the normal Jersey response handler. This is the point where the two streams join again, i.e., the `Document` generated by Katharsis also goes through here.

The Jersey response handler can decide if serialization is needed. If the response object is serializable (in particular, if it is a Katharsis `Document` object), it will be serialized (using Jackson). However, if it is, for example, just a `String`, it will be returned without processing as if it were JSON (with no validation). After this step, the HTTP response will then be generated, containing the serialized object.

Processing errors

When an error is thrown during a part of the process controlled by Katharsis, the error is handled automatically and a response is built, conforming to the JSON API specification [16]. We provide our custom exception handler, `AggregationExceptionMapper`, to process our own exceptions.

On the other hand, when an error is thrown in Jersey-controlled methods (for example, in a `@Path`-annotated method), no automatic handling is provided, and if left unhandled, the error will be an HTTP 500 error. In this case, we need to register another exception mapper to handle them. In our case, it is again `AggregationExceptionMapper`: we use the same class to handle Katharsis-side exceptions (by implementing the `JSONApiExceptionMapper` bundled with Katharsis) and to handle the same exceptions when they happen on Jersey side (by implementing the JAX-RS `ExceptionMapper`). Note that, for Jersey, we had to register our mapper manually when we initialized the API, but Katharsis discovers it automatically as part of its initialization process.

The result is that all subclasses of `AggregationException` should be handled correctly and throw an HTTP 40x error, no matter where the exception is thrown. Other exceptions that represent an internal error and are not caused by the user are uncaught and show an HTTP 500 error.

2.3 Query builder

2.3.1 Why a query builder?

When Katharsis receives an URL request, it parses its parameters to generate an object of type `QuerySpec`. This object contains all the information necessary to query and return the data: fields requested, filters, ordering... For almost every endpoint, we need to translate this information to an equivalent SQL query. For performance reasons, we do not want to bring all columns for a table from the database if only a handful of them will be needed. Similarly, is prohibitively inefficient to retrieve all rows from a database and then filter on the Java side. Lastly, database-side sorting can take advantage of indexes, not available on the Java side.

Our solution for this problem is the query builder class. This is a class that is able to generate a complete database query given a `QuerySpec`. It is in fact an abstract class, which is subclassed by each endpoint to customize it. The endpoints must specify in which column is the information for each attribute stored, how are tables joined, etc.

Arguably, libraries already exist to avoid building SQL queries manually. However, the main objective of the query builder is not to write the query easily, but to manage the process of translating the URL parameters to adequate database restrictions (`WHERE` clauses, sorting...).

2.3.2 History

Before my arrival, the project already had a query builder. However, it was not used in every endpoint, and had a very cumbersome syntax to initialize the tables and attributes. Besides, it was designed to cover simple queries, but more complex endpoints were requested often at the time.

The solution was a new query builder, completely redesigned, that kept only the interface from the old one. However, the table and attribute initialization syntax changed, which had a very obvious negative consequence: all classes that used the old query builder, now deprecated, had to be migrated to the new one. In fact, both query builders coexisted for almost a year, and only at the end of my internship we were finally close to completely phasing out the old version.

The decision to write a new query builder was not taken lightly. We had several discussions where we considered the risks of moving away from code that was already working. Our main worry was, of course, that writing a class that complex from scratch could result in code with too many bugs or that was simply not working, wasting development time. Besides, the expected development time for a new solution was much higher than the time needed to amend the old code. In fact, the decision was delayed several months for this reason. Ad-hoc additions to the old query builder were written during those months to support functionality that was needed at the time. It slowly became clear that the old query builder was not designed for the kind of extensions that we had in mind, and that trying to adapt it would severely impact code quality. The longer development time needed to create a new query builder was justified in this way, even if, at this point, changing the query builder meant upgrading a large number of old endpoints.

2.3.3 Interface

The query builder has two public methods: `createQueries()` and `createPaginatedQueries()`. The first one generates the usual query, and the second one generates a query that limits the number of results requested taking into account the `page[limit]` and `page[offset]` query parameters. The query generated by the second method is usually faster, but it has a problem: we do not know the total number of rows that match the filters, which is necessary to implement pagination both on the API side (via links to the previous and next pages) and on the GUI. That is why both this methods also return a "counter query". This is a special query that calculates the number of total items matching the request. It is also possible to customize the query builder to return an extra column storing the total count of objects, to avoid querying the database twice.

2.3.4 Query tables

All tables in our model are subclasses of `QueryTable`. The actual subclass of the query table is not relevant in most situations, and for a normal use case we will only need to differentiate between the abstract `QueryTable` superclass and the `QueryTableMaster` subclass. The three main types of tables are:

- **Master tables:** conceptually, a master table is the principal table from which data are coming from. The master table of an endpoint has to be added to the query builder by modifying the protected `masterTable` attribute.
- **Detail tables:** a detail table is a table which only adds information to the master table, i.e., it has at most one row per row of the master table. It will be joined with a `LEFT JOIN`, which means that its presence does not influence the number of rows returned by the query. As such, if the attributes using the detail table are not requested, then the table will just not be present in the query.
- **Joined tables**: this table, internally, is a `QueryTableDetail`, just like the previous one. However, the difference is that the join is an `INNER JOIN` and the table is always present, even if no attributes are requested. This means that the number of returned rows might be different after adding a joined table.

Grouping, connection and subquery behavior must also be specified in the table structure. A master table can be wrapped into another table, which will then act as the master table. This wrapping table can just add a `GROUP BY` or `CONNECT BY` clause to the original query, or it can be a subquery table, that is, it will generate a query where the original query for the master table is nested as a subquery in the `FROM` clause.

2.3.5 Query attributes

Query attributes are the core of the query builder. Each query attribute roughly represents an attribute in the response object. All query attributes are subclasses of `QueryAttribute`. They are implemented using the decorator design pattern: the subclasses of `QueryAttributeBase` store the core information about the attribute, and can be decorated with subclasses of `QueryAttribute Decorator`. These subclasses modify or extend the attribute behavior with respect to its definition in the `SELECT` clause, filtering, sorting... This means that attributes can be as complex as necessary, while keeping the code relatively understandable.

After creating an attribute, it has to be added to the query builder via `addAttribute()` or `addIdentifyingAttribute()`. These two methods have a very similar effect, but the latter additionally tells the query builder that this attribute should always appear in the `SELECT` clause, independently of whether the user requested them or not. It should be used with attributes which are part of the identifier, because we need to generate this identifier on the Java code even if its parts were not requested. For the rest of the attributes, they will only appear on the `SELECT` list if they are requested either via the `fields` query parameter or if they are used for filtering.

Decorator attributes, on the other hand, are available in a wide variety of flavours. Normally most decorators only focus on modifying one aspect of the attribute: its `SELECT` expression, its `WHERE` translation... Custom query attribute decorators can also be used for endpoint-specific behaviours. For example, in some endpoints a decorator is created that translates the URL parameter `filter[attribute]=latest` to a subquery in the `WHERE` clause retrieving only the latest row for the table.

Detailed `QueryAttribute` interface

The query attribute superclass is one of the most complex classes of the Aggregation API. The decorator pattern allows very complex attributes to be created without needing a subclass for every variation. Still, since the creation of the query builder we have had a lot of different requests and requirements, and both the number of attribute subclasses and the number of functions available on the base class have steadily increased.

Attributes generate directly the SQL code that is relevant for the field represented. We will review the main methods used for SQL code generation, along with the convenience methods that are typically overridden by subclasses. Afterwards, we will see some examples of concrete subclasses.

```
    public String getAliasedSQLSelectValue(List<SortSpec> parentSort,
            GroupingInfo groupingInfo, QueryInfo query) {
        String select = getSQLSelectValue(parentSort, groupingInfo, query);
        if (select == null) {
            return null;
        } else {
            return select + " AS " + getAlias();
        }
    }

    public abstract String getSQLSelectValue(List<SortSpec> parentSort,
            GroupingInfo groupingInfo, QueryInfo query);

    public abstract String getName();

    public String getAlias() {
        return getName();
    }
```

When the query builder generates a `SELECT` clause, the aliased SQL select value is obtained for each attribute to be included, using the methods above. Base classes always override `getSQLSelectValue()`, while decorators can add extra requirements (for example, a decorator can wrap it on a `NVL()` function if the decorator provides a default value).

```
    public final String getSQLColumnExpression(QueryTable inTable, QueryInfo queryInfo) {
        QueryTableMaster attrTable = getTable().getJoinedTo().getInternalTable();
        QueryTableMaster inTableMaster = inTable.getJoinedTo().getInternalTable();

        if (attrTable.equals(inTableMaster)) {
            // Both attributes are on the same level. This means we can just forward the
            // call to the attributes themselves.
            return this.getSQLColumnExpression(queryInfo, getAlias());
        } else if (attrTable.equals(inTableMaster.getSubqueryTable())) {
            // The attribute is on the next nesting level. We can just take its alias and
            // the table abbreviation and generate the column expression with that.
            queryInfo.getSubquery().addSelectAttribute(this);
            return inTableMaster.qualify(getAlias());
        } else {
            throw new QueryBuilderInvalidStructureException("The provided attribute is
            ↪  not on the same level as the containing attribute nor on the immediately
            ↪  nested level.");
        }
    }

    public abstract String getSQLColumnExpression(QueryInfo queryInfo,
            String topLevelAlias);
```

When a query attribute is referenced from a query on which it is nested, we need to obtain the "column" value, that is, an expression of type `table.attribute` or `subquery.attribute`. The previous methods obtain that expression. The same kind of reference is also used, for example, when using a renamed attribute on an `ORDER BY` clause.

The `QueryInfo` object already appeared in the previous code snippet, and it is important to understand the query builder. It stores the data from the query that is currently being built, and most importantly from its nested queries (which we will often call subqueries). Thus, attributes of the subqueries are added dynamically, depending on whether they are needed: every time that an attribute is used on the top level query that requires an attribute from the nested query, this information is saved on the `QueryInfo` to build the subquery properly. This is why the previous snippet saves the attribute in the subquery as soon as its column expression is built.

```
    public final String getSQLAnalyticClauseExpression(QueryTable inTable,
            QueryInfo queryInfo) {
        QueryTableMaster attrTable = getTable().getJoinedTo().getInternalTable();
        QueryTableMaster inTableMaster = inTable.getJoinedTo().getInternalTable();

        if (attrTable.equals(inTableMaster)) {
            if (isSelectableInAnalyticClause()) {
                return getSQLSelectValue(
                        Collections.emptyList(), new GroupingInfo(), queryInfo);
            } else {
                throw new QueryBuilderInvalidStructureException("Trying to use the
                ↪  attribute " + getName() + " in an analytic clause, but this is not
                ↪  allowed.");
            }
        } else if (attrTable.equals(inTableMaster.getSubqueryTable())) {
            // The attribute is on the next nesting level. We can just take its alias and
            ↪  the table abbreviation and generate the ordering expression with that.
            queryInfo.getSubquery().addSelectAttribute(this);
            return inTableMaster.qualify(getAlias());
        } else {
            throw new QueryBuilderInvalidStructureException("The provided attribute is
            ↪  not on the same level as the containing attribute nor on the immediately
            ↪  nested level.");
        }
    }

    public abstract boolean isSelectableInAnalyticClause();
```

Analytic functions, also known as window functions for other SQL flavours, are heavily used in our queries, and thus supported by the query builder. The method `getSQLAnalyticClause Expression()` is used to retrieve the expressions to be used as part of the `PARTITION BY` list or the `ORDER BY` list. The method returns the `SELECT` value, if it is queried from an attribute on the same level, or else the column expression of the attribute. It also checks that the attribute can be used on the analytic clause, to detect possible syntax errors (i.e., nested analytic functions) before executing the query.

```
    public abstract String getSQLWhereValue(FilterOperator operator, Object value,
    ↪  Set<String> flags, QueryInfo query,
            BindingMap bindings, boolean isPartOfCompositeFilter);

    public final String getSQLOrderByValue(Direction direction, QueryInfo query) {
        return getSQLOrderByValue(direction, query, getAlias());
    }

    public abstract String getSQLOrderByValue(Direction direction, QueryInfo query,
    ↪  String topLevelAlias);
```

The methods to generate the `WHERE` and `ORDER BY` clauses are much simpler. Both are left to the subclasses, just like we did for the `SELECT` clause. However, ordering is implemented by default by all base query attributes, since it is normally the alias of the column plus the sort direction; while filtering is usually handled by a decorator (since some rows are not filterable).

Base attributes

As discussed in Subsection A.1.3, there are five base attributes that can be decorated. However, the best example is by far the most used attribute: `QueryAttributeForColumn`. This class represents a query attribute that is read directly from a database column. We will show the superclass for all base attributes, and then the attribute itself, although only the most interesting methods will be shown.

```
public abstract class QueryAttributeBase extends QueryAttribute {
    private String name;
    protected QueryTable table;

    // Constructor and other methods

    @Override
    public Collection<QueryTable> getAllNecessaryTables() {
        return Collections.singletonList(getTable());
    }

    @Override
    public List<String> getRequiredAttributeNames() {
        return Collections.singletonList(this.getName());
    }

    @Override
    public String getSQLOrderByValue(Direction direction, QueryInfo query,
            String topLevelAlias) {
        String condition = topLevelAlias;
        if (direction == Direction.DESC) {
            condition += " DESC";
        }
        return condition + " NULLS LAST";
    }
}

public class QueryAttributeForColumn extends QueryAttributeBase {
    private String column;

    // Constructor and other methods

    @Override
    public String getSQLSelectValue(List<SortSpec> parentSort, GroupingInfo groupingInfo,
            QueryInfo query) {
        return qualifiedColumn();
    }

    @Override
    public String getSQLColumnExpression(QueryInfo queryInfo, String topLevelAlias) {
        return qualifiedColumn();
    }

    @Override
    public boolean isSelectableInAnalyticClause() {
        return true;
    }

    @Override
    public String getSQLWhereValue(FilterOperator operator, Object value,
            Set<String> flags, QueryInfo query, BindingMap bindings,
            boolean isPartOfCompositeFilter) {
        return null;
    }
}
```

There are some overridden methods that we have not discussed for `QueryAttribute`, but their only purpose is to provide a way for the query builder to know what tables and attributes should be included for the outermost query.

The most interesting method on the attribute base is `getSQLOrderByValue()`. It is the implementation of what we have discussed before: for most attributes, sorting is as easy as taking the projection alias and adding the sort direction. In our case we also specify that we want nulls to always be last.

For the subclass, `QueryAttributeForColumn`, it is important to realize that the implementation for the `SELECT` value and the column expression just returns the qualified column name. Of course, more complicated attributes can have more complicated expressions.

Decorator attributes

All decorators inherit from the `QueryAttributeDecorator` class, which is in turn a `QueryAttribute`. The decorator class simply redirects all the calls to the wrapped attribute. This provides a default implementation that allows decorators to override only the methods that they need to change, as our simplest attribute decorator shows:

```
public class QueryAttributeChangeAlias extends QueryAttributeDecorator {
    private String rename;

    public QueryAttributeChangeAlias(QueryAttribute wrappee, String rename) {
        super(wrappee);
        this.rename = rename;
    }

    @Override
    public String getAlias() {
        return rename;
    }
}
```

All the other methods are just handled by the wrapped attribute.

We have more than a dozen decorator attributes, and even more if we count endpoint-specific ones. Their functionalities can range from modifying the `SELECT` clause (for example, wrapping it with a SQL function) to changing the filter translation (for example, to have special behaviour for a certain filter operator). More examples can be found in Subsection A.1.3.

2.4 Repository base classes

Together with the query builder, the base classes were one of my most important contributions to the project. The base classes are extremely useful to reduce the amount of code that has to be used for a new endpoint. Over time they have significantly evolved to allow a lot of customization with very little code written on the endpoint.

Adding the base classes was considerably less risky than creating the new query builder. The most obvious risk was the possibility that the hierarchy of base classes was not general enough to fit the needs of most endpoints, rendering the base classes pointless. However, the structure of the already existing endpoint classes led us to believe that most, if not all, of them could benefit from this change. Another possible risk was that old endpoints were changed to use the new base classes, and then an important problem was found on the classes themselves. To minimize this risk, we decided to first introduce the new classes as a non-breaking change, without modifying old classes. We also added two new endpoints using the new base classes and rewrote a single one of the old endpoints. The general migration of old endpoints was only approved after the three new endpoints were carefully checked and tested.

2.4.1 Endpoint structure

To understand the impact of the base classes, some context on the endpoint structure is necessary. A basic endpoint has five core classes:

- **Resource class**: the class that specifies the JSON structure that will be returned by the API. It is normally a class devoid of logic, containing only class variables, getters and setters.
- **Mapper (or other result generator)**: the class that maps (or somehow transforms) the result of the query execution into a list of resources.
- **Meta class**: a class that generates the metadata for this endpoint.
- **Repository**: the class that encodes the behaviour of the endpoint. The framework parses the URL and then calls one of the methods in the repository, which then does all the work to get the requested resource(s) and returns them to the framework. For this reason, all repositories are subclasses of one of the base repository classes provided by Katharsis.
- **Query builder**: already described in Section 2.3.

2.4.2 Resource repositories

Given this structure, subclassing seems like the obvious choice to abstract some shared code away from the base classes. Most endpoints initially shared big portions of code, especially in the repository and the mapper. This led to the creation of common abstract superclasses to unify this code.

For the repositories, three abstract classes were created as possible superclasses:

- **BasicRepository**: subclasses the base Katharsis repository, providing a default implementation for several methods, but it is agnostic with regard to the method used to get the data (in particular, it does not assume that the data come from a database). The very few endpoints that take their data from a non-database source typically inherit from this class.
- **DatabaseRepository**: a subclass of the previous one, it is database-aware. It overrides the data retrieval methods to implement calls to the database, although the specific query would need to be provided by each endpoint. Endpoints that access the database but do not use the query builder typically inherit from this class.
- **QueryBuilderRepository**: a subclass of the previous one, it requires a query builder to generate the database queries. Its main advantage is that, for the average endpoint, it can be subclassed and it works almost out-of-the-box, only by overriding two simple methods and providing a specialized query builder.

The full code of the three superclasses is long and complex, and we believe that the best way to explain it is through a diagram, such as the sequence diagram shown in Figure 2.2. The rest of the section will list and explain the most relevant code snippets.

Basic resource repository

In Katharsis repository interfaces, there are only two methods that are relevant for a read-only repository such as ours: `findOne()` and `findAll()`. The sequence diagram only shows the latter, because the implementation of the former is much simpler:

```java
@Override
public ResourceT findOne(IdT id, QuerySpec querySpec) {
    QuerySpec idQuerySpec = querySpec.duplicate();
    fillQueryWithId(id, idQuerySpec);

    ResourceList<ResourceT> list = findAll(idQuerySpec);
    Iterator<ResourceT> iterator = list.iterator();

    if (iterator.hasNext()) {
        ResourceT resource = iterator.next();

        // Check uniqueness and throw if several resources are found.
        // Generate and add the meta.
        // ...

        return resource;
    } else {
        throw new ResourceNotFoundException("resource not found");
    }
}

protected abstract void fillQueryWithId(IdT id, QuerySpec querySpec);
```

Apart from the helper methods to generate the meta, the only called method is `fillQueryWithId()`. It is abstract, since its implementation is endpoint-specific. Normally, the identifiers are generated

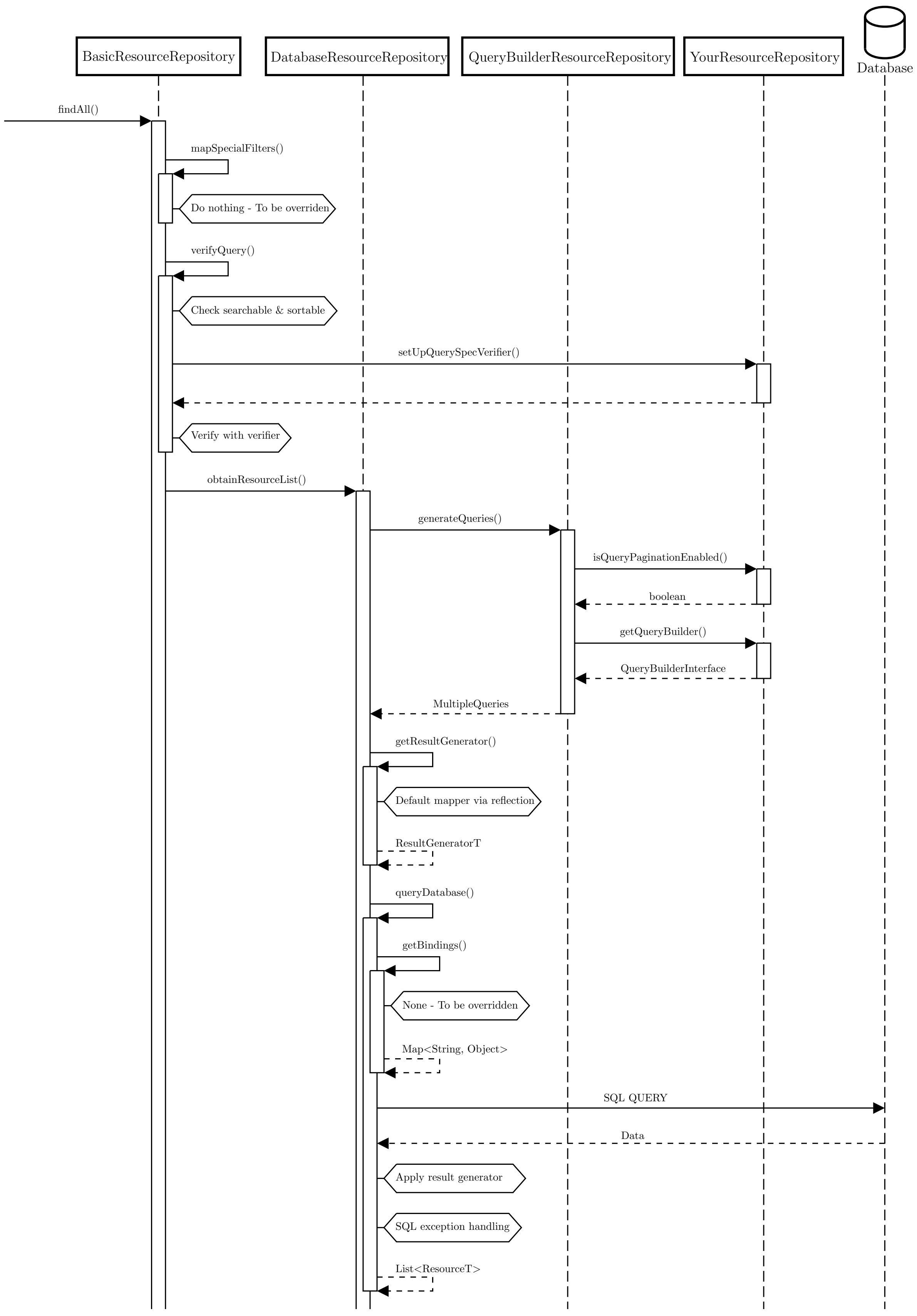


Figure 2.2: Repository implementation sequence diagram (page 1). (Source: own elaboration)

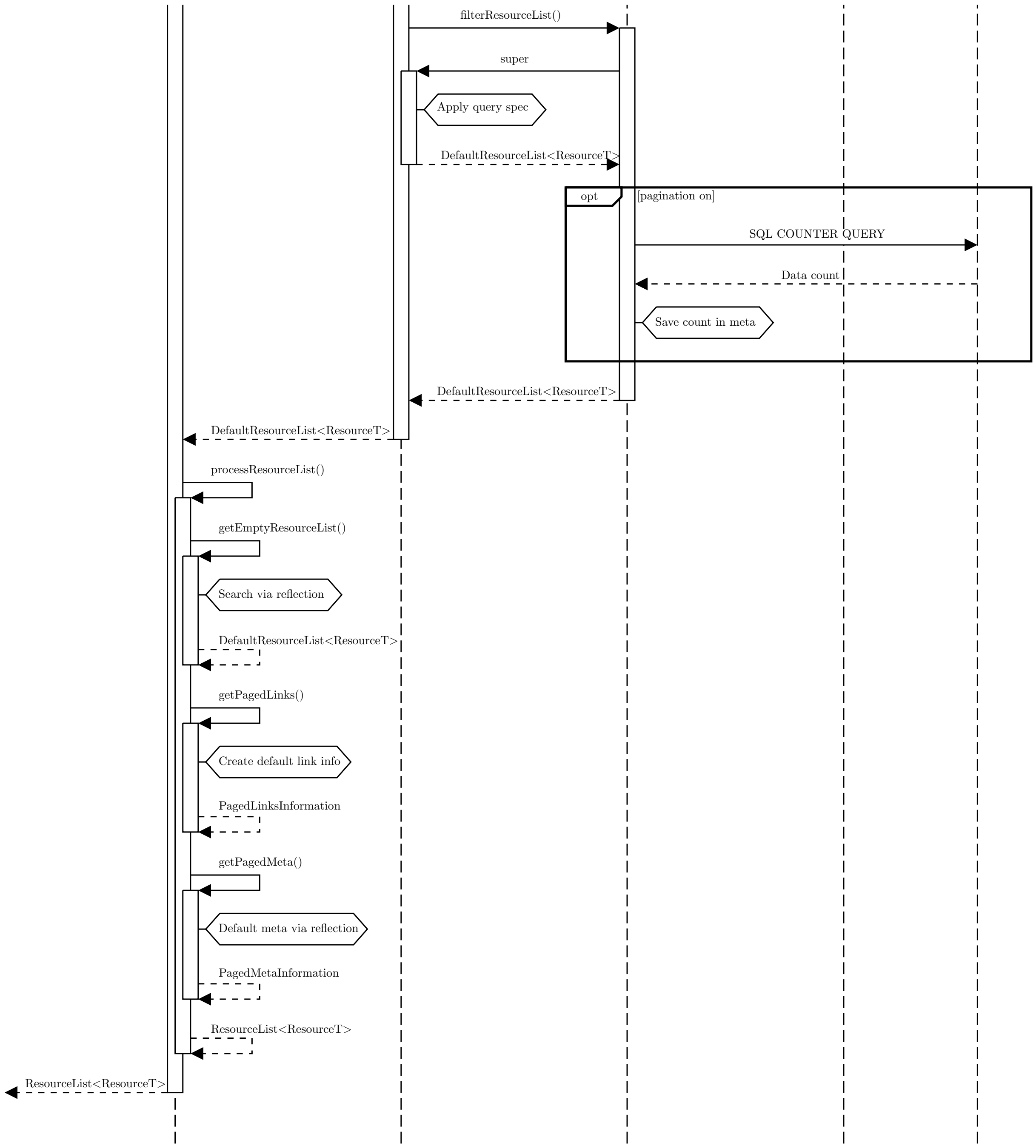


Figure 2.2: Repository implementation sequence diagram (page 2). (Source: own elaboration)

from the primary key fields of the table, joined with underscores. The typical implementation, then, just breaks down the identifier into its pieces and translates every part into a new filter.

The `findAll()` method is much more interesting. Its implementation has four steps on the `BasicResourceRepository`, clearly divided into functions, which means that subclasses should not override this method, and instead go for one of its parts. The code, in fact, is very simple: all the complexity is hidden away in one of the other methods.

```java
    @Override
    public ResourceList<ResourceT> findAll(QuerySpec querySpec) {
        mapSpecialFilters(querySpec);

        // We should never modify the original query spec after mapping the special
        ↪  filters, because we want the original to create the links information
        QuerySpec duplicateQuerySpec = getDuplicateQuerySpec(querySpec);

        verifyQuery(duplicateQuerySpec);
        return processResourceList(obtainResourceList(duplicateQuerySpec),
                duplicateQuerySpec, querySpec);
    }
```

The four steps are performed by the following functions:

1. `mapSpecialFilters()`: this method does nothing on its default implementation. It should be overridden by endpoints that offer "fake" filters, that is, filters that do not correspond to a database column or to an endpoint field, and are just added for convenience. Those filters should be translated to filters on real fields on this method.

2. `verifyQuery()`: this method performs query verification, to catch problematic requests very early on the process. This includes automatic verification, which checks that fields set as non-filterable or non-sortable in the meta are not included on the filters or sorts. However, the method also performs endpoint-defined query verification, for endpoints overriding the `setUpQuerySpecVerifier()` method. Verifiers are discussed in Section A.4.

3. `obtainResourceList()`: this is the function which returns the original resource list. Since the `BasicResourceRepository` is agnostic as to data retrieval, this method is abstract. However, we will discuss later how the repository subclasses implement it. This method is also responsible to return data that are already filtered and paginated.

4. `processResourceList()`: this method uses the generated list of resources and the original and duplicated query specifications to generate the appropriate metadata and link information.

Database resource repository

The implementation of the `DatabaseResourceRepository` provides all the methods needed to access the database, and the `QueryBuilderResourceRepository` only refines it insofar as it uses the query builder to generate the queries, adding a handful of convenience methods to toggle query builder settings. The implementation of `obtainResourceList()` on `DatabaseResource Repository` and a selection of the methods that it uses follows:

```java
    @Override
    public DefaultResourceList<ResourceT> obtainResourceList(QuerySpec querySpec) {
        // Generate the SQL query
        MultipleQueries queries = generateQueries(querySpec);

        // Query the database
        ResourceAndCounter<List<ResourceT>> resources = queryDatabase(querySpec, queries,
                getResultGenerator(querySpec), isRowCountingAttributeEnabled());

        // Save the retrieved row counter if it is not null
        DefaultResourceList<ResourceT> filteredResources =
                filterResourceList(querySpec, resources.getResource(), queries);
        if (resources.getCounter() != null) {
            DefaultPagedMetaInformation dpmi =
                    (DefaultPagedMetaInformation) filteredResources.getMeta();
            dpmi.setTotalResourceCount(resources.getCounter());
        }
        return filteredResources;
    }

    public abstract MultipleQueries generateQueries(QuerySpec querySpec);


    public ResourceAndCounter<List<ResourceT>> queryDatabase(QuerySpec querySpec,
            MultipleQueries queries, ResultGeneratorT generator,
            boolean isRowCountingEnabled) {
        BindingMap bindings = queries.getBindings();
        updateBindings(querySpec, bindings);
        Map<String, Object> bindingMap = bindings.getBindings();
        String queryString = queries.getStandardQuery();

        if (!isRowCountingEnabled) {
            return new ResourceAndCounter<>(executeDatabaseHandleCallback(h ->
                    generator.extractResources(h.createQuery(queryString)
                            .bindMap(bindingMap))), null);
        } else {
            List<ResourceAndCounter<ResourceT>> list = executeDatabaseHandleCallback(h ->
                    generator.extractResourcesWithCounter(h.createQuery(queryString)
                            .bindMap(bindingMap)));
            Long counter = list.stream().findFirst()
                    .map(ResourceAndCounter::getCounter).orElse(0L);
            return new ResourceAndCounter<>(list.stream()
                    .map(ResourceAndCounter::getResource)
                    .collect(Collectors.toList()), counter);
        }
    }

    public DefaultResourceList<ResourceT> filterResourceList(QuerySpec querySpec,
            List<ResourceT> resources, MultipleQueries queries) {
        // Default implementation for repositories with no extra setting
        if (shouldFilterDatabaseResult()) {
             return querySpec.apply(resources);
        } else {
            return ((AggregationQuerySpec) querySpec).applyOnlyPaging(resources);
        }
    }
```

Note that `DatabaseResourceRepository` has three type parameters: two that are already present on the base repository provided by Katharsis, the identifier `IdT` and the resource `ResourceT`; and the result generator `ResultGeneratorT`.

The implementation of `obtainResources()` is not overly complex and just agglutinates calls to other methods. The only real logic here is the last `if`-clause, that checks whether the rows have been counted as part of the previous steps, and if so sets the value on the metadata class.

The access to the database happens inside `queryDatabase()`. First, the default bindings for the repository and the extra ones generated with the query are merged. After that, the query can be executed in two different ways, depending on whether row counting is enabled (recall that we need to count resources on the database side because the database may already paginate the results, as we explained in Subsection 2.3.3). If it is not, the query is just executed traditionally; but, if it is, the total resource count is extracted from the rows immediately. Note that all this setup avoids a second call to the database, saving precious milliseconds of network delay. The manipulation is needed because every row has a row counting attribute, but they are all the same, and thus the first one is kept and the rest are discarded. Note that `executeDatabaseHandleCallback()` just adds custom exception checking to the `Jdbi.withHandle()` method, used to access the database. The `Jdbi` object, that handles a database connection, is already in all repositories as part of the initialization process.

The other interesting method shown above is `filterResourceList()`. The case distinction is important, again, for performance reasons. Endpoints that guarantee that the results coming from the database are already filtered and sorted (in particular, all endpoints using the query builder) can opt out of Java-side filtering, while the rest are filtered at this step.

Query builder resource repository

The usage of a query builder to generate the database queries is straightforward from the repository side, since the complex functionality is hidden on the query builder itself. However, the repository overrides `filterResourceList()` to provide two-step database counting. This feature consists on running two database queries, one for getting the actual resource rows, and a second one to get the total resource count. This option has been mostly phased out by the possibility of running both the resource query and the counting query simultaneously, but it is still used in some cases.

2.4.3 Mapper (and result generators) base classes

Typical endpoints map rows in the database to a row in the returned JSON file. These endpoints typically feature a mapper, which simply transforms the current row of a JDBC `ResultSet` to an instance of the resource class. Besides, all these mappers do share a similar logic: for all of them, we only want to map the fields that are actually requested. This has a tiny performance advantage, since it avoids writing to fields that will not be used. However, the real advantage is that with this we do not need to get these fields from the database (which in certain cases can be a very significant performance improvement), and not retrieving them from the result set means that no SQL exception will be thrown.

Originally, the solution was to create a `SimpleMapper` class with methods to generate the fields needed for a given query specification. This worked nicely, but some of the newest endpoints were more complicated, since they were not mapping a single row to a single model object, but several rows to the same resource. This kind of situation is typically solved with an accumulator, that is used to implement a reduce-like operation. However, our model consisted exclusively of mappers, which were not easy to extend.

The solution is a more general abstract class, `ResultGenerator`. Up until then, the mappers were responsible of generating the list of fields that were requested on the URL, to be able to map only the necessary columns. However, the new class decoupled this logic from the actual mapping, so that the old `SimpleMapper` could extend it, but also other classes such as accumulators. What is more, a `ResultGenerator` is able to apply itself, that is, result generators must implement a method transforming a Jdbi `ResultBearing` (the raw query results) to a list of resources. This is the implementation of the result generator:

```java
public abstract class ResultGenerator<ResourceT> {
    protected QuerySpec querySpec;
    private Set<String> requestedFields = null;
    protected boolean presentationTimestampRequested;

    public ResultGenerator(QuerySpec querySpec) {
        this.querySpec = querySpec;
        presentationTimestampRequested = querySpec.getIncludedRelations().stream()
                .anyMatch((relation) -> relation.toString()
                        .equals(RequestParameters.PRESENTATION_TIMESTAMP.toString()));
    }

    public abstract Collection<String> getAllResourceFields();

    public Collection<String> generateMissingFields(Collection<String> currentFields) {
        return Collections.emptyList();
    }

    public Set<String> getRequestedFields() {
        if (requestedFields == null) {
            requestedFields = generateFields();
        }

        return requestedFields;
    }

    private Set<String> generateFields() {
        Set<String> fields = new HashSet<>();

        if (querySpec.getIncludedFields().isEmpty()) {
            fields = new HashSet<>(getAllResourceFields());
        } else {
            for (IncludeFieldSpec field : querySpec.getIncludedFields()) {
                fields.add(field.toString());
            }

            FilterSpecBaseIterator it =
                    new FilterSpecBaseIterator(querySpec.getFilters());
            it.stream().map(QuerySpecUtils::lastPathComponent).forEach(fields::add);

            fields.addAll(generateMissingFields(fields));
        }

        return Collections.unmodifiableSet(fields);
    }

    public abstract List<ResourceT> extractResources(ResultBearing jdbiResultBearing);

    public List<ResourceAndCounter<ResourceT>> extractResourcesWithCounter(ResultBearing jdbiResultBearing) {
        throw new UnsupportedOperationException(getClass().getSimpleName() + " does not support counter generation with the resources.");
    }
}
```

The code is quite straightforward and shows the capabilities of a result generator. The abstract method `getAllResourceFields()` returns all the fields for the given resource. To generate the subset needed, `getRequestedFields()` is called, which uses the query specification to generate a list of necessary fields — not only the ones specifically requested, but also the ones necessary to filter. Besides, `generateMissingFields()` allows subclasses to specify additional mandatory fields. This is necessary, for example, if a field needs another to be calculated, but only the first one is requested by the user.

The last abstract method left, `extractResources()`, acts upon the query results to generate a list of resources. Its twin `extractResourcesWithCounter()` is very similar, but acts on a query that has a column with the total row count. A mapper just calls the `map` method in the result-bearing object, using itself as an argument, and then gets a list from the result, as the following example taken from the `SimpleMapper` class shows:

```java
    @Override
    public List<ResourceT> extractResources(ResultBearing jdbiResultBearing) {
        return jdbiResultBearing.map(this).list();
    }

    @Override
    public List<ResourceAndCounter<ResourceT>> extractResourcesWithCounter(ResultBearing jdbiResultBearing) {
        return jdbiResultBearing.map(new RowMapperWithCounter<>(this)).list();
    }
```

Note that a `RowMapperWithCounter` is a wrapper for a `RowMapper` that forwards to the wrapped mapper the reading of every column but for the row counting column.

For result generators that are not mappers, code can be more convoluted. The typical case is an accumulator, such as the one used by the run keys endpoint:

```java
public class RunKeysAccumulator extends ResultGenerator<RunKeys>
        implements ResultSetAccumulator<Map<Integer, RunKeys>> {
    public RunKeysAccumulator(QuerySpec querySpec) {
        super(querySpec);
    }

    @Override
    public Collection<String> getAllResourceFields() {
        return RunKeysMeta.ATTRIBUTES;
    }

    @Override
    public Collection<String> generateMissingFields(Collection<String> currentAttributes) {
        return RunKeysMeta.MANDATORY_ATTRIBUTES;
    }


    @Override
    public Map<Integer, RunKeys> apply(Map<Integer, RunKeys> accumulated, ResultSet rs,
    ↪  StatementContext ctx)
            throws SQLException {
        Integer runNumber = Converters.getInteger(rs, RunKeysMeta.ATTRIBUTE_RUN_NUMBER);
        if (!accumulated.containsKey(runNumber)) {
            accumulated.put(runNumber, new RunKeys());
        }
        RunKeys runKeys = accumulated.get(runNumber);

        // Add contents of result set to runKeys.
        // Avoid overwriting previous content.
        // ...

        return accumulated;
    }

    @Override
    public List<RunKeys> extractResources(ResultBearing jdbiResultBearing) {
        return new ArrayList<>(
                jdbiResultBearing.reduceResultSet(new HashMap<>(), this).values());
    }
}
```

Accumulators are a great example to show how a result generator subclass is more or less self-contained and is able to deal from all aspects of the mapping or reducing from a raw query result to a structured resource list.

The abstraction that the result generator class enables is not only useful to reuse the field-generation code, but it also allows the `DatabaseResourceRepository` class to be even more generic. While in the first iterations it was implemented exclusively for mappers, it takes now a result generator as a type parameter, as we have mentioned above. The repository implementation does not need to know anymore how to transform the results of the database, because the result generator takes care of this whole process.

The `QueryBuilderResourceRepository`, on the other hand, has not been adapted to use a generic result generator, that is, it only works with one-to-one mappers (extending `SimpleMapper`). We needed to provide a class that was, above all, easy to extend. Typically, endpoints needing custom row reduction do not use a query builder either, because the queries are too specific and ill-fitted for it, and we preferred to keep the most concrete resource repository to be used as the default.

2.4.4 Meta base class

The meta classes also shared a fair amount of code, which was also moved up in the class hierarchy. The common meta class, however, is less interesting than the result generators, because it is a single class that just creates some collections to reduce boilerplate on its subclasses.

2.5 CSV export

CSV is a well-liked format among physicists because of its simplicity and readability, and particularly because it can be imported into an Excel sheet. WbM had a lot of visitors that made heavy use of the “export to CSV” functionality, and OMS received dozens of requests to implement it. However, we wanted to provide a export feature that worked for old endpoints right out of the box.

The solution passed through a deeper understanding of the Katharsis request lifecycle, which is explained in detail in Section 2.2, from which we will reference Figure 2.1 often. Indeed, we add a method annotated with a JAX-RS `@Path` annotation. Katharsis, then, delegates this method to our JAX-RS implementation, Jersey.

The `@Path`-annotated method is added to a new interface, `CsvSerializableResourceRepository`, which extends the resource repository bundled by Katharsis and provides only this method, with a default implementation. The whole interface is shown here:

```java
public interface CsvSerializableResourceRepository<ResourceT, I extends Serializable>
        extends ResourceRepositoryV2<ResourceT, I> {

    @GET
    @Path("csv")
    @Produces("text/csv; charset=utf-8")
    public default String getCSV(@Context UriInfo info) {
        AggregationAPIService api = AggregationAPIService.getInstance();
        ResourceInformation resourceInformation = api.getResourceRegistry()
                .findEntry(getResourceClass()).getResourceInformation();
        QuerySpec querySpec = UriInfoDeserializer.querySpec(info, resourceInformation);

        ServiceUrlProvider serviceUrlProvider =
                api.getResourceRegistry().getServiceUrlProvider();
        if (serviceUrlProvider instanceof UriInfoServiceUrlProvider) {
            ((UriInfoServiceUrlProvider) serviceUrlProvider).onRequestStarted(info);
        }

        List<ResourceT> resources = findAll(querySpec);

        if (serviceUrlProvider instanceof UriInfoServiceUrlProvider) {
            ((UriInfoServiceUrlProvider) serviceUrlProvider).onRequestFinished();
        }

        CsvResourceSerializer serializer =
                new CsvResourceSerializer(resourceInformation);
        return serializer.serialize(resources, querySpec.getIncludedFields());
    }
}
```

Using an interface has a very big advantage here: simply implementing this interface from one of the already existing endpoint repositories will add a `/csv` subendpoint, which should produce a CSV file.

Besides, the code is relatively simple. First, we retrieve the `ResourceRegistry` of Katharsis, which is a class that stores information of the structure of every resource and repository. We pass it to the `UriInfoDeserializer`, which is a custom class that internally uses the methods of Katharsis to deserialize the URL parameters into a fully-fledged query specification.

Then, we have to manually notify Katharsis that we have started processing a request. Once it is aware, we can safely call the `findAll()` method that every repository provides to obtain a resource list.

The only thing that is left is serializing the resources as a CSV, which is managed by the custom `CsvResourceSerializer`. The serializer not only generates the CSV, but it also ensures that it is similar to the JSON output for the same query (for example, it only includes the fields requested by the user).

2.6 Scaling info

WbM had a tendency to ignore data units when displaying data. Units were usually hardcoded and just displayed alongside the values. However, this means that a big portion of the older data (which did not have standardized units) is not displayed correctly.

For OMS, we tried to address this problem with correctness as the principal priority. For that, we added a scaling table with units information for the most common magnitudes. Any value representing one of these magnitudes will need to reference the scaling object, and any other value with dynamic scaling will need to store the scale alongside the value. This system also allows for optional display scaling to be stored alongside the value: sometimes, we know when storing the data that the values are very small, and as such we would like to display numbers with smaller units to improve readability.

These changes in the database require, of course, changes in the API. Several query builder attributes were added, so each query builder can control the source of the scale for a given value. A new decorator applied to the attribute correctly scales the value on the database, and furthermore the applied scaling is returned as an additional column. This allows the mapper to build a `ScaledBigDecimal` object, which bundles a `BigDecimal` together with a scale value.

A general abstract class `UnitConverter` was also created, together with instances for each usual magnitude type. These converters are able to scale a `ScaledBigDecimal` taking into account specific magnitude considerations, and to generate the appropriate unit string for the given `ScaledBigDecimal` and magnitude. Each specific converter is customized, since we have to adapt to certain conventions in the field: for example, the preferred way of displaying certain units is using the closest SI prefix, but for other units using 10^{scale} with the base unit is more common. Another case where the converter deviates from standard behaviour is time units, since time is the only magnitude not using base 10. The `TimeUnitConverter` can adequately convert between seconds (scale 0), minutes (scale 1) and hours (scale 2), and use milliseconds, microseconds... for scales smaller than 0.

2.7 New endpoints

A good number of new endpoints were created as part of my internship. They greatly vary in complexity and difficulty. Some of them were easy to create, because they just read information from a single database table and could almost be created automatically from the table structure. However, others posed a bigger challenge, and even in some cases forced the development of one of the new features above.

We will analyze some of the most interesting endpoints that I have created.

2.7.1 Luminosity summaries

This was the first endpoint that was designed specifically to plot data. Support for the `group[size]` and `group[count]` query parameters (described in Subsection A.3.6) was added originally just because of this endpoint, although it has been then used in several other endpoints. The changes necessary to support these parameters mostly affected the deserialization process, and the query builder. To support the new parameters, we created a new attribute that added, for each row, a rank depending on the `group` parameter. This counter, then, can be used in the `GROUP BY` of an external `SELECT` to aggregate the rows.

The rank column is generated using window functions:

```sql
-- All this examples assume that the default sorting order is by columns a, b
-- For group[size]=n
TRUNC((ROW_NUMBER() OVER (ORDER BY a, b) - 1) / n) + 1

-- For group[count]=n
NTILE(n) OVER (ORDER BY a, b)
```

The end result is an endpoint that can aggregate its information dynamically. Of course, the most convenient for plotting is `group[count] = <reasonable number of data points>`. While at first sight this might seem useless (the API is still going through all rows on the database), it is useful to reduce the data load for both the API and the GUI, resulting on quicker JSON serialization in the API and deserialization in the GUI, and less time necessary to plot the points in the GUI.

However, there is a big problem that arises, again, from the fact that we still have to iterate over every row: the endpoint cannot be used to aggregate big data ranges. The base table for this endpoint has, roughly, three rows inserted per minute during normal data taking. This means that aggregating over a range of more than two weeks starts to be slow, even when the table is well indexed and the query plan is reasonable. For this reason, it was decided to create a table aggregating these data per hour. The aggregation is performed on the database, and the user of the endpoint can choose to access the highly granular data (`group[granularity]=lumisection`) or the per-hour data (`group[granularity]=hour`). The queries generated are, then, very similar, and the `group[size]` and `group[count]` parameters can be used normally with both options. The hourly version of the data makes accessing them much faster, with date ranges of over a year being served almost instantly.

2.7.2 Dip logger

The DIP logger is the most complex endpoint that we have built to date. It deviates from all the standard features and behaviour that we expect from all the other endpoints, because its purpose is also very particular.

DIP is an information distribution service built by and for CERN to distribute experiment data [6]. DIP uses a publisher-subscriber architecture, in which clients must subscribe to the DIP endpoint(s) that they are interested in, and they will receive information every time that a new entry is available. This means that the DIP publisher itself only stores the latest entry, and there is no way for services to request old data, since it is not stored anywhere. The reasoning behind this is that DIP exposes a lot of data sources with very frequent updates, and trying to store all the data would quickly become unmanageable.

However, certain specific applications need to access DIP historical data for different reasons. Typically, this is done by a DIP subscriber that writes every entry somewhere, and indeed that is how WbM did it. WbM has been storing this information almost since the inception of CMS, and every necessary DIP endpoint has its own table where the data are logged. The so-called "DIP Logger" was built as a highly dynamic system, to adapt to the frequent changes in DIP endpoints and structure of returned data and to allow to start logging quickly new endpoints that were not considered interesting beforehand. However, the Aggregation API is envisioned as a static resource provider with a stable interface to access data. In particular, endpoints should never change their response structure without a very good reason, presenting always the same set of attributes, that is, the same interface.

The challenge, then, was to create an endpoint that keeps this static facade while allowing for all the dynamism that the DIP tables are designed for. Furthermore, we could not redesign the old tables, since this is out of scope for OMS.

Our approach was to introduce subendpoints. This means that the DIP data can be accessed via `/diplogger/path/to/dip/subendpoint`, and this subendpoint will work exactly like a real endpoint. However, the attribute list, the types... cannot be hardcoded, because we have around 500 DIP endpoints and it would become completely unmaintainable. What we do, then, is to take advantage of the metadata tables that the DIP Logger offers, which include a mapping from DIP endpoints to database tables and information about every column on these tables and their type. The code for our DIP endpoint does two database queries: one to discover what table to access and what attributes are filterable, sortable... and should be returned, and a second query that retrieves the actual data. This is even more complicated because every DIP subendpoint allows using `group[size]` and `group[count]` as long as none of the attributes requested are arrays.

From the technical point of view, we faced two big challenges. The first one was to actually create a dynamic endpoint, because Katharsis forces endpoints to be static. The solution was to modify slightly the code of Katharsis, just to support some JSON-related annotations (included on the Jackson library). In particular, support for the `@JsonUnwrapped` annotation was added. It "unwraps" one of the attributes, stored as a map, so each value appears as a top-level attribute, instead of just being a value inside a JSON dictionary.

The second problem is that Katharsis does not support dynamic endpoints, but DIP "subendpoints" are dynamic in nature. However, Katharsis does support standard subendpoints that are handled manually, via the JAX-RS `@Path` annotation. Again, a very small code modification was needed to enable support of path annotations containing a regex path specification. Since JAX-RS supports this, it was only a matter of changing the strict rules that Katharsis puts in place deciding what is a valid path. Once the path is correctly recognized, it can be translated into a filter and the rest of the endpoint can behave as a normal endpoint, but for the exceptions described above.

Finally, this was a very big deviation of the standard endpoint architecture. Some of the benefits of the base classes, the query builder... could still be used, but the dynamism forced a very particular endpoint structure, which posed its own challenge.

2.8 Rewriting old endpoints

Almost all endpoints have been rewritten during my internship to take advantage of the new functionality (particularly the query builder and the base classes). For a lot of them, the conversion is straightforward and of little interest, but we will list here the ones that were more tricky or challenging. We will avoid mentioning some of them here because the discussion is related with the new tables that will be introduced in Chapter 3.

2.8.1 Deadtimes per scaler

The deadtimes per scaler endpoint is interesting because it is a perfect candidate to use some database functionality that is not used often: the `PIVOT` and `UNPIVOT` features.

This enpdoint shows data from a view that has one row per run number. For every row, it has the average, minimum, maximum and root mean square of 20 different deadtime categories (called "scalers"). This means that the view has more than 80 columns. However, the data that we want the endpoint to show should be keyed by run number and scaler, and then have 4 attributes: average, minimum, maximum and root mean square (RMS).

To make it easier for the API, we decided to create a database view on top of the previously mentioned view. This uses the `UNPIVOT` feature to achieve the intended effect:

```
CREATE VIEW cms_oms_agg.run_scaler_deadtime_v AS
SELECT *
```

```
FROM CMS_OMS_AGG.oms_run_deadtime_v
UNPIVOT INCLUDE NULLS
((avg_dt, max_dt, min_dt, rms_dt) FOR scaler_name IN (
    (avg_dt_apve, max_dt_apve, min_dt_apve, rms_dt_apve) AS 'APVE',
    (avg_dt_bx_mask, max_dt_bx_mask, min_dt_bx_mask, rms_dt_bx_mask) AS
    ↪  'BX_MASK',
    (avg_dt_calib, max_dt_calib, min_dt_calib, rms_dt_calib) AS 'CALIB',
    -- ... Other scalers ...
    )
);
```

2.8.2 Fill bunches

The fill bunches endpoint is interesting for the implementation of the calculation of one specific attribute, and also for the scaling of that attribute. We have a core table with the information about each bunch crossing, as we will discuss in Subsection 3.4.2, and it contains all the data to be displayed. However, some of it needs to be calculated from the values in the table.

The fill bunches endpoint retrieves information about bunch crossings, that is, the possible moments where the two beams may collide for a given revolution. One of the values that are retrieved is pileup, which measures the number of collisions per bunch [22]. Nowadays, measured pileup normally oscillates between 30 and 50.

However, pileup cannot be measured directly without reconstructing the tracks and analyzing the collision. The pileup that OMS displays is estimated from the instantaneous luminosity, using an additional parameter called cross-section that depends on collision type and energy and is estimated for every fill. For that, the following calculation is used:

$$P = \frac{L_{inst} \cdot \sigma}{f_{Hz}},$$

where P is the pileup, L_{inst} is the instantaneous luminosity for the given bunch crossing, σ is the cross-section and f_{Hz} is the frequency of the crossings, which is $11,246$ Hz. This calculation would be easy to implement in the SQL query, but the cross-section is not available anywhere on the database.

The optimal solution would be to get the cross-section written to the database, but the responsible team could not do it at the moment. The alternative solution was to reverse-engineer the value from our lumisection table, which contains both the average pileup per lumisection and the instantaneous luminosity. The following view, then, extracts the cross-section from the lumisection data:

```
CREATE VIEW cms_oms_agg.fill_cross_section_v AS
SELECT
  f.fill_number AS fill_number,
  AVG(l.avg_pileup * f.n_colliding_bunches * 11246 / NULLIF(l.instantaneous_lumi,
  ↪  0)) AS cross_section,
  -MAX(s.inst_lumi_scale_stored) AS scale_cm_2
FROM cms_oms.fills f
  JOIN cms_oms.lumisections l ON f.fill_number = l.fill_number
  JOIN cms_oms.scaling_info s ON l.scale_id = s.scale_id
WHERE l.physics_declared = 1 AND l.avg_pileup > 0
GROUP BY f.fill_number;
```

For a given fill, all lumisections with non-zero instantaneous luminosity are considered in order to calculate the cross-section. The result is then averaged to clean up possible inconsistencies caused by calculation precision, although the number should be constant for all lumisections. The cross-section is calculated as follows:

$$\sigma = \frac{P_{avg} \cdot n_{\text{colliding bunches}} \cdot f_{Hz}}{L_{inst}},$$

where the only new variable is $n_{\text{colliding bunches}}$, the number of colliding bunches, which we store in our core tables too.

Our testing revealed that the previous view was outputting reasonable cross-section values that matched with very good precision the values published for example fills. Thus, we decided that, until we get this information written to the database, this is our only option.

However, another problem remained unsolved: scaling. Pileup is a dimensionless quantity, but the parameters that it is calculated from do have dimensions, and we have to ensure that they match for the calculation to work. Specifically, the cross section has area units, while the instantaneus luminosity has the dimensions of events per time and per area. It is easy to see that dimensions cancel out on the pileup cancellation, but, of course, data has to be appropriately scaled for that to happen. Luckily, we have scaling data for both the lumisection table and the bunches table. This means that we can add the `scale_cm_2` column to the view above, and use that one to calculate the correct scale for the pileup. In the end, the query for the endpoint is the following:

```sql
SELECT
      b.fill_number AS fill_number,
      b.peak_lumi * cs.cross_section * POWER(10, cs.scale_cm_2 +
       ↪  scaling.inst_lumi_scale_stored) / 11246 AS pileup,
      ...
FROM cms_oms.fill_bunches b
     LEFT JOIN cms_oms.scaling_info scaling ON b.scale_id = scaling.scale_id
     LEFT JOIN cms_oms_agg.fill_cross_section_v cs ON b.fill_number =
      ↪  cs.fill_number
WHERE b.fill_number = :fill_number;
```

The units are adjusted in the calculation to ensure that the pileup value is correct. It is important here that the stored values are multiplied first, and then scaling is applied as once, because precision errors would be way bigger if we scaled first and then multiplied.

3. Database redesign

3.1 Motivation

At the start of my studentship, the database schemas used for the aggregation API were the old WbM schemas. However, a big portion of the data that these schemas are populated with were provided by a hardware device called SCAL, which injected data from the CMS data acquisition stream into the online database. However, SCAL was decommissioned at the end of Run 2, which means that the whole process of populating the online database with data had to be redesigned.

On the other hand, the WbM schema was old and had the same problems that afflicted WbM itself: it was designed without a general vision on mind, which led to a suboptimal table and column structure. Thus, it was decided that replacing SCAL by other data sources was a good opportunity to rewrite the whole database schema.

3.2 Necessary domain knowledge

The replaced tables contained information about the behavior of the LHC and the CMS detector itself, before, during, and after data-taking. A few concepts related to the LHC and CMS are key to understand what do these tables store. The information in this section is based on [19, 10, 3].

The accelerator tunnel hosts two beams made of particles that circulate in opposite directions and collide at some points along the circumference. CMS sits on one of these points. This means that all the information stored on OMS tables can be broadly classified on one of two groups: LHC-level information or CMS-level information. Scientists analyzing the data need to access some of this information to be able to reach conclusions about the full system.

When fully accelerated, a beam completes about $11,246$ revolutions of the LHC tunnel per second. However, a beam is not a continuous flow of particles, nor a single packet of them: the beams are made of small packets, called bunches. This setup is necessary because particles can only be accelerated when the electric field has the correct orientation, and the gaps allow the orientation to be flipped. There are at most 2808 of these bunches. Each bunch is a few centimeters long and a millimeter wide when they are far from a collision point. However, when they approach the collision points, they are squeezed to about 20 micrometers. At its fullest capacity, the LHC uses a bunch spacing of 25ns ($\sim$7.5m). The bunches are not evenly spaced, as for practical reasons there are several bigger gaps in the pattern.

One of the most important properties of the beam is its luminosity. Luminosity is a magnitude which is directly proportional to the number of particle collisions that we can expect if we collide the beams. The highest the luminosity, the more collisions we will observe. Moreover, we can talk of instantaneous luminosity, when we are concerned about the luminosity at a particular moment of time, or integrated luminosity, when we are interested in the luminosity for a time interval. Note that this magnitude is completely different from the beam energy, which is of course proportional to the beam speed and determines the kind of particles that could be produced on the collision. Luminosity only is relevant to measure how many collisions will be observed.

A fill names a period of time on which the LHC is filled with the same beam. That is, particles are injected into the LHC tunnel at the beginning of a fill and are not replaced during the fill. The tunnel is emptied at some point in the middle of the fill, but the fill is not considered to end until the start of the next fill, that is, until particles are injected again. The process of emptying the tunnel is called beam dump and can be triggered manually or automatically. Normally, a manual beam dump means that the beam has been in there for long enough that its luminosity has degraded and a new beam is necessary. If it is automatic, that means that something has gone wrong and the beam has unexpected behavior, and it is dumped safely to make sure it does not constitute a hazard for anything or anybody. The total energy in each beam at maximum energy is 350 MJ, which is about as energetic as a 400 t train traveling at 150 km/h. Thus, an uncontrolled beam loss would be a disaster, but safety systems are in place to detect a problem and safely dump the beam within 3 revolutions (less than 0.3 ms).

A fill, then, describes a time period between the moment when the LHC is filled with a new beam and the moment when it is dumped (although it technically does not end until later). While most LHC fills collide protons with protons, there are also fills that collide heavy ions (generally lead), and even fills colliding protons with heavy ions. This information is encoded on the runtime type of the fill, and we also store information about the particle type for each beam (which is either protons or the concrete type of ion). Swapping the particles is a costly process time-wise, and thus the runtime type usually remains fixed for weeks or even months. Therefore, it is useful for CMS to group similar fills into so-called eras, which are periods with the same high-level detector configuration, including the collided particle type.

We are interested in other fill properties too. A fill is said to have reached stable beams if the beam has had the adequate properties for data taking, i.e., the beam is adequately focused and correctly aligned to be collided. Only when stable beams are declared do the detectors start collecting data. Of course, this means that, most of the time, only fills with stable beams are interesting.

A run is a period of time which designs CMS data-taking. Normally, a fill encompasses several runs. All those runs share, of course, the same beam, but they might differ on detector-specific low-level configuration settings. Again, the runs of real interest are the ones occurring during stable beams. However, there are some special runs called cosmic runs. These runs happen after the beam has been dumped, so there are no collisions happening. However, CMS is on and trying to measure the cosmic rays that go through the detector all the time. These runs are useful for studying the performance of CMS and its sub-detectors and for alignment and calibration tasks.

Every run is, on turn, divided into lumisections (not to be confused with luminosity). Lumisections are periods of time of a fixed length, slightly over 23 seconds. This length is not arbitrary: it is the number of seconds that the particle beam needs to orbit the accelerator $2^{18} = 262144$ times. Lumisections are the smallest units that OMS displays data on, and they are defined to be short enough that the luminosity can be considered constant inside a lumisection. Every lumisection is identified by the run number for the run it happens on, and by another number which indicates its position inside the run (starting by 1).

3.3 Conceptual design

We can encapsulate the previous domain knowledge into an entity-relationship diagram, which would be the base to create our new database. The resulting diagram is shown in Figure 3.1.

It is important to note that our table design was not derived from this diagram, which has been

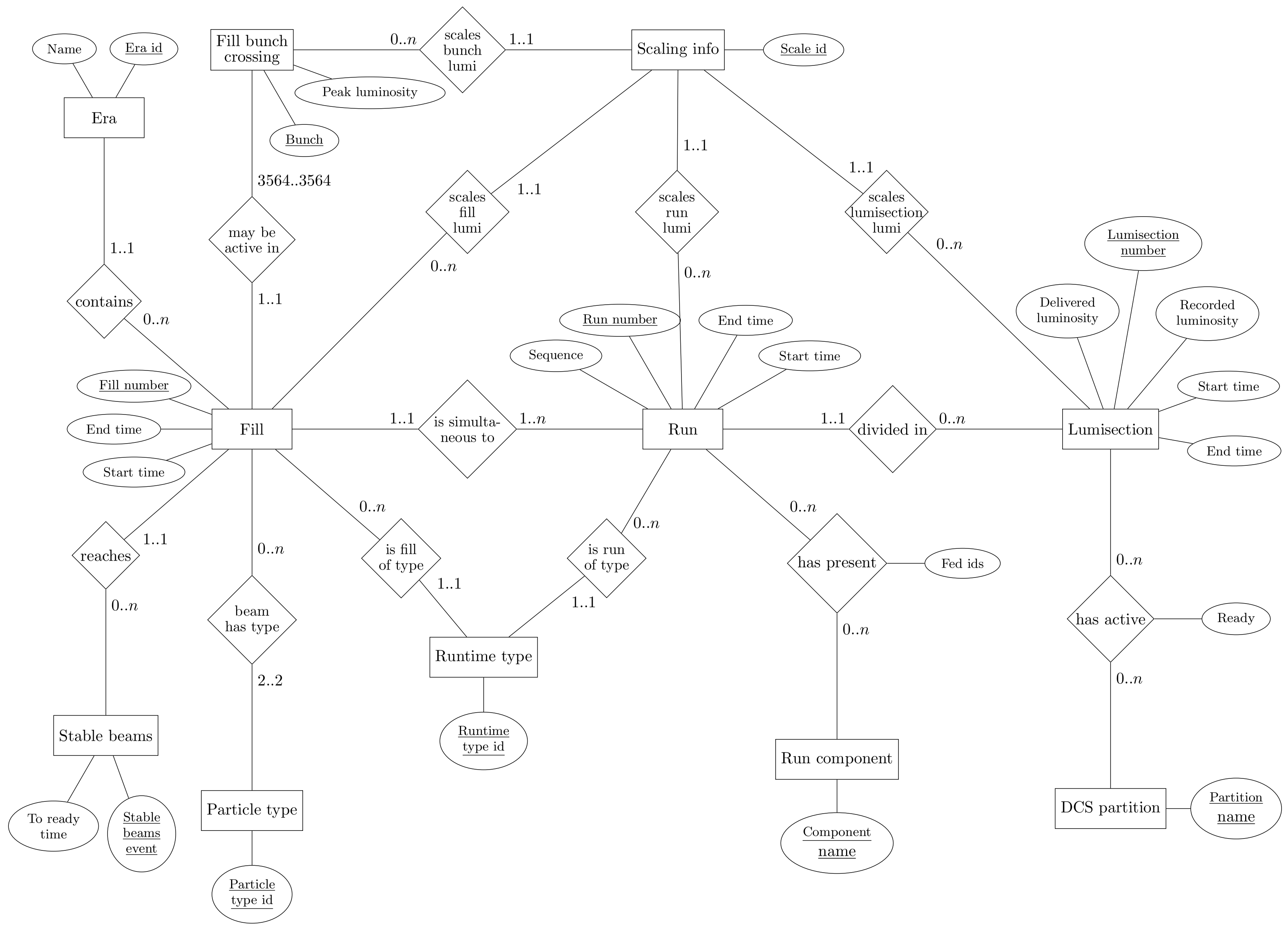


Figure 3.1: Entity-relationship diagram of the redesigned database. Note that only the most relevant attributes are included. (Source: own elaboration)

created after the fact to summarize and clarify the decisions made.

There are some entities shown in the diagram that have not been discussed in Section 3.2. The most important of those is probably the scaling, an entity which describes the scale of the luminosity information. Luminosity is an attribute common to most core tables (because we need to store different kinds of luminosity per fill, run, lumisection...), but it can come from different data sources and have different units, even inside the same table. Furthermore, the order of magnitude of the values can vary greatly, and therefore adjusting the value before storing it could lead to precision errors. Lastly, we also want to store the recommended display scale, which might be different for different runs. For all these reasons, we consider the luminosity scaling to be its own entity, and every row with a luminosity value is also linked to an scaling object in some way. Of course, this setup enables the automatic retrieval and manipulation done on the API side, as we already described in Section 2.6.

There are two other entities missing in Section 3.2: run components and partitions. They are both similar, with a slight difference in granularity. A run component stores the state of a subdetector present in a run, while a partition stores similar information but for a partition, that is, a part of a subdetector. The main difference between them, other than granularity, is that subdetectors cannot be added or removed in the middle of a run, while a partition can be turned on or off at different points in time without affecting the rest of the detector and without forcing a change of run.

Lastly, the number of bunches in a fill shown in Figure 3.1 (3564) seems to contradict the information from Section 3.2 (2808). However, this is not the case. The bunch entity represents a bunch crossing, that is, a moment of time where bunches might cross at the CMS detector depending on the configuration of the LHC. There are 3564 such moments for a single revolution of the beams. All of those are valid entities that might have relevant information, because even points where no bunches cross may register residual luminosity or beam intensity, which is interesting to store. Furthermore, this information can reveal errors, if for example luminosity is measured at a crossing when no beam is expected. For this reason, even if there are (at most) 2808 active bunches per beam and fill, we will always store all 3564 bunch crossings.

3.4 Logical design

We have divided the process in four different parts, broadly grouping the tables by topic. The final schema is shown in Figure 3.2. This section explains the choices made to transform the conceptual design into a logical design.

3.4.1 Standalone tables (eras, scaling...)

This group encompasses the tables that do not have foreign keys to other tables. It includes all the tables that do not fit on the other groups: eras, particle types, runtime types, and scaling info.

The tables in this section had a small number of attributes and little room for changes. Furthermore, these tables have a “supporting” role and the attributes will probably only be queried by their identifier, which simplifies the integrity constraints. On top of that, indexes on them are almost useless, since all these tables will be very small, most likely less than 100 rows.

The big design choice we had to face at this point was whether we should have a separate table for all these concepts (eras, runtime types and particle types) or we should just save each one on its corresponding column. All three entities already have an official identifier and they do not have very important attributes. Adding just the identifier to the other tables could improve performance and also simplify the queries. However, we finally decided that each entity should have a table. We

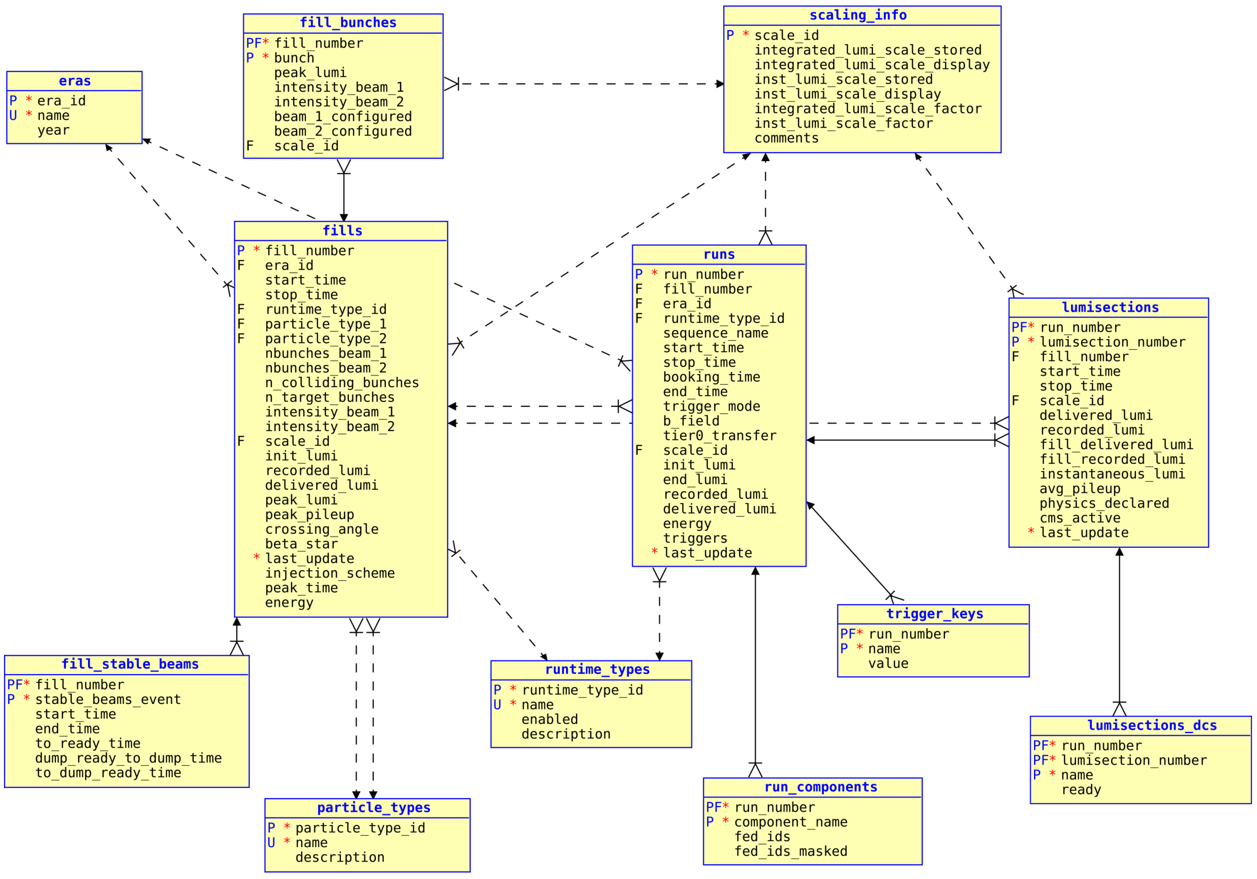


Figure 3.2: Snapshot of the final schema. Members of the primary key are represented with P, unique fields with U, and members of a foreign key with F. (Source: own elaboration)

wanted to enforce foreign key checking and reduce the probability of data inconsistencies. Besides, this allowed us to add useful attributes, such as the description attribute for the particle types.

The scaling info table was the biggest deviation from the previous design. This table stores the units on which luminosity information is stored and the units on which it should be displayed. On the old tables, this information was saved per year and runtime type. However, this was not flexible at all, and the WbM team had frequent requests asking from scaling changes mid-era, which were of course impossible. At that point, it made much more sense to have an identifier for each scaling configuration and store them on a separate table. Each table that stores luminosity values would then have a reference to one scaling object.

Other argument supporting the decision was aggregation. The new schema was designed to be queried by the Aggregation API, but the old structure was not prepared for running queries grouping big data ranges. With the old structure, it was complicated to join the luminosity info with its scaling, which is essential to aggregate data correctly. This functionality is needed all throughout the API, for example to create the lumisummaries endpoint described in Subsection 2.7.1.

3.4.2 Fill-related tables

This section includes the `fills` table and the `fill_stable_beams` table, which depends on it. These tables store all the information relative to LHC fills. It also includes the `fill_bunches` table, which stores one entry per bunch and per fill.

The fills table needs almost no explanation, as it is mostly a table for storing all the information we need. The columns are similar to those of the old fills table, but for a handful of columns that were not being used anymore and thus removed. This table now stores references to all the standalone tables described previously.

The main difference with the old system is the separate fill_stable_beams table, which stores the stable beams events that happen during a fill. Previously, the fill table contained a time for the start and the end of stable beams. In general, stable beams are declared only once during the fill, so this model was acceptable. However, on the new design we want to support the 3% of the fills that have several stable beam events. There are also a handful of timestamps that depend on when the stable beam was declared, that were previously attributes of a fill but are now columns of this table.

The fill_bunches table had no equivalent table in WbM, since information was taken directly from the source. However, we knew what attributes would need to be displayed, and therefore creating the table was straightforward. Here, the scaling that we store is different from the scaling for the given fill. It scales the peak_lumi attribute, and this is normally two or three orders of magnitude smaller than the init_lumi for the same fill, so we need different criteria.

3.4.3 Run-related tables

This category includes the runs table, as well as the run_components table and the trigger_keys table.

The design rationale is very close to that of the fill tables. The main table, runs, just stores columns with the information we need, taken almost verbatim from the old table. However, we have decided to migrate some of the information that was previously stored on columns to new tables.

It is worth mentioning that the runs table stores a reference to both the era and the runtime type. However, the references were added for entirely different reasons. The era is only stored in this table for performance reasons, especially when filtering on it, since it could be retrieved easily from the fills table. This is why there is no corresponding relationship in the entity-relationship diagram. However, a run has its own runtime type, which may be different from the type of the fill the run is in. This quirk of the design of the detector is the reason that the run and the runtime type are also related in Figure 3.1. Fills normally finish with some cosmic runs where the beam is off, but the detector is still on for calibration purposes. In this case, the run has runtime type "cosmic", but the fill has still the runtime type of the particles that made up the beam in previous runs.

There is one more table that does not appear in the original entity-relationship model in Figure 3.1: the trigger_keys table. It is a key-value style table that stores trigger keys: identifiers for the different configurations of the trigger systems on a run. Trigger keys are conceptually a part of the run, they are not an entity on their own. However, although they are mostly static, trigger keys are subject to change from time to time. A key-value storage provides increased flexibility, which we value over the possible performance degradation. Without this approach, tables would have to be changed every time that we have a new trigger key, and we would also need to keep columns storing old trigger keys that are not used anymore.

Finally, the run_components table contains information about the status of the different CMS subsystems during each run. This is information that was previously available on an external schema as a key-value system which was not easily queried. We plan to port all this information to our new tables. It is important to note that this table is not an entity-derived table, but a relationship-derived table; it stores (run_number, component) tuples. The relationship represents whether a component is included on a run.

Attributes are interesting in this case: we decided to store the fed identifiers, a list of data streams that are active during a run, as nested tables, using the `fed_ids` and `masked_fed_ids` attributes. Feds are integer sets, and we are normally interested on looking for runs where specific feeds are available or masked. The Oracle-specific nested table model offers decent performance for this task with the only drawback of a slightly more complicated syntax for inserting and querying. Again, this fed information was previously stored on external tables as a format very difficult to query (a CLOB of a CSV-like string), so any improvement is welcome. However, we did not want to manually maintain two extra tables for this purpose.

3.4.4 Lumisection-related tables

This category includes the `lumisections` and `lumisections_dcs` tables.

The `lumisections` table, just as the `runs` and `fills` table, contains just the necessary columns retrieved from the old lumisections table. The only difference with the conceptual design is that this table has a reference to the `fill_number`. As with the era identifier in the `runs` table, this reference does not appear in the conceptual design because it is a redundant reference which is in place just for speed when querying.

The main difference with the old table is that it contained one column per partition that specified whether it was ready or not. This design was not easy to scale, since partitions change from time to time. Therefore, it has been replaced by the `lumisections_dcs` table, which stores a row per lumisection and partition. It is named after the Detector Control System, which provides this information.

This is a relationship table, very similar to the `run_components` table. Here, we have an extra attribute, `ready`. A row is present in the table if the partition was present, but the `ready` column marks whether it was turned on. Therefore, it is easy to differentiate a partition that did not exist at the time from a partition that was off for the lumisection. Besides, this structure facilitates easy insertion and deletion of partitions, and allows old partition data to be stored, even if the partitions do not exist anymore.

We also decided to store the full partition name on this table, instead of having an additional table with partition identifiers and saving only those in the `lumisections_dcs` table. This is a difficult decision to make, since the approach with the additional table has a very clear advantage: identifiers are much more condensed and, thus, much less information is stored. However, it has two significant drawbacks. First, every time a new partition is added, the table has to be manually edited to add the new identifier. Second, having an additional table would make both querying the `lumisections_dcs` table and adding more rows to it more cumbersome.

The final reason that swayed our opinion was that partition names, while they are obviously less efficient in space than an identifier, are quite short, and therefore they should not be using too much storage. To validate our decision, we calculated the size of the table up until now, which contains the data of all runs since 2010. Right now, the LHC is expected to run for other 20 years, what would triple the space required for our table. Considering its current size, that is perfectly acceptable for us.

3.5 Constraints and indexes (DDL)

3.5.1 Primary keys

As a rule, we have opted for natural primary keys instead of surrogate primary keys. The reasoning is simple: most of our tables have an obvious natural identifier, which is unique by definition (the fill number, the run number...) In fact, the fill number or the run number are actually surrogate keys from a certain point of view: they are just generated as an increasing sequence of identifiers. The only difference with traditional surrogate keys is that they are generated from an external system. Not only there is no need to add an extra column when we have such good candidate keys, but also a lot of external tables, that are out of our control, use these values already as identifiers. Since the Aggregation API needs to join those tables often, it is convenient to keep the same standards as the rest of the database.

Choosing surrogate keys would have other negative consequences. The most impactful would be maintenance of an extra index, since the candidate key would have to be indexed in any case, for joining and because, for the core tables, filtering on the identifier is very frequent.

Besides, there are some of the typical drawbacks of natural keys that do not apply in our case, at least for the core tables (fills, runs and lumisections). Surrogate keys are often praised by their performance, since the key value is smaller and thus accessing the index requires less disk IO. However, both fill and run numbers are consecutive, which means that these columns should not be bigger than a surrogate key column. Besides, surrogate keys are often deemed more future-proof than natural keys, because they are safe from changes on business meaning. However, the whole detector is built using these identifiers. If they were to change, our whole software (and that of many other teams) would need to be rewritten anyway.

The lumisections table has a multi-column primary key, and it is here where a strong argument for a surrogate key could be made. However, the standard throughout our database is to use both these columns (run number and lumisection) as our identifier. It would not make sense to have a lumisection identifier including both that could only be used internally, if then we have to join external tables on several columns.

3.5.2 Foreign keys

From the beginning, we decided to enforce foreign keys as part of our system. WbM did not use foreign keys at all, which made its data much more difficult to understand and less efficient to query. WbM was designed in this way because their data sources were not completely reliable, and it was a way to avoid costly errors when inserting new data. However, we decided that the improvement in data consistency was worth making the data insertion process more delicate. Data dependencies must now be watched carefully during the insertion process, and the delays of the different data streams has to be taken into account. However, we believe this to be an opportunity to treat carefully the data from their insertion, instead of logging everything and fix the errors afterwards, which was the philosophy of the WbM team.

3.5.3 Other constraints

Check constraints are used to maintain data integrity where there is only a limited number of options, especially for Boolean or enumeration-like columns.

An example of this is the `physics_declared` column on the `lumisections` table. There, the values are restricted via a simple constraint:

```sql
CONSTRAINT lumisections_physics_dec_bool CHECK (physics_declared IN (0, 1));
```

3.5.4 Indexes

One of the usual concerns when designing indexes is space. However, space is not a problem for us in general, although of course we will try to minimize it if possible. Insert and update performance is, however, a bigger concern.

Fortunately, most of our core tables are not written to very often, and updated very rarely. The runs and the fills table will have at most twenty or thirty new rows during a busy day, which means that costly inserts are not very problematic. Even then, we have tried to keep the number of indexes to a minimum, although in the end we want to support several types of query. Specific indexes and their effect on queries will be discussed in Section 3.7, where some of the queries using the new schema are discussed.

For example, the indexes for the run table are the following:

```sql
CREATE INDEX cms_oms.runs_fk_fill_number
ON cms_oms.runs(fill_number);

CREATE BITMAP INDEX cms_oms.runs_fk_era_id_idx
ON cms_oms.runs(era_id);

CREATE INDEX cms_oms.runs_sequence_name_idx
ON cms_oms.runs(sequence_name);

CREATE INDEX cms_oms.runs_start_time_idx
ON cms_oms.runs(start_time);

CREATE INDEX cms_oms.runs_stop_time_idx
ON cms_oms.runs(stop_time);
```

The whole rationale behind the choice of columns to index is detailed in Subsection 3.7.3, so we will not discuss it in depth here.

The lumisection-related tables are the ones with the highest new-data frequency, with one new lumisection every 23 seconds when the system is on, and bulk updates that can affect all lumisections during the last 48 hours. However, we do not need lots of indexes here, and it is not a problem to have only three indexes.

In any case, we have the advantage that data should always be queried from the Aggregation API. Indexes are significantly easier to design with a reduced list of possible queries. Thus, the index design process has been relatively easy and similar for all tables.

3.6 Data migration (DML)

Data migration also proceeded following the same divisions by topic. For the most part, this was a straightforward process, with `INSERT` statements that just insert the relevant columns from the old tables into the new tables. This section will describe the tables which differed significantly from their WbM analogues, and thus needed some data manipulation to be populated.

It is important to note that the general rule to follow for this process was not to change old data unless there is no other option. Even with data that are likely wrong, the policy of the OMS group has been not modifying it unless it causes obvious problems. While, of course, right data are preferred, we operate under two assumptions:

1. We do not have critical data. OMS and its predecessor WbM are used by CMS physicists and engineers to discuss what happens in the detector and the LHC, but they are not used for scientific analysis or similar purposes.

2. All our data are either correct or irrelevant. These data has been shown for years in WbM, and it has been used for multiple purposes, shown in meetings, etc. Users do often report

some data that are wrong or missing and the problem is quickly fixed. Thus, it is very unlikely for wrong data to survive for long.

The decision to keep wrong data is consequence of the usage patterns of the API. We serve data to many people, who use it for their analyses, their meetings, their notes... Consistency is expected from them, and any change should be approached carefully and on a case-by-case basis. In particular, we should not change values such as timestamps even if we have access to a more precise data source that the one already stored.

3.6.1 Missing fills

The new fill table should be populated directly from the old fill table. However, we discovered that we have more than a hundred fills that were referenced from either the run or the lumisection table but were not present on the fill table. This was not a problem in the past, given that no foreign key constraints were enforced. However, the foreign key constraints of OMS prevented us from inserting a fill that was not present on the `fills` table.

The problem was not just discovering these fills, but also being able to fill their basic information (i.e., their start and stop time), given that they were not registered at all. The solution, for fills that had lumisections, was easy: just set the minimum and maximum time for all lumisections as the start and stop time of the fill.

However, this was slightly more difficult with runs, as the beginning and end of a run do not align with the beginning and the end of a fill. Lumisections did not cause a similar issue because they are small enough to be inside a tolerable margin of error.

3.6.2 Runs with several fills

Throughout this document, we have assumed that a run is somewhat “inside” of a fill: the runs table even has a `fill_number` column pointing to the fill for that run. However, a detailed analysis of the data revealed that this is not completely true. The process of declaring a new run is manual and at the CMS level, and thus unrelated to the process of declaring a new fill, which happens at the LHC level. This means that, even if in most cases a new run is declared once the new fill is on, this does not necessarily happen.

The saving grace was that this does never happen on fills with stable beams. When the beam is dumped, data taking stops and consequently the run is automatically ended. For this reason, we can assume that a run has one and only one fill number, even though we can find runs spanning several fills if looking at the timestamps. The `fill_number` column should be understood as the fill number for the fill *on which the run started*, but this consideration is fortunately irrelevant for fills with stable beams declared.

3.6.3 Fill stable beams

For old fills, only one of the stable beams events was saved on the table. Typically, stable beams are declared exactly once per fill, but as we already mentioned in Subsection 3.4.2, it sometimes happens twice. With the new design, we finally had the means to store this information, but we needed a way to obtain it in the first place. Luckily, we had access to another table, part of the DIP Logger system described in Subsection 2.7.2, to which a new row is logged every time the state of the beam changes. This was, then, our way of calculating the start and end time of each stable beams event, by analyzing the transitions of the beam state.

However, we did not want to change the old data. After careful consideration, we decided to keep the old timestamps for the old fills. However, the new procedure developed to fill this information will be adapted in the future to keep the tables up to date after the new run starts.

3.6.4 Scaling info

As this was one of the biggest paradigm changes in our design, migrating the data was not so straightforward. First, we analyzed the scaling on WbM tables, which was stored per year and runtime type, and a new row was created for the `scaling_info` table. It was a bigger problem to add the correct scale identifier to the fill, run and lumisection tables.

Populating the fill table was not a problem: we had the year and the runtime type, which means we only needed to find the scale info for each fill from the tables described above.

For the runs and the lumisections, we used a `MERGE` statement to assign to each row the scale identifier for the fill referenced. As both the runs and the lumisections table have a `fill_number` column, this was also easy to do.‘

3.7 Redesigning endpoints

In Chapter 2, we discussed the changes made to the API. Rewriting old endpoints took a big portion of my time, but in the end the whole API was consistent and used the new tables. While some of them were discussed in Section 2.8, we will describe here the ones that used the new tables and had some influence on their design. We will emphasize the solutions to the performance problems that we faced, both when we changed or added indexes and when we changed the queries themselves.

3.7.1 Performance concerns

Pagination

The queries generated by the new query builder are of course completely different from the queries that the endpoints were using previously. Early in the process, it started to be obvious that the new queries were performing poorly in respect to pagination. Several endpoints have specific fields that aggregate into a single value hundreds or thousands of rows of data stored in a secondary table. The optimizer of the Oracle database was not able to discover that some of those aggregations should not be computed, because the corresponding rows will never be shown anyway due to pagination.

As an example, consider the table `cms_oms.fills` (with column `fill_number`) and the view `cms_oms_agg.fill_peak_specific_lumi_v` (with columns `fill_number`, `peak_luminosity`). The view has one row per fill, but this row is the result of aggregating all the $3,564$ rows on the `fill_bunches` table to get the `peak_luminosity` value.

The query generated was the following:

```
SELECT i.*
FROM (
  SELECT
    f.fill_number AS fill_number,
    p.peak_specific_lumi AS peak_specific_luminosity,
    ROW_NUMBER() OVER (ORDER BY f.fill_number) AS row_counter__
  FROM cms_oms.fills f
    JOIN cms_oms_agg.fill_peak_specific_lumi_v p ON f.fill_number = p.fill_number
  WHERE f.fill_number > 2000) i
WHERE i.row_counter__ BETWEEN 1 AND 100;
```

Ideally, this query would only retrieve the first 100 fills with fill number greater than 2000 from the fills table and only join the view to those fills, thus avoiding the expensive aggregation for all the other fills. However, our database yields the query plan shown in Table 3.1.

Id	**Operation** (name)	**Rows**	**Bytes**	**Cost**	**Time**
0	Select statement	2290	89310	13465	00:02:42
1	View	2290	89310	13465	00:02:42
2	Window sort pushed rank	2290	68700	13465	00:02:42
3	Nested loops	2290	68700	13465	00:02:42
4	View (`fill_peak_specific_lumi_v`)	2290	59540	13465	00:02:42
5	Hash group by	2290	87020	13465	00:02:42
6	Hash join right outer	74791	2775K	13462	00:02:42
7	Table access full (`scaling_info`)	22	198	3	00:00:01
8	Table access by index rowid (`fill_bunches`)	74791	2118K	13459	00:02:42
9	Index skip scan (`fill_bunches_conf_idx`)	441K		6922	00:01:24
10	Index unique scan (`fills_pk`)	1	4	0	00:00:01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("I"."ROW_COUNTER__">=1 AND "I"."ROW_COUNTER__"<=100)
   2 - filter(ROW_NUMBER() OVER ( ORDER BY "F"."FILL_NUMBER")<=100)
   6 - access("B"."SCALE_ID"="BS"."SCALE_ID"(+))
   8 - filter("B"."INTENSITY_BEAM_1">0 AND "B"."INTENSITY_BEAM_2">0)
   9 - access("B"."FILL_NUMBER">2000 AND "B"."BEAM_1_CONFIGURED"=1 AND
   ↪   "B"."BEAM_2_CONFIGURED"=1)
       filter("B"."BEAM_1_CONFIGURED"=1 AND "B"."BEAM_2_CONFIGURED"=1)
  10 - access("F"."FILL_NUMBER"="P"."FILL_NUMBER")
```

Table 3.1: Explain plan for an inefficiently paginated query.

As we can see, the view is queried for all the 2, 729 rows on the fills table, instead of obtaining only the 100 rows that we have requested.

To solve this problem, we modified the query builder to optionally include certain attributes as a subquery in the `SELECT` clause instead of a `JOIN`. The replacement would not work on the general case, but, for our specific query, all the fills in the view are also in the table, and we know that rows are matched one-to-one. Therefore, we could make the replacement, which yielded the following query:

```sql
SELECT i.*
FROM (
  SELECT
    f.fill_number AS fill_number,
    (SELECT p.peak_specific_lumi
       FROM cms_oms_agg.fill_peak_specific_lumi_v p
       WHERE f.fill_number = p.fill_number)
    AS peak_specific_luminosity,
    ROW_NUMBER() OVER (ORDER BY f.fill_number) AS row_counter__
  FROM cms_oms.fills f
  WHERE f.fill_number > 2000) i
WHERE i.row_counter__ BETWEEN 1 AND 100;
```

with the explain plan output shown in Table 3.2. This is a much more reasonable plan, that only accesses the relevant rows in the view.

Id	Operation (name)	Rows	Bytes	Cost	Time
0	Select statement	4811	183K	8	00:00:01
1	Sort group by nosort	1	38	10	00:00:01
2	Hash join outer	33	1254	10	00:00:01
3	Table access by index rowid (`fill_bunches`)	33	957	6	00:00:01
4	Index range scan (`fill_bunches_conf_idx`)	193		3	00:00:01
5	Table access full (`scaling_info`)	22	198	3	00:00:01
6	View	4811	183K	8	00:00:01
7	Window sort pushed rank	4811	19244	8	00:00:01
8	Index fast full scan (`fills_pk`)	4811	19244	7	00:00:01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("B"."SCALE_ID"="BS"."SCALE_ID"(+))
   3 - filter("B"."INTENSITY_BEAM_1">0 AND "B"."INTENSITY_BEAM_2">0)
   4 - access("B"."FILL_NUMBER"=:B1 AND "B"."BEAM_1_CONFIGURED"=1 AND
              "B"."BEAM_2_CONFIGURED"=1)
   6 - filter("I"."ROW_COUNTER__">=1 AND "I"."ROW_COUNTER__"<=100)
   7 - filter(ROW_NUMBER() OVER ( ORDER BY "F"."FILL_NUMBER")<=100)
   8 - filter("F"."FILL_NUMBER">2000)
```

Table 3.2: Execution plan for the improved query.

The advantage of the new approach is clear: performance. However, there are also drawbacks. First, the generated query is significantly less clear for detail tables used by several attributes. It is also less clear if we filter on one of these attributes. Second, the new query generation adds a new layer of complexity of the query builder. Now for each detail attribute we need to choose if we want it as a `LEFT JOIN` or as a subquery. The performance boost is significant enough to ignore the drawbacks, but it is important to consider them.

Counting

When we paginate, we only return a fixed number of objects from the database, but we need to know the total amount of objects that we would return without pagination. This is important, for example, to generate links information to navigate to the previous, next, first or last page.

The approach that was encoded originally in the query builder was to generate two queries, one that returns the full list of objects with every attribute, and a different one which only queried the main table or tables and only returned the count. These queries were supposed to be executed one after the other. However, to improve performance, we decided to do it on a single query. This means that the query is more complicated, but we connect once to the database instead of connecting twice, avoiding the extra delay from the second connection.

To enable this feature, it was important to also change the base classes accordingly. Specifically, we needed to interrupt the mapping process to obtain the number of rows from one of the SQL rows, and then continue parsing that row normally. For this, as we mentioned in Subsection 2.4.3, we created a specific mapper that is composed with the endpoint-specific mapper. This means that the outer mapper only retrieves counter information (if present), and the endpoint-specific mapper does its job normally to retrieve the full details for each row. Then, those two results (the object and the number of rows) are composed in a pair-like object, which is then returned. The number of rows is then retrieved from one of these objects (all of them should have the same value), and the list stripped of the numbers is also retrieved.

When this option is enabled, a query such as the one discussed above would look like this:

```
SELECT i.*, cnt.total_row_count__
FROM (
  SELECT
    f.fill_number AS fill_number,
    (SELECT p.peak_specific_lumi
                FROM cms_oms_agg.fill_peak_specific_lumi_v p
                WHERE f.fill_number = p.fill_number)
        AS peak_specific_luminosity,
    ROW_NUMBER() OVER (ORDER BY f.fill_number) AS row_counter__
  FROM cms_oms.fills f
  WHERE f.fill_number > 2000) i
  CROSS JOIN
    (SELECT COUNT(*) AS total_row_count__
    FROM
      (SELECT f.fill_number AS fill_number
      FROM cms_oms.fills f
      WHERE f.fill_number > 2000)) cnt
WHERE i.row_counter__ BETWEEN 1 AND 100;
```

Of course, all the filtering is set up automatically in both queries, so none of this work has to be done manually.

A different option, using Oracle window functions, was also tried, but was discarded because the performance was significantly worse. The query was the following:

```
SELECT i.*, i.total_row_count__
FROM (
  SELECT
    f.fill_number AS fill_number,
    (SELECT p.peak_specific_lumi
                FROM cms_oms_agg.fill_peak_specific_lumi_v p
                WHERE f.fill_number = p.fill_number)
        AS peak_specific_luminosity,
    ROW_NUMBER() OVER (ORDER BY f.fill_number) AS row_counter__,
    COUNT(*) OVER (ORDER BY f.fill_number
                   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS total_row_count__
  FROM cms_oms.fills f
  WHERE f.fill_number > 2000) i
WHERE i.row_counter__ BETWEEN 1 AND 100;
```

This option is definitely simpler and, naively, it should generate a similar execution plan. However, the plan generated is very alike to the plan in Table 3.1, calculating every peak luminosity even if most of them will never be shown. This is disappointing, because this is the perfect use case for window functions to simplify the query, but performance is key for our purposes.

3.7.2 Fills

The fills endpoint is quite complex, as one of our core endpoints. It does not only aggregate information from our tables, but also from several external tables not under our control, which means that we cannot decide which columns are indexed.

Endpoint-wide changes were necessary to adapt it to the modern endpoint structure. The fills endpoint was one of the oldest and it was completely phased out. The endpoint code was greatly simplified by using the new base classes and the new features.

The new endpoint was affected by the performance problems described above, but after the changes the new endpoint is as performant as the old one. Careful analysis of the execution plans reveals that the query execution is quite close to being optimal.

About the similarities with the old endpoint, we can find that the queries share a common structure, as all these detail tables have been unaffected by the migration. Furthermore, the fills table itself is not so different from the old one. The main difference is that additional tables (such as the particle type tables) have been joined to provide data that were hardcoded before.

3.7.3 Runs

The runs endpoint is the biggest endpoint (judging by the number of attributes) available in OMS and it was the biggest challenge in terms of optimization. It aggregates data from more than 10 tables, and about half of them are not under our control. In addition, we do not have permissions to explain the execution plan for several of the views used, as we do not have permission to access the underlying tables. This means that, for the parts of the query containing those views, it was very difficult to debug the performance problems without relying on intuition almost exclusively.

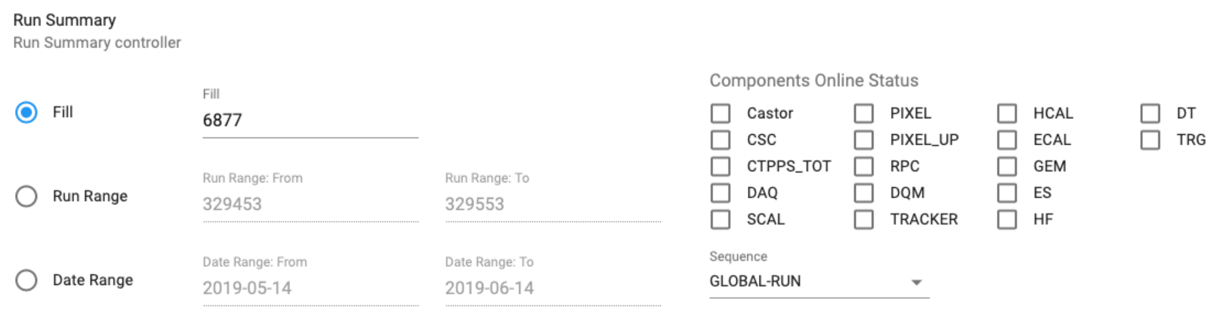
Indexing and ranges

The original indexes for the run table included indexes on the following columns:

- `run_number` (primary key).
- `fill_number` (foreign key).
- `era_id` (foreign key, bitmap index).
- `start_time`.
- `stop_time`.
- `sequence_name`.
- `trigger_mode`.

All of those are columns that are likely to be filtered, but the indexes are not well optimized for filtering on several columns. It was then decided to rethink the whole indexing in this table to try and optimize it. The goal was to have decent speed on all the queries used by the GUI, that is, all the queries generated from the controller shown in Figure 3.3.

As the image shows, runs can be filtered by fill number, run number range or date range. In addition to that, they can be filtered by run component and sequence. We will not consider the


Run Summary
Run Summary controller

[✓ Fill] Fill: 6877

Run Range — Run Range: From 329453 — Run Range: To 329553

Date Range — Date Range: From 2019-05-14 — Date Range: To 2019-06-14

Components Online Status

☐ Castor	☐ PIXEL	☐ HCAL	☐ DT
☐ CSC	☐ PIXEL_UP	☐ ECAL	☐ TRG
☐ CTPPS_TOT	☐ RPC	☐ GEM	
☐ DAQ	☐ DQM	☐ ES	
☐ SCAL	☐ TRACKER	☐ HF	

Sequence: GLOBAL-RUN

Figure 3.3: Run summary controller

run components, because they are stored on a different table. Moreover, it is important to note that the date range is translated as `start_time >= ...` and `end_time <= ...`. In addition to these possible filters, typical queries may filter by fill range, by single run number, by range of start times...

The new indexes act on the following columns:

- `run_number` (primary key).
- `sequence_name`, `run_number`: optimizes filtering by run range and sequence.
- `fill_number`, `sequence_name`: optimizes filtering by fill number, and maybe filter by sequence afterwards.
- `era_id` (foreign key, bitmap index).
- `start_time`: optimizes filtering by `start_time` range.
- `sequence_name`, `start_time`: optimizes filtering by `start_time` range and sequence.
- `stop_time`: optimizes filtering by `stop_time` range.
- `trigger_mode`.

A careful look at the list of indexes above reveals that the only use case that is not covered is filtering by date range. However, it is clear that no index will solve this problem: our filters have inequalities in both `start_time` and `end_time`, and an index on both of them would be the same than sorting just on start time. In fact, indexes are linear in nature, while it is clear that storing the order of end times for each start time would be quadratic, and thus prohibitively expensive even if it were possible.

Filtering by date

To exemplify the problem, we can present the query generated originally:

```sql
SELECT i.*, cnt.total_row_count__
FROM (
  SELECT
      r.fill_number AS fill_number,
      scaling.integrated_lumi_scale_display AS integrated_lumi_scale_display,
      r.stop_time AS end_time,
      scaling.integrated_lumi_scale_factor AS integrated_lumi_scale_factor,
      r.stop_time - r.start_time AS duration,
      r.start_time AS start_time,
      r.sequence_name AS sequence,
      r.run_number AS run_number,
      r.delivered_lumi * scaling.integrated_lumi_scale_factor AS delivered_lumi,
      r.recorded_lumi * scaling.integrated_lumi_scale_factor AS recorded_lumi,
      ROW_NUMBER() OVER (ORDER BY r.run_number DESC) AS row_counter__
  FROM cms_oms.runs r
       LEFT JOIN cms_oms.scaling_info scaling ON r.scale_id = scaling.scale_id
  WHERE r.sequence_name = 'GLOBAL-RUN'
        AND r.start_time >= TIMESTAMP '2018-11-01 00:00:00.0'
        AND r.stop_time <= TIMESTAMP '2018-11-30 00:00:00.0'
  ORDER BY run_number DESC) i
CROSS JOIN
  (SELECT COUNT(*) AS total_row_count__ FROM (
    SELECT
        r.run_number AS run_number,
        r.stop_time AS end_time,
        r.sequence_name AS sequence,
        r.start_time AS start_time
    FROM cms_oms.runs r
    WHERE r.sequence_name = 'GLOBAL-RUN'
          AND r.start_time >= TIMESTAMP '2018-11-01 00:00:00.0'
          AND r.stop_time <= TIMESTAMP '2018-11-30 00:00:00.0'
    ORDER BY run_number DESC)) cnt
WHERE i.row_counter__ BETWEEN 41 AND 60;
```

We can use this opportunity to also compare the execution plans with the old and the new indexing, to see that the difference is very small. The old execution plan is shown in Table 3.3, and the new one in Table 3.4.

We can see that the new index helps partially, but we still have a slow query in our hands. The reason is clear. With the old indexes, only the `stop_time` index was being checked, and this returns all runs that finish before the specified date. Then, we need to check them one by one to see that the `start_time` and `sequence_name` conform to what we want. Even though Oracle decides to optimize this process using the `start_time` and `sequence_name` indexes to generate a bitmap, we still have a slow process.

For the new indexes, the query is a little bit more efficient because it now has the `sequence_name, start_time` index that can be used fully to apply the filters. Still, it is necessary to apply manually the filter for the `stop_time` in all remaining rows.

To solve this problem efficiently, we need to leverage the extra information that we are aware of but the optimizer does not have. In this case, the domain knowledge is clear: we know that `start_time` is smaller than `stop_time`. And the solution presents itself: we can add a fourth condition to the where clause, that is redundant given our domain knowledge, but that can guide the optimizer into making different choices.

Thus, we customize the query builder to add this extra filter when necessary. After the changes, it generates the following query:

```
SELECT i.*, cnt.total_row_count__
FROM (
```

Id	Operation (name)	Rows	Bytes	Cost	Time
0	Select statement	894	273K	1532	00:00:19
1	Table access by index rowid (`trigger_keys`)	1	46	3	00:00:01
2	Index unique scan (`trigger_keys_pk`)	1		2	00:00:01
3	Nested loops	894	273K	1532	00:00:19
4	View	1	13	890	00:00:11
5	Sort aggregate	1	26		
6	Table access full (`runs`)	894	23244	890	00:00:11
7	View	894	261K	642	00:00:08
8	Window sort pushed rank	894	46488	642	00:00:08
9	Hash join right outer	894	46488	642	00:00:08
10	Table access full (`scaling_info`)	22	198	3	00:00:01
11	Table access by index rowid (`runs`)	894	38442	639	00:00:08
12	Bitmap conversion to rowids				
13	Bitmap and				
14	Bitmap conversion from rowids				
15	Sort order by				
16	Index range scan (`runs_start_time_idx`)			23	00:00:01
17	Bitmap conversion from rowids				
18	Index range scan (`runs_sequence_name_idx`)			309	00:00:04

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("HLT"."RUN_NUMBER"=:B1 AND "HLT"."NAME"='HLT_KEY')
   6 - filter("R"."START_TIME">=TIMESTAMP' 2018-11-01 00:00:00.000000000' AND
              "R"."SEQUENCE_NAME"='GLOBAL-RUN' AND "R"."STOP_TIME"<=TIMESTAMP' 2018-11-30
              ↪  00:00:00.000000000')
   7 - filter("I"."ROW_COUNTER__">=41 AND "I"."ROW_COUNTER__"<=60)
   8 - filter(ROW_NUMBER() OVER ( ORDER BY INTERNAL_FUNCTION("R"."RUN_NUMBER") DESC
  ↪  )<=60)
   9 - access("R"."SCALE_ID"="SCALING"."SCALE_ID"(+))
  11 - filter("R"."STOP_TIME"<=TIMESTAMP' 2018-11-30 00:00:00.000000000')
  16 - access("R"."START_TIME">=TIMESTAMP' 2018-11-01 00:00:00.000000000')
       filter("R"."START_TIME">=TIMESTAMP' 2018-11-01 00:00:00.000000000')
  18 - access("R"."SEQUENCE_NAME"='GLOBAL-RUN')
```

Table 3.3: Old execution plan for runs with date filtering.

```sql
SELECT
    r.fill_number AS fill_number,
    scaling.integrated_lumi_scale_display AS integrated_lumi_scale_display,
    r.stop_time AS end_time,
    scaling.integrated_lumi_scale_factor AS integrated_lumi_scale_factor,
    r.stop_time - r.start_time AS duration,
    r.start_time AS start_time,
    r.sequence_name AS sequence,
    r.run_number AS run_number,
    r.delivered_lumi * scaling.integrated_lumi_scale_factor AS delivered_lumi,
    r.recorded_lumi * scaling.integrated_lumi_scale_factor AS recorded_lumi,
```

Id	Operation (name)	Rows	Bytes	Cost	Time
0	Select statement	894	273K	1044	00:00:13
1	Table access by index rowid (trigger_keys)	1	46	3	00:00:01
2	Index unique scan (trigger_keys_pk)	1		2	00:00:01
3	Nested loops	894	273K	1044	00:00:13
4	View	1	13	414	00:00:05
5	Sort aggregate	1	26		
6	View (index$_join$_008)	894	23244	414	00:00:05
7	Hash join				
8	Index range scan (runs_sequence_start_time_idx)	894	23244	10	00:00:01
9	Index range scan (runs_stop_time_idx)	894	23244	1310	00:00:16
10	View	894	261K	631	00:00:08
11	Window sort pushed rank	894	46488	631	00:00:08
12	Hash join right outer	894	46488	631	00:00:08
13	Table access full (scaling_info)	22	198	3	00:00:01
14	Table access by index rowid (runs)	894	38442	627	00:00:08
15	Index range scan (runs_sequence_start_time_idx)	1524		9	00:00:01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("HLT"."RUN_NUMBER"=:B1 AND "HLT"."NAME"='HLT_KEY')
   6 - filter("R"."START_TIME">=TIMESTAMP' 2018-11-01 00:00:00' AND
   ↪   "R"."SEQUENCE_NAME"='GLOBAL-RUN' AND "R"."STOP_TIME"<=TIMESTAMP' 2018-11-30
   ↪   00:00:00')
   7 - access(ROWID=ROWID)
   8 - access("R"."SEQUENCE_NAME"='GLOBAL-RUN' AND "R"."START_TIME">=TIMESTAMP'
   ↪   2018-11-01 00:00:00' AND "R"."START_TIME" IS NOT NULL)
   9 - access("R"."STOP_TIME"<=TIMESTAMP' 2018-11-30 00:00:00')
  10 - filter("I"."ROW_COUNTER__">=41 AND "I"."ROW_COUNTER__"<=60)
  11 - filter(ROW_NUMBER() OVER ( ORDER BY INTERNAL_FUNCTION("R"."RUN_NUMBER") DESC
  ↪   )<=60)
  12 - access("R"."SCALE_ID"="SCALING"."SCALE_ID"(+))
  14 - filter("R"."STOP_TIME"<=TIMESTAMP' 2018-11-30 00:00:00')
  15 - access("R"."SEQUENCE_NAME"='GLOBAL-RUN' AND "R"."START_TIME">=TIMESTAMP'
  ↪   2018-11-01 00:00:00' AND "R"."START_TIME" IS NOT NULL)
```

Table 3.4: New execution plan for runs with date filtering.

```sql
      ROW_NUMBER() OVER (ORDER BY r.run_number DESC) AS row_counter__
  FROM cms_oms.runs r
       LEFT JOIN cms_oms.scaling_info scaling ON r.scale_id = scaling.scale_id
  WHERE r.sequence_name = 'GLOBAL-RUN'
       AND r.start_time >= TIMESTAMP '2018-11-01 00:00:00.0'
       AND r.stop_time <= TIMESTAMP '2018-11-30 00:00:00.0'
       AND r.start_time <= TIMESTAMP '2018-11-30 00:00:00.0'
  ORDER BY run_number DESC) i
CROSS JOIN
  (SELECT COUNT(*) AS total_row_count__ FROM (
    SELECT
        r.run_number AS run_number,
        r.stop_time AS end_time,
        r.sequence_name AS sequence,
        r.start_time AS start_time
    FROM cms_oms.runs r
    WHERE r.sequence_name = 'GLOBAL-RUN'
          AND r.start_time >= TIMESTAMP '2018-11-01 00:00:00.0'
          AND r.stop_time <= TIMESTAMP '2018-11-30 00:00:00.0'
          AND r.start_time <= TIMESTAMP '2018-11-30 00:00:00.0'
    ORDER BY run_number DESC)) cnt
WHERE i.row_counter__ BETWEEN 41 AND 60;
```

that has the execution plan shown in Table 3.5.

Id	Operation (name)	Rows	Bytes	Cost	Time
0	Select statement	143	44759	212	00:00:03
1	Table access by index rowid (`trigger_keys`)	1	46	3	00:00:01
2	Index unique scan (`trigger_keys_pk`)	1		2	00:00:01
3	Nested loops	143	44759	212	00:00:03
4	View	1	13	104	00:00:02
5	Sort aggregate	1	26		
6	Table access by index rowid (`runs`)	143	3718	104	00:00:02
7	Index range scan (`runs_sequence_start_time_idx`)	247		3	00:00:01
8	View	143	42900	108	00:00:02
9	Window sort pushed rank	143	7436	108	00:00:02
10	Hash join outer	143	7436	108	00:00:02
11	Table access by index rowid (`runs`)	143	6149	104	00:00:02
12	Index range scan (`runs_sequence_start_time_idx`)	247		3	00:00:01
13	Table access full (`scaling_info`)	22	198	3	00:00:01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("HLT"."RUN_NUMBER"=:B1 AND "HLT"."NAME"='HLT_KEY')
   6 - filter("R"."STOP_TIME"<=TIMESTAMP' 2018-09-30 00:00:00.000000000')
   7 - access("R"."SEQUENCE_NAME"='GLOBAL-RUN' AND "R"."START_TIME">=TIMESTAMP'
   ↪  2018-09-01
              00:00:00.000000000' AND "R"."START_TIME"<=TIMESTAMP' 2018-09-30
              ↪  00:00:00.000000000')
   8 - filter("I"."ROW_COUNTER__">=41 AND "I"."ROW_COUNTER__"<=60)
   9 - filter(ROW_NUMBER() OVER ( ORDER BY INTERNAL_FUNCTION("R"."RUN_NUMBER") DESC
   ↪  )<=60)
  10 - access("R"."SCALE_ID"="SCALING"."SCALE_ID"(+))
  11 - filter("R"."STOP_TIME"<=TIMESTAMP' 2018-09-30 00:00:00.000000000')
```

Table 3.5: Final execution plan for runs with date filtering.

The difference is immediately clear. The index on `sequence_name, start_time` can be used to retrieve a much smaller range of runs. These runs will be tested for the `stop_time` condition, and then returned. But, given what we know about the relationship between `start_time` and `stop_time`, at most one run will be filtered out. That is, we could use the index to retrieve the exact list of runs that we wanted, except for maybe one extra run.

This is a very ad-hoc solution but it is extremely useful. Runs are very often filtered this way, which means that we needed to optimize it.

3.7.4 Lumisections

The lumisections endpoint was probably the easiest to migrate, because the old endpoint only used data from one external table. However, we still faced some problems concerning some data that were missing from the new database design, but we still needed to return. Besides, it was also necessary to tune the table and indexes.

Beam properties

Some Boolean columns representing beam properties (for example, information concerning if beam 1 and 2 are present or stable) were not deemed important enough to be stored in our new lumisections table (as opposed to the old table). However, for backwards compatibility we wanted to still return these columns in the endpoint. We decided to create a view showing these data with a row per lumisection.

After studying the trigger filling these columns in the old table, it was clear that the data was stored as a bitmap in the table `cms_lhcgmt_cond.lhc_gmt_events`. A new row is logged to this table only when a value on the bitmap changes. Furthermore, not all rows in this table store values relevant to what we want. And, instead of storing a timestamp, it stores two columns: `seconds` (the seconds since 1970, Unix epoch) and `nseconds` (the nanoseconds to add to the previous number of seconds).

Summarizing, we had to find a fast and efficient way to assign to each lumisection the latest flag in the above table, that is, the latest row stored before the start time of the lumisection. This presents both query design and query performance problems.

The `cms_lhcgmt_cond.lhc_gmt_events` table has four relevant columns:

Column	Description
`seconds`	Seconds since 01/01/1970.
`nseconds`	To add to the seconds for a precise timestamp.
`value`	The value that we are interested in. Bitmap.
`groupindex`	Only rows with `groupindex = 8` contain beam information.

Only two indexes could be relevant for our purposes: an index on `seconds` and an index on `(groupindex, seconds)`. The final version of the query was:

```sql
SELECT run_number,
     lumisection_number,
     LEAST(BITAND(beam_smp, 1), 1) AS beam1_present,
     LEAST(BITAND(beam_smp, 2), 1) AS beam1_safe,
     LEAST(BITAND(beam_smp, 4), 1) AS beam1_stable,
     LEAST(BITAND(beam_smp, 8), 1) AS beam1_moveable,
     LEAST(BITAND(beam_smp, 1), 1) AS beam2_present,
     LEAST(BITAND(beam_smp, 2), 1) AS beam2_safe,
     LEAST(BITAND(beam_smp, 4), 1) AS beam2_stable,
     LEAST(BITAND(beam_smp, 8), 1) AS beam2_moveable
FROM (
   SELECT run_number,
         lumisection_number,
          (SELECT MAX(gmt.value) KEEP (DENSE_RANK LAST ORDER BY nseconds)
              FROM cms_lhcgmt_cond.lhc_gmt_events gmt
              WHERE seconds =
                 (SELECT MAX(seconds)
                 FROM cms_lhcgmt_cond.lhc_gmt_events gmt
                 WHERE seconds <= (cast(start_time as date) - date '1970-01-01')
                  ↪  * 24 * 60 * 60
                       AND groupindex = 8)
                 AND groupindex = 8)
       AS beam_smp
   FROM cms_oms.lumisections l);
```

Here, we use `LEAST()` to avoid complicated `CASE` expressions. `BITAND()` will return either 0 or its second argument, since this one is a power of two. Thus, `LEAST()` will return 1 if `BITAND()` returns a non-zero argument, and 0 otherwise. This is exactly what we are looking for.

The resulting execution plan is shown in Table 3.6, where a filter on the run number is added to avoid querying the full `lumisections` table. The idea is to leverage the indexes: first, the index in `(groupindex, seconds)` is used for a special range scan to retrieve the maximum, that is, the second value of the timestamp we need. The outer query then uses the same index to identify a set of values for that second, which will typically be very small; and thus sort on the unindexed `nseconds` over a very small set of values, taking the value for the last row. Note that the query might return a row after the lumisection start time, but it would be only by a fraction of a second and this is not a problem for us. Of course, we could filter on the `nseconds` too, if we wanted to prevent this.

Id	**Operation** (name)	**Rows**	**Bytes**	**Cost**	**Time**
0	Select statement	159	3180	6	00:00:01
1	Sort aggregate	1	22		
2	Table access by index rowid (`lhc_gmt_events`)	1	22	5	00:00:01
3	Index range scan (`lhc_gmt_events_secgrp_i`)	1		4	00:00:01
4	Sort aggregate	1	10		
5	First row	1	10	4	00:00:01
6	Index range scan (min/max) (`lhc_gmt_events_secgrp_i`)	1	10	4	00:00:01
7	Table access by index rowid (`lumisections`)	159	3180	6	00:00:01
8	Index range scan (`lumisections_pk`)	159		3	00:00:01

```
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("GROUPINDEX"=8 AND "SECONDS"= (SELECT MAX("SECONDS") FROM
              "CMS_LHCGMT_COND"."LHC_GMT_EVENTS" "GMT" WHERE "SECONDS"<=(CAST(:B1 AS
              ↪  date)-TO_DATE(' 1970-01-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))*24*60*60 AND "GROUPINDEX"=8))
   6 - access("GROUPINDEX"=8 AND "SECONDS"<=(CAST(:B1 AS date)-TO_DATE(' 1970-01-01
   ↪  00:00:00',
              'syyyy-mm-dd hh24:mi:ss'))*24*60*60)
   8 - access("RUN_NUMBER"=TO_NUMBER(:RUN_NUMBER))
```

Table 3.6: Final execution plan for the beam properties per lumisection.

Indexing

As described above, not a lot of indexes are necessary for the lumisections table. In the end, we have four indexes:

- `run_number, lumisection`: index for the primary key.
- `start_time`: to allow querying by time.
- `fill_number, run_number, lumisection`: useful when we join with the fills table.
- `last_update`: to be used for the insertion procedures.

In this case, the indexes are what could be expected for the kind of queries that we have described, and there is not much to discuss about them.

3.7.5 Eras

The eras endpoint, as simple as it is (it only returns around 40 rows of data), presented some problems when migrating. Those were caused from the change in structure, since the old era table contained all data about the fill, run and time range that an era spans. However, the information is not stored in the new `eras` table, since it is implicit in the fill and run tables.

The problem is that dynamically rebuilding the range information was not so easy, at least with the original index structure. Our runs and fills tables were indexed on `era_id`, but this did not prevent what was, in essence, a full table sort, to be able to determine the minimum and the maximum for each given era.

The solution, for the fills table, was changing the index on `era_id` to one on the pair `(era_id, fill_number)`. Exactly the same thing was done on the runs table, where now `(era_id,run_number)` are indexed. This only left a small problem: the start and end time of the era, which are the start time of the first run and the end time of the last run. This, however, could be solved with a smarter query: instead of taking the `MIN(start_time)` and `MAX(stop_time)`, we queried those columns in the following way:

```
MIN(start_time) KEEP (DENSE_RANK FIRST ORDER BY run_number) AS start_time,
MAX(stop_time) KEEP (DENSE_RANK FIRST ORDER BY run_number) AS end_time
```

which results in the index being used, just as we wanted.

In general, we have used the window functions above at several places in the API. They are useful in combination with a `GROUP BY` clause, to retrieve the first and last item in the group sorted by the given criteria. A lot of the work needed to optimize database queries is based on informing the optimizer about data dependencies that it is not aware of. Functions like `FIRST` and `LAST` are extremely useful for this: for example, in the query fragment above, we are aware that sorting on `start_time` of a run is the same than sorting on `run_number`. By introducing this kind of information in our query, we can make the optimizer use the indexes that are already in the tables.

4. OMS for subsystems

4.1 Subsystems and OMS

CMS is a multipurpose detector built from more than a dozen subdetectors, also called subsystems. The ultimate purpose of the detector is to reconstruct the trajectory of the particles generated in each collision and infer their properties from that analysis. To achieve this, each subdetector measures different magnitudes with different precision, and often also only for certain kinds of particle. For example, the tracker, closest to the beam, measures the track of each particle very accurately, while the muon detector exclusively registers muons, and others are only able to detect charged particles. Some measurements overlap, but in general each subdetector is highly specialized.

The highly modular approach used by CMS has a lot of advantages, but also requires teams to be highly synchronized among themselves and with the teams managing detector-wide infrastructure (for example, the data acquisition group, which includes OMS). OMS itself is a very good tool to ensure that all teams are on the same page and have accurate and real-time information of the status of the detector.

However, OMS strives to be a multipurpose tool, useful for every subsystem, and most subsystem experts need access to very specific data that the other teams are not interested in. It is for this reason that the OMS for subsystems project started. The aim of the project is simple: offer tools for subsystems to build their own OMS pages and workspaces, that they can structure freely to display the data that they need. The idea is to offer a very accessible system where simple data pages can be created with minimal development effort, and to help subsystems that are interested in more complex pages.

Historically, with WbM, everything, including the pages of each subsystem, was under control of the core WbM group. While this approach allowed the core developers to review and rewrite the code coming from the subsystems before deploying it, it had serious drawbacks. For example, every time that a subsystem wanted to change something in their private pages, the central team had to take care of it. What is more, the development of any new subsystem specific page was also the responsibility of the core WbM team, which demanded very high personpower.

For OMS, the approach has been radically different. We have built the whole system to be easy to extend, and each subsystem will be responsible to develop and maintain their own pages. This is, of course, more work for the subsystems, but it also allows them much more freedom to define their own requirements and change them when needed, without waiting for the response of the core team. For the core team, this enables the whole system to be supported with much less personpower. The biggest challenge of this approach is to ensure that the subsystems cannot interfere with the core system, while ensuring that their pages can be accessed from it.

4.1.1 Database

The data storage that the subsystems decide to use does not concern OMS, although subsystems that use an Oracle database can take advantage of the solutions that we have already developed. Luckily, most of them do, and they already manage all their tables themselves.

The OMS core team recommends subsystems to aggregate their data into database views. This makes it easier to query it from an API (as will be discussed in the next subsection), and furthermore simplifies data changes. Most subsystems have highly dynamic data sources that share a common interface. With WbM, changing the data source meant editing the webpage code, even if the whole display logic remained unchanged. By using views, the subsystems can change the underlying data source without having to write and deploy a new version of the code. Furthermore, avoiding deployment is useful because these are usually time-sensitive issues, where the subsystem wants to update their pages as soon as possible.

4.1.2 Aggregation API

Every subsystem needs some kind of API to display their data. Furthermore, most of the GUI components are built to support an API that complies with the JSON API specification. Thus, the Aggregation API was built to be easy to extend, and the recommendation of the core OMS team to the subsystems was to run their own instance of the Aggregation API, customizing their own endpoints. However, some subsystems have successfully set up other APIs with a similar interface and are using them without an issue.

The decision of having each subsystem running their own API instance has to do with the OMS core being safe from interference. The core OMS data should be available even if subsystems deploy faulty code that crashes their API instance. This was a complicated decision to make, because it implied quite a lot of work for the core team in the short term: adapting the API server to be used by external teams, writing documentation, meeting with subsystems to help them set up their own machines... However, we considered that it was the best option in the long term. It is less work for the core team, because at that point subsystems should have their own API instances and be mostly self-sufficient. But, what is more important, the core team should not assume those responsibilities. Subsystem experts know how their subsystem works and what data they want to display, and they have enough technical knowledge to make the middleman (the core team) redundant.

Part of my responsibilities included making the API accessible to subsystems and collaborate in the meetings to guide them in the migration process, although the main contact with the subsystems was one of my colleagues. Most of the subsystem representatives that visited us were not interested in building a complex API, and just wanted to display their database table in the OMS webpage. To this end, we tried to simplify the structure of a basic endpoint in the API. The base classes were key to achieve this, but we also created an script that is able to generate all the endpoint files from a JSON file containing information about a given database table and its columns. This means that subsystems can design a database view for every endpoint that they want to expose via their API, run the script for each view and have a complete API running, with little to no Java development.

4.1.3 Web portal

From the beginning, the web portal was built with a modular approach on mind, anticipating this kind of use case. To achieve this, all pages and portlets are completely dynamic, and there is a graphical interface in place to add, modify, and delete them. There are a good number of default portlets (data tables, different kinds of plots...) that can be heavily customized. Adding, for example, a new data table is a drag and drop process if an endpoint is already available, without writing a single line of code.

The collection of default portlets is quite extensive and it covers most use cases. If a subsystem still wanted to have their custom portlet, it would have to be added to the core system, and in this case it would be carefully vetted. While this is similar to the approach of WbM, the difference is the frequency of such requests, that should be much lower.

4.2 Last value pages

Some of the most complex pages in WbM are the last value pages, an example of which is shown in Figure 4.1.

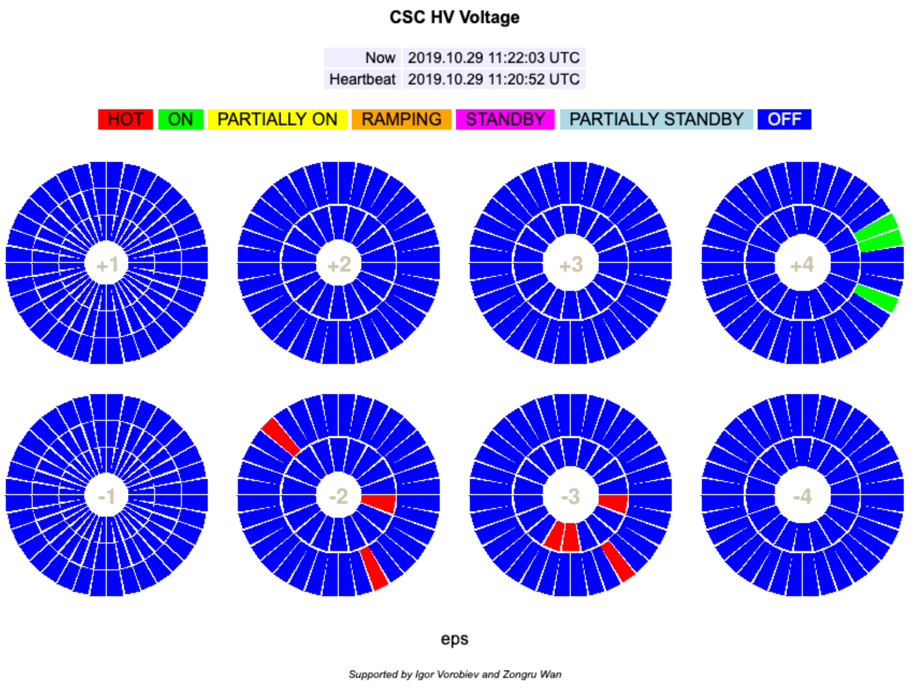

CSC HV Voltage
Now 2019.10.29 11:22:03 UTC
Heartbeat 2019.10.29 11:20:52 UTC
HOT
ON
PARTIALLY ON
RAMPING
STANDBY
PARTIALLY STANDBY
OFF
+1
+2
+3
+4
-1
-2
-3
-4
eps
Supported by Igor Vorobiev and Zongru Wan

Figure 4.1: A WbM Last Value page for the CSC subsystem.

The last value pages offer a schematic of each subsystem, highlighting parts with certain colours depending on their status. Shapes vary wildly between subsystems, and while several of them would still like to access these pages, rewriting the 10,000 line C++ that draws them would require too much manpower.

One of my tasks related to subsystems was to find out how to migrate these pages with minimal effort. In the end, I deemed unreasonable to try to rewrite or translate the C++ file. However, the webpage still uses HTML `<map>` tags to register clicks in each quadrilateral. This means that it is possible to run a script to retrieve the coordinates of each piece.

The script was built with extension in mind. Every subsystem has their own identifier system for their parts, but the last value pages in WbM showed a standard identifier, calculated from the position of each part. The scripts allow each subsystem to translate this positional identifier into a custom identifier. This one is, in turn, the one used to match the database data with each part.

In the end, the script is able to generate a geoJSON file with all the coordinate information and the WbM identifier for each section. This geoJSON is then read by highcharts, our plotting library, to generate the schematic in real time, generating a result that can be seen in Figure 4.2.

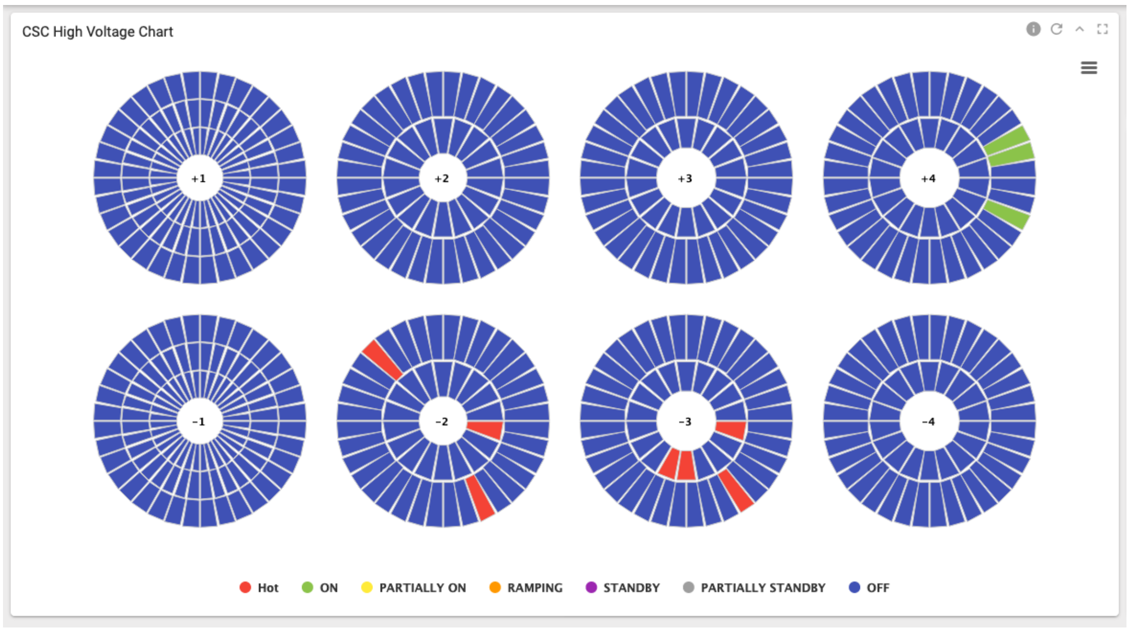


Figure 4.2: An OMS Last Value page for the CSC subsystem.

4.2.1 Displaying the colours

Correctly displaying the colours is more complicated, since it involves matching database entries with the plot. The colours are retrieved from a specific endpoint, which returns a list of sections for each colour. Besides, colours are not stored directly in the database, but are calculated dynamically from the parameters that are there.

The previous Figures 4.1 and 4.2 show the webpage for CSC, and this is also a good subsystem to show the complexity on the database level. Each chamber shown in the detector has a number of positions, which are sensors that can measure voltage. There are voltage ranges defined to decide the status of each position, and then extra rules to decide about the whole chamber. For example, the chamber is considered to be "hot" if there is a single position that is "hot", but it is only "on" if every position is "on".

Initially, the representative of CSC proposed a solution involving procedures calculating the colour data dynamically. However, each query took several seconds to return. I collaborated with the CSC expert to translate their logic into views, that we will discuss below. Using pure SQL to retrieve the data improved the efficiency more than tenfold, with the new views taking less than 100 ms to retrieve the colour for all chambers.

In the end, this is a prototype of the view that we created. The idea is simple: the innermost query checks the voltage ranges, manually defined. They are mutually exclusive by definition, so all values per row will be 0, but a single one with value 1. Those values are then grouped and summed to check the total number of values with the given status. The rest of the logic is on the outermost query, which compares the numbers with the total positions for the given chamber.

```
SELECT
  chamber,
  CASE
    WHEN total_hot > 0 THEN 1
    WHEN total_green = channel_count THEN 2
    WHEN total_green > 0 THEN 3
    WHEN total_ramp > 0 THEN 4
    WHEN total_standby = channel_count THEN 5
    WHEN total_standby > 0 THEN 6
    ELSE 7 -- total_low = channel_count
  END AS color
FROM
   (SELECT
     chamber,
     MAX(position) AS channel_count,
     SUM(hot) AS total_hot,
     SUM(green) AS total_green,
     SUM(ramp) AS total_ramp,
     SUM(standby) AS total_standby,
     SUM(low) AS total_low
   FROM
      (SELECT chamber,
        position,
        CASE WHEN hv > (nominal + 40) THEN 1 ELSE 0 END AS hot,
        CASE WHEN ABS(nominal - hv) <= 40 THEN 1 ELSE 0 END AS green,
        CASE WHEN hv < (nominal - 40) AND hv >= 3100 THEN 1 ELSE 0 END AS ramp,
        CASE WHEN hv < (nominal - 40) AND hv > 2900 AND hv < 3100 THEN 1 ELSE 0
        ↪  END AS standby,
        CASE WHEN hv < (nominal - 40) AND hv <= 2900 THEN 1 ELSE 0 END AS low
      FROM cms_csc_pvss_map.csc_hv_v_lv)
   GROUP BY chamber)
ORDER BY chamber;
```

The original procedures were not much more complicated or inefficient than the query in theory, but in practice they needed a lot of context switches between SQL and PL/SQL, which made them much slower.

Coming back to the discussion in Subsection 4.1.1, CSC is a prime example of why each subsystem should manage their own views. The criteria to decide the status of each chamber changes often, particularly the ranges for voltages. Having the expert of the subsystem managing his own views allows him to update this data as soon and as often as needed.

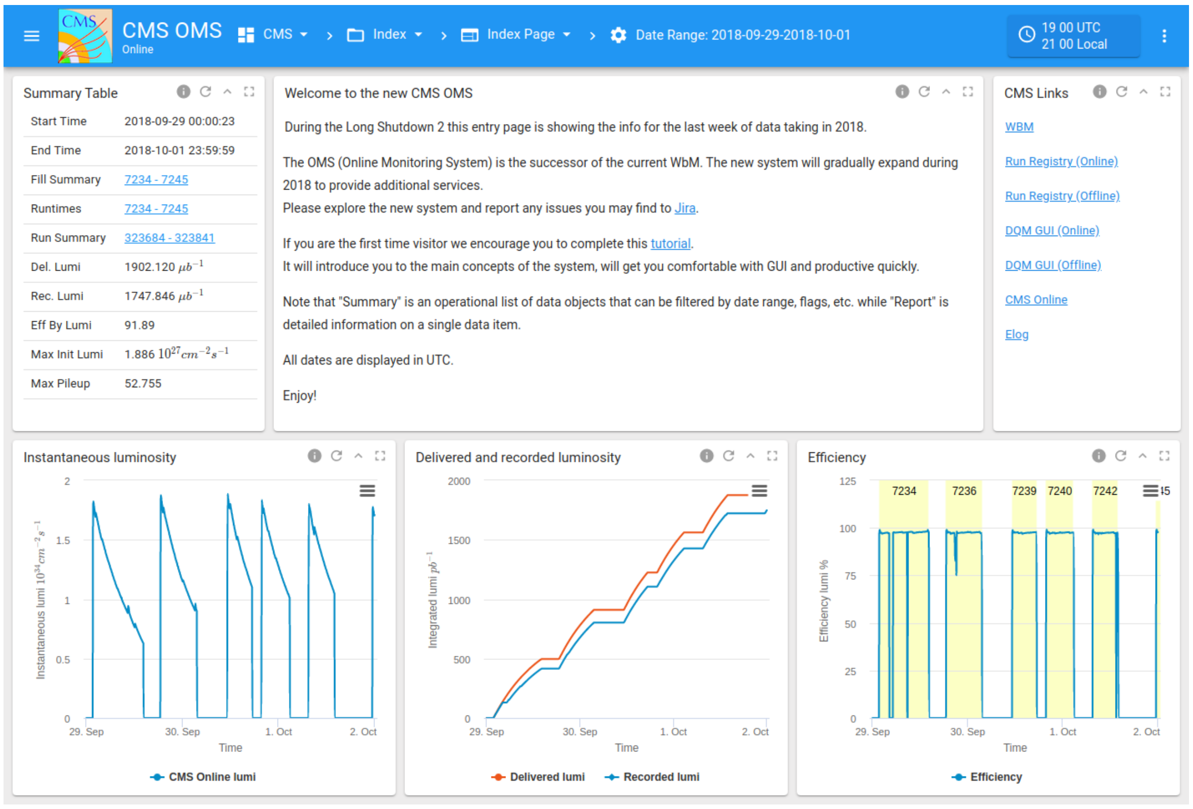
Once the database view is ready, the endpoints are not a problem. We created two endpoint templates: one for the colours, which takes its data from a view just like the one above; and one for a datatable, for subsystems that want to display data per chamber (for example, CSC wanted to display the voltage of each position in the chamber). The templates are available for the subsystems and are ready to run, just by changing the name of the endpoint and of the database tables.

5. Results and conclusion

5.1 Results: the OMS webpage

The previous chapters have discussed all the parts of the OMS system. The best way to showcase the results of all this work, and to present how all those parts interact, is to show the actual OMS webpage. Although I have not written any GUI code at all, all the data shown is provided by the Aggregation API, and ultimately by the database.

The OMS home page is shown in Figure 5.1. This is the style that is used throughout the page, with independent portlets querying one endpoint each with parameters controlled by the top bar. For the index page, the controller is automatically set to filter data for the last week, but the screenshot shows that it can be modified to return data for an arbitrary time period.



Summary Table

Start Time	2018-09-29 00:00:23
End Time	2018-10-01 23:59:59
Fill Summary	7234 - 7245
Runtimes	7234 - 7245
Run Summary	323684 - 323841
Del. Lumi	1902.120 μb^{-1}
Rec. Lumi	1747.846 μb^{-1}
Eff By Lumi	91.89
Max Init Lumi	1.886 $10^{27} cm^{-2} s^{-1}$
Max Pileup	52.755

Welcome to the new CMS OMS

During the Long Shutdown 2 this entry page is showing the info for the last week of data taking in 2018.

The OMS (Online Monitoring System) is the successor of the current WbM. The new system will gradually expand during 2018 to provide additional services.
Please explore the new system and report any issues you may find to Jira.

If you are the first time visitor we encourage you to complete this tutorial.
It will introduce you to the main concepts of the system, will get you comfortable with GUI and productive quickly.

Note that "Summary" is an operational list of data objects that can be filtered by date range, flags, etc. while "Report" is detailed information on a single data item.

All dates are displayed in UTC.

Enjoy!

CMS Links

WBM
Run Registry (Online)
Run Registry (Offline)
DQM GUI (Online)
DQM GUI (Offline)
CMS Online
Elog

Figure 5.1: Screenshot of the OMS home page.

The plots shown in the home page are not chosen randomly. The plots on the left and middle show luminosity, which is the single most important value that OMS provides. It summarizes the performance of the accelerator and the detector: the higher the luminosity, the more collisions are observed. The efficiency plot shown on the right is actually used for the same purpose, since efficiency measures the percentage of time that the detector is collecting data.

All three plots on this page, plus the data table on the top left, take their data from the lumisection summaries endpoint discussed in Subsection 2.7.1. The homepage of OMS shows how important this endpoint is, and from a technical standpoint it also showcases the need of endpoints that are able to return aggregated data for a period of time. To maximize GUI performance, the endpoint just returns 1000 entries, instead of the tens of thousands that could be available for a given date range. While all entries have to be checked and averaged on the database side, we keep the data processing in the front-end to a minimum.

This is also a good opportunity to mention that all plots in OMS support zooming. When the user zooms, data is actually queried again from the original endpoint. This is a testament to the speed and reliability of the Aggregation API, because the zoom feature works immediately and the user does not realize that data is queried again. Zooming in this way also allows the API to return data in the current granularity. The API will return 1000 data points for the original date range, and the same number of points for the zoomed date range, aggregating less rows per point returned.

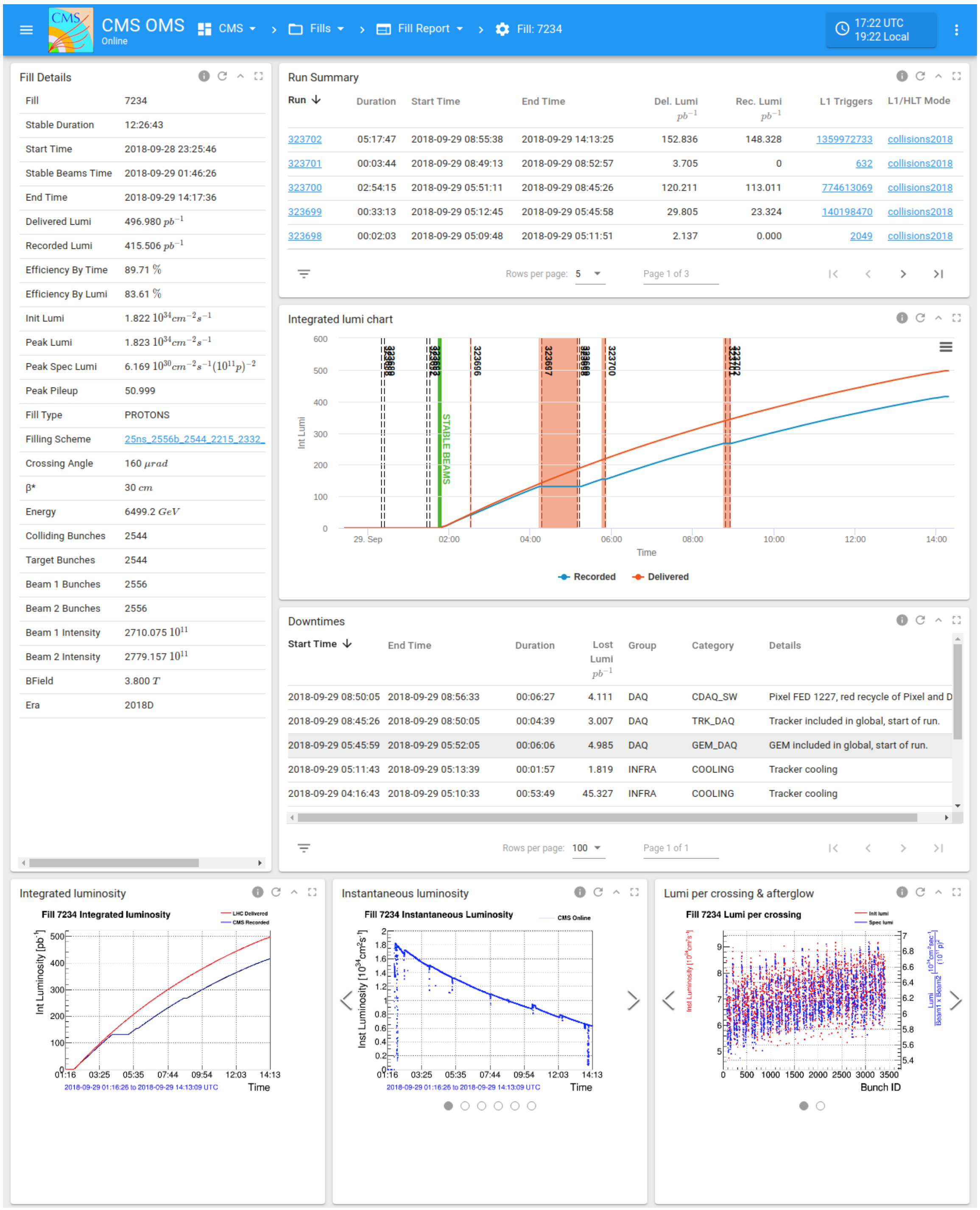
Finally, the main page shows how data units are important in our system, and justifies the need of carefully storing and retrieving them, as we discussed in Section 2.6 and Subsection 3.4.1. All values and plots in OMS are displayed with their units, if they have them. For example, the integrated luminosities in the summary table are displayed in μb^{-1} (where b stands for barns, a unit equivalent to $10^{-28}m^2$), and the instantaneous luminosity is displayed in $cm^{-2}s^{-1}$. In both cases, the value is easier to read if the adequate units are displayed, such that the magnitude can be scaled to a reasonable value (typically between 1 and 1000). However, the way to do this differs because there are different conventions for different types of magnitude. For example, for instantaneous luminosity, it is customary to always use $cm^{-2}s^{-1}$, with an appropriate power of ten before it; while integrated luminosity is typically displayed by adjusting the units to their SI prefix. It is even worse than this: depending on the particles collided, luminosity can be several orders of magnitude different, and there are conventions for the appropriate scaling to use with each kind of fill. We already described in the referenced sections how to solve these problems, but this is a great opportunity to show how they arise with real data.

In addition to the index page, the most important OMS pages are the ones displaying runs and fills. Those display the information in our core tables (fills, runs and lumisections) and also information coming from a handful of additional tables.

The Fill Report shown in figure 5.2 is the go-to page for information about a fill. Apart from the fill details and the additional data tables, the page shows the integrated luminosity per fill. Again, luminosity is key for scientists to understand what happened during a fill, and the customized plot shows even more details. First, the start of each run inside the fill and the start of stable beams is shown. It is interesting to note that no luminosity is delivered until stable beams are declared: as we previously explained, stable beams are declared when the beam is ready for collisions; thus, luminosity is meaningless before that, since the beam is not yet prepared.

Downtimes are also shown in the plot, with orange shading. A downtime is a period of time when the detector isn't working during stable beams, i.e., they mark moments when the beam is correctly configured and collisions are happening but no data are being taken. This is the perfect opportunity to show the difference between delivered and recorded luminosity: if we focus on the bigger downtime period shown in the plot, it is easy to see that recorded luminosity remains constant, while delivered luminosity keeps growing. This is precisely the definition of downtime: LHC keeps providing luminosity, but CMS cannot make use of it.

The last thing that is worth mentioning about the Fill Report is the static plots on the bottom



Fill Details

Fill	7234
Stable Duration	12:26:43
Start Time	2018-09-28 23:25:46
Stable Beams Time	2018-09-29 01:46:26
End Time	2018-09-29 14:17:36
Delivered Lumi	496.980 pb^{-1}
Recorded Lumi	415.506 pb^{-1}
Efficiency By Time	89.71 %
Efficiency By Lumi	83.61 %
Init Lumi	1.822 $10^{34} cm^{-2} s^{-1}$
Peak Lumi	1.823 $10^{34} cm^{-2} s^{-1}$
Peak Spec Lumi	6.169 $10^{30} cm^{-2} s^{-1} (10^{11} p)^{-2}$
Peak Pileup	50.999
Fill Type	PROTONS
Filling Scheme	25ns_2556b_2544_2215_2332_
Crossing Angle	160 μrad
β*	30 cm
Energy	6499.2 GeV
Colliding Bunches	2544
Target Bunches	2544
Beam 1 Bunches	2556
Beam 2 Bunches	2556
Beam 1 Intensity	2710.075 10^{11}
Beam 2 Intensity	2779.157 10^{11}
BField	3.800 T
Era	2018D

Run Summary

Run ↓	Duration	Start Time	End Time	Del. Lumi pb^{-1}	Rec. Lumi pb^{-1}	L1 Triggers	L1/HLT Mode
323702	05:17:47	2018-09-29 08:55:38	2018-09-29 14:13:25	152.836	148.328	1359972733	collisions2018
323701	00:03:44	2018-09-29 08:49:13	2018-09-29 08:52:57	3.705	0	632	collisions2018
323700	02:54:15	2018-09-29 05:51:11	2018-09-29 08:45:26	120.211	113.011	774613069	collisions2018
323699	00:33:13	2018-09-29 05:12:45	2018-09-29 05:45:58	29.805	23.324	140198470	collisions2018
323698	00:02:03	2018-09-29 05:09:48	2018-09-29 05:11:51	2.137	0.000	2049	collisions2018

Rows per page: 5 Page 1 of 3

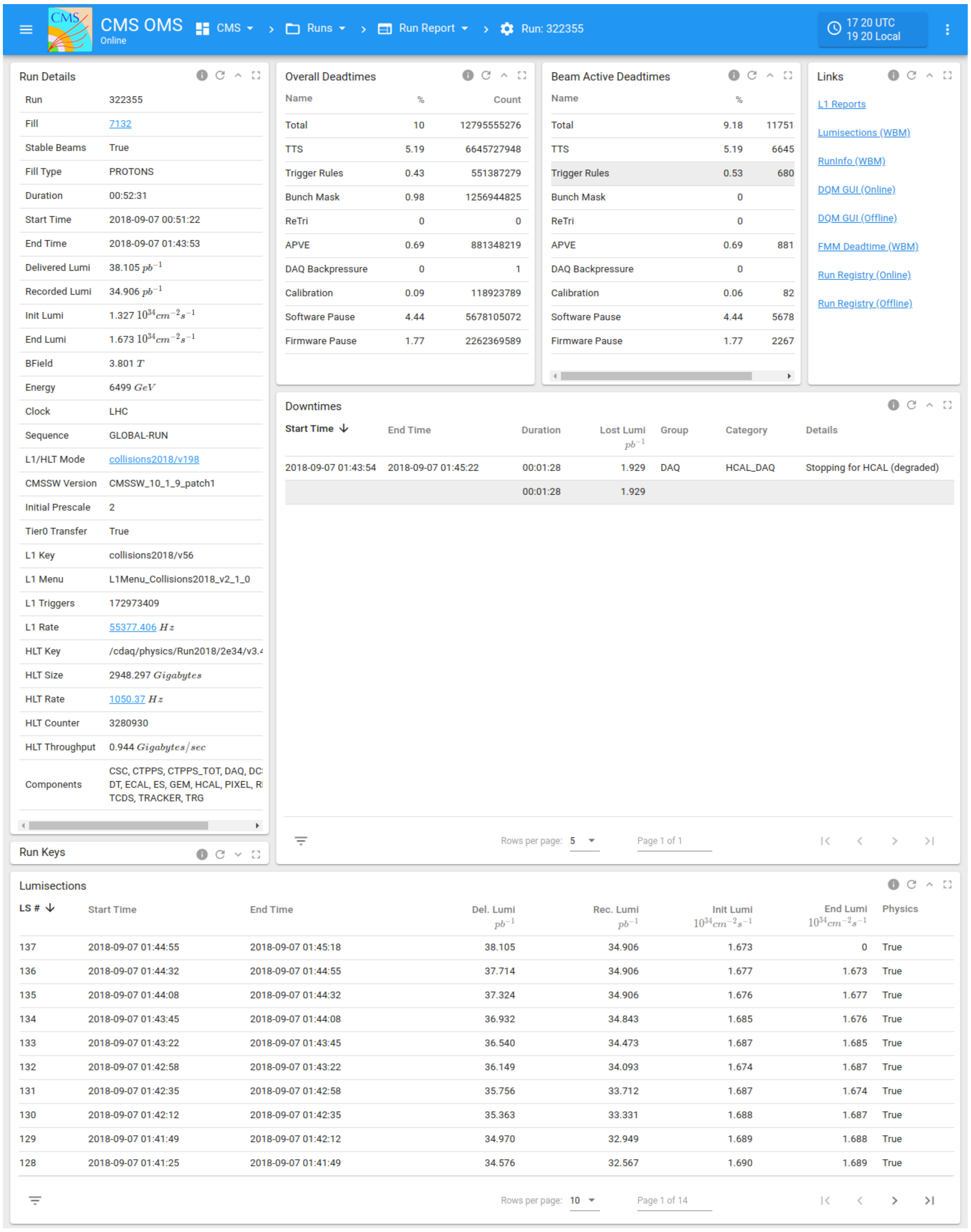
Downtimes

Start Time ↓	End Time	Duration	Lost Lumi pb^{-1}	Group	Category	Details
2018-09-29 08:50:05	2018-09-29 08:56:33	00:06:27	4.111	DAQ	CDAQ_SW	Pixel FED 1227, red recycle of Pixel and D
2018-09-29 08:45:26	2018-09-29 08:50:05	00:04:39	3.007	DAQ	TRK_DAQ	Tracker included in global, start of run.
2018-09-29 05:45:59	2018-09-29 05:52:05	00:06:06	4.985	DAQ	GEM_DAQ	GEM included in global, start of run.
2018-09-29 05:11:43	2018-09-29 05:13:39	00:01:57	1.819	INFRA	COOLING	Tracker cooling
2018-09-29 04:16:43	2018-09-29 05:10:33	00:53:49	45.327	INFRA	COOLING	Tracker cooling

Rows per page: 100 Page 1 of 1

Figure 5.2: Screenshot of the OMS Fill Report.

of the page. These differ significantly from the other plots shown, and that is because they are coming from a different source. These plots are the last piece of the old system to be migrated. They are generated by an old and quite complex C++ program, and creating endpoints to replace them is not easy, mainly because a lot of the data sources are deprecated and new ones would have to be located. The fill report will show the old plots until we are able to recreate them using the technology of OMS.



CMS OMS Online › CMS › Runs › Run Report › Run: 322355 — 17 20 UTC 19 20 Local

Run Details

Run	322355
Fill	7132
Stable Beams	True
Fill Type	PROTONS
Duration	00:52:31
Start Time	2018-09-07 00:51:22
End Time	2018-09-07 01:43:53
Delivered Lumi	38.105 pb^{-1}
Recorded Lumi	34.906 pb^{-1}
Init Lumi	1.327 $10^{34} cm^{-2} s^{-1}$
End Lumi	1.673 $10^{34} cm^{-2} s^{-1}$
BField	3.801 T
Energy	6499 GeV
Clock	LHC
Sequence	GLOBAL-RUN
L1/HLT Mode	collisions2018/v198
CMSSW Version	CMSSW_10_1_9_patch1
Initial Prescale	2
Tier0 Transfer	True
L1 Key	collisions2018/v56
L1 Menu	L1Menu_Collisions2018_v2_1_0
L1 Triggers	172973409
L1 Rate	55377.406 Hz
HLT Key	/cdaq/physics/Run2018/2e34/v3.
HLT Size	2948.297 $Gigabytes$
HLT Rate	1050.37 Hz
HLT Counter	3280930
HLT Throughput	0.944 $Gigabytes/sec$
Components	CSC, CTPPS, CTPPS_TOT, DAQ, DC DT, ECAL, ES, GEM, HCAL, PIXEL, R TCDS, TRACKER, TRG

Run Keys

Overall Deadtimes

Name	%	Count
Total	10	12795555276
TTS	5.19	6645727948
Trigger Rules	0.43	551387279
Bunch Mask	0.98	1256944825
ReTri	0	0
APVE	0.69	881348219
DAQ Backpressure	0	1
Calibration	0.09	118923789
Software Pause	4.44	5678105072
Firmware Pause	1.77	2262369589

Beam Active Deadtimes

Name	%	
Total	9.18	11751
TTS	5.19	6645
Trigger Rules	0.53	680
Bunch Mask	0	
ReTri	0	
APVE	0.69	881
DAQ Backpressure	0	
Calibration	0.06	82
Software Pause	4.44	5678
Firmware Pause	1.77	2267

Links

- L1 Reports
- Lumisections (WBM)
- RunInfo (WBM)
- DQM GUI (Online)
- DQM GUI (Offline)
- FMM Deadtime (WBM)
- Run Registry (Online)
- Run Registry (Offline)

Downtimes

Start Time	End Time	Duration	Lost Lumi pb^{-1}	Group	Category	Details
2018-09-07 01:43:54	2018-09-07 01:45:22	00:01:28	1.929	DAQ	HCAL_DAQ	Stopping for HCAL (degraded)
		00:01:28	1.929			

Rows per page: 5 Page 1 of 1

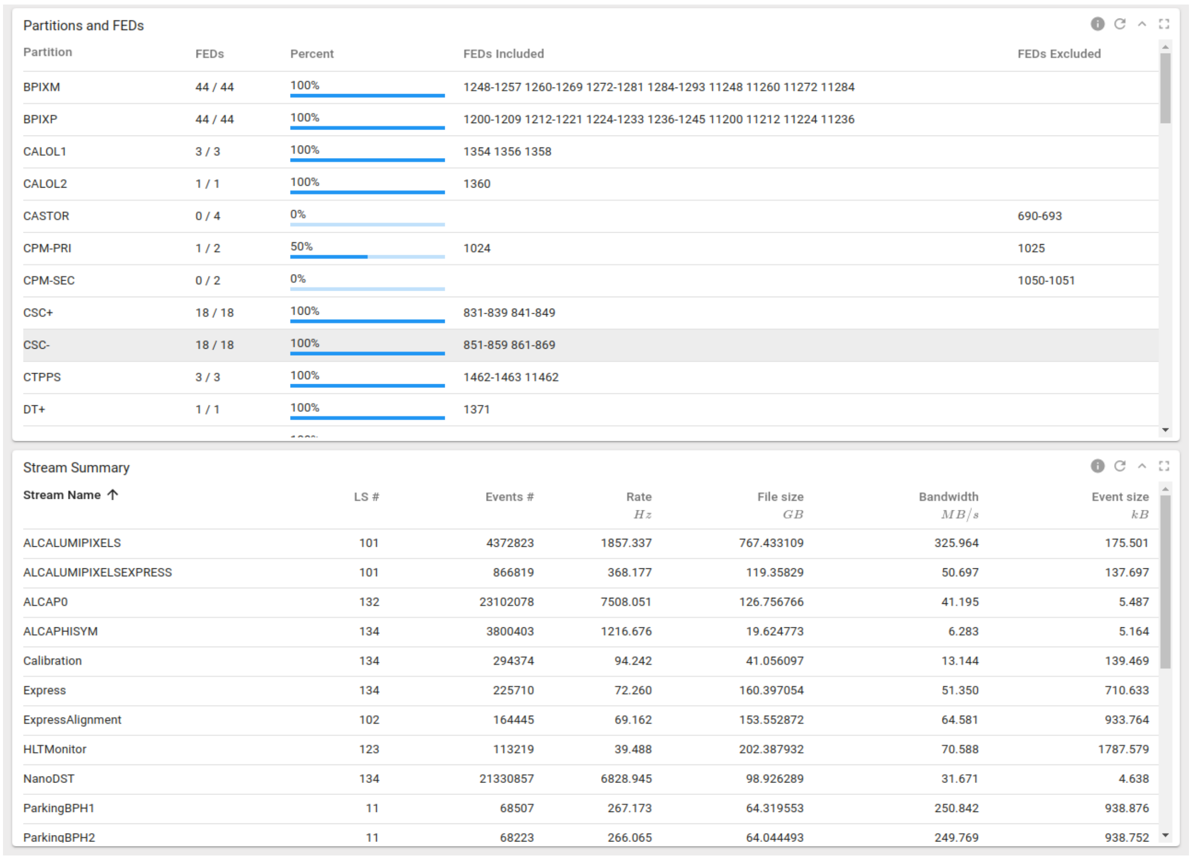
Lumisections

LS #	Start Time	End Time	Del. Lumi pb^{-1}	Rec. Lumi pb^{-1}	Init Lumi $10^{34} cm^{-2} s^{-1}$	End Lumi $10^{34} cm^{-2} s^{-1}$	Physics
137	2018-09-07 01:44:55	2018-09-07 01:45:18	38.105	34.906	1.673	0	True
136	2018-09-07 01:44:32	2018-09-07 01:44:55	37.714	34.906	1.677	1.673	True
135	2018-09-07 01:44:08	2018-09-07 01:44:32	37.324	34.906	1.676	1.677	True
134	2018-09-07 01:43:45	2018-09-07 01:44:08	36.932	34.843	1.685	1.676	True
133	2018-09-07 01:43:22	2018-09-07 01:43:45	36.540	34.473	1.687	1.685	True
132	2018-09-07 01:42:58	2018-09-07 01:43:22	36.149	34.093	1.674	1.687	True
131	2018-09-07 01:42:35	2018-09-07 01:42:58	35.756	33.712	1.687	1.674	True
130	2018-09-07 01:42:12	2018-09-07 01:42:35	35.363	33.331	1.688	1.687	True
129	2018-09-07 01:41:49	2018-09-07 01:42:12	34.970	32.949	1.689	1.688	True
128	2018-09-07 01:41:25	2018-09-07 01:41:49	34.576	32.567	1.690	1.689	True

Rows per page: 10 Page 1 of 14

Figure 5.3: Screenshot of the OMS Run Report (page 1).

The Run Report is shown in figure 5.3. It aggregates all the information from an specific run, and queries a handful of different endpoints to obtain the information it needs: the runs endpoint itself, but also the deadtimes, the downtimes, the lumisections, the DAQ readouts, and the stream summaries. This page is a good example of how performance is key for the Aggregation API. The



Partitions and FEDs

Partition	FEDs	Percent	FEDs Included	FEDs Excluded
BPIXM	44 / 44	100%	1248-1257 1260-1269 1272-1281 1284-1293 11248 11260 11272 11284	
BPIXP	44 / 44	100%	1200-1209 1212-1221 1224-1233 1236-1245 11200 11212 11224 11236	
CALOL1	3 / 3	100%	1354 1356 1358	
CALOL2	1 / 1	100%	1360	
CASTOR	0 / 4	0%		690-693
CPM-PRI	1 / 2	50%	1024	1025
CPM-SEC	0 / 2	0%		1050-1051
CSC+	18 / 18	100%	831-839 841-849	
CSC-	18 / 18	100%	851-859 861-869	
CTPPS	3 / 3	100%	1462-1463 11462	
DT+	1 / 1	100%	1371	

Stream Summary

Stream Name ↑	LS #	Events #	Rate Hz	File size GB	Bandwidth MB/s	Event size kB
ALCALUMIPIXELS	101	4372823	1857.337	767.433109	325.964	175.501
ALCALUMIPIXELSEXPRESS	101	866819	368.177	119.35829	50.697	137.697
ALCAP0	132	23102078	7508.051	126.756766	41.195	5.487
ALCAPHISYM	134	3800403	1216.676	19.624773	6.283	5.164
Calibration	134	294374	94.242	41.056097	13.144	139.469
Express	134	225710	72.260	160.397054	51.350	710.633
ExpressAlignment	102	164445	69.162	153.552872	64.581	933.764
HLTMonitor	123	113219	39.488	202.387932	70.588	1787.579
NanoDST	134	21330857	6828.945	98.926289	31.671	4.638
ParkingBPH1	11	68507	267.173	64.319553	250.842	938.876
ParkingBPH2	11	68223	266.065	64.044493	249.769	938.752

Figure 5.3: Screenshot of the OMS Run Report (page 2).

page has to load with no perceptible delay, to provide a very good user experience. Users expect to change between different runs, compare their data, navigate around... and they should be able to do so without noticeable waiting times.

The previous pages are only a very small summary of what OMS has to offer, but they show the purpose and usefulness of a handful of features that we have described throughout the text. The detailed description of the OMS presentation layer can be found in [2]. Of course, all this interesting features that we just showed for the core OMS page are also available to subsystems, which we discussed in Chapter 4. The current section also shows the kind of components that we use to display our own data, that are generic enough to be adapted to display the data of subsystems. As we detailed in Subsection 4.1.3, subsystem can often incorporate all this functionality without writing a single line of front-end code on their side.

5.2 Conclusion

During my internship, I have worked to improve the whole backend of the OMS webpage. Thanks to my contributions, we have been able to move away from the old WbM database tables, which were a very big source of problems, and we are using a newer schema. Furthermore, the new schema is more adequate to our needs because the requirements are now much more precise and much better defined than when the old schema was designed. Lastly, the new schema only has to cater to a single user, the Aggregation API, which allows for more specific customization, such as indexes for particular queries.

All these changes have been propagated to the API, and have improved it greatly, enabling queries that are faster and easier to write. However, I have also worked on improving the API code itself. Apart from the new endpoints, which of course are necessary to display all the data needed, I have contributed with a handful of new features, useful for users and developers alike. Internally, writing new endpoints is now considerably easier, and most of the endpoint code is now on the abstract shared classes. Externally, the API interface has been improved, with support for new parameters for grouping, improved filtering for certain fields, and a more uniform behaviour all throughout the API (concerning responses, errors, filtering...)

Besides, I have worked on other features that have been requested by other groups or by the core team. Examples of these include CSV serialization for certain pages, or extended support for units and scaling. Finally, I have collaborated to bridge the gap with subsystems and to help them extend the Aggregation API taking advantage of all those new features.

However, even though the system can be considered to be finished in some sense, there are a big number of additions that could be made, both improvements to the existing code and new features that might be helpful.

For the Aggregation API, there is one big change that was discussed throughout my internship but never came to fruition: moving away from Katharsis. We already discussed in Subsection 2.2.2 that Katharsis has been unmaintained for some time and that we are relying on its latest stable version. The Aggregation API would be definitely easier to maintain if we were to move to another library that fulfills Katharsis role, and that is actively maintained. Other possibility is to keep using all the network libraries that Katharsis is based on (Dropwizard, Jersey, Jetty...) to build our RESTful service, and manually implement the serialization and deserialization process that we use Katharsis for. While this second option would definitely be more work, it would prevent a similar situation to happen in the future, since the potential alternatives to Katharsis, such as Crnk [7], are not as established as the network libraries and could also face the same problem at some point.

To stand the test of time, the Aggregation API also needs to be upgraded from Java 8 to the next Long Term Support update, Java 11. This is intertwined with the upgrade from Katharsis, because some of its dependencies do only support Java 8 and the whole Katharsis project would need to be upgraded by us. Public updates are not released anymore for Java 8 [21], so it would be convenient to migrate to Java 11 soon.

The Aggregation API will also need to adapt to the needs of the subsystems, even more than it already has. We expect the subsystems to try to show other kinds of data that the core system did not use. While the Aggregation API has been designed to be easy to extend, so each subsystem can make their own improvements, one of the objectives of the core team is to incorporate all the features that can be useful. For example, a lot of subsystems store files in the database as a BLOB, and the Aggregation API should be able to handle this kind of queries. Others might want to be able to extract data from a secondary database, while the Aggregation API supports one single database by default. There are a lot of quality of life improvements that would be extremely useful for subsystems and that are likely to guide the API development during the months to come, until all the subsystems are satisfied with their instances of the API.

From the point of view of the database, the core tables have already been tested and should not face big changes for the following years. Since they are describing core aspects of the system, it is unlikely that the requirements change during Run 3 (2021 - 2024). However, there are other database improvements that could be of interest.

First, there are still some core tables to be created: the downtime tables. These are the only tables that have not been migrated from WbM, since a redesign of the downtime system is being worked on. These tables will have to be designed and tested carefully, and the corresponding Aggregation API endpoint will need to be adapted to them.

Apart from the core tables, we have mentioned all through this text that the Aggregation API reads data from several different tables that are owned by external groups. In some cases, very detailed data is provided and dynamic aggregation is slower than we would like, slowing the API and, in turn, the OMS webpage. Since we lack direct access to the tables and their indexes, the best option to circumvent this problem would be to build intermediate aggregation tables. These would be filled dynamically with already-aggregated data, either from a trigger or by polling. The obvious maintenance cost would be compensated by the speed-up on the API, very noticeable in some cases. While some experiments in this direction have already been made for specific tables, there is still a lot of work that could be done here.

In conclusion, the Online Monitoring System has been built to replace WbM, but it provides additional functionality on top of what WbM already included. Access to the prototype was provided to the CMS community on early 2018, and right now the latest version is being actively used. Since mid 2019, we have considered OMS to be finished, in the sense that it already has all needed functionality. However, internal improvements are still being made to perfect it, with hopes that it is used for at least another 10 years. Integration with subsystems is also happening simultaneously, aiming for a tool which integrates all subsystems when Run 3 starts in May 2021.

Bibliography

[1] *A new schedule for the LHC and its successor.* URL: https://home.cern/news/news/accelerators/new-schedule-lhc-and-its-successor (visited on 28/03/2020).

[2] Jean-Marc André et al. *Presentation layer of CMS Online Monitoring System.* Tech. rep. CMS-CR-2018-390. Geneva: CERN, Nov. 2018. DOI: 10.1051/epjconf/201921401044. URL: http://cds.cern.ch/record/2649402.

[3] Ulf Behrens. Personal communication. 2019.

[4] *CERN experiments observe particle consistent with long-sought Higgs boson.* URL: https://home.cern/news/press-release/cern/cern-experiments-observe-particle-consistent-long-sought-higgs-boson (visited on 28/03/2020).

[5] *CMS.* URL: https://home.cern/science/experiments/cms (visited on 28/03/2020).

[6] Brice Copy. *DIP Tutorial.* URL: https://readthedocs.web.cern.ch/display/ICKB/DIP+Tutorial (visited on 11/04/2020).

[7] *Crnk.* URL: https://www.crnk.io (visited on 02/04/2020).

[8] *Dropwizard (GitHub page).* URL: https://github.com/dropwizard/dropwizard (visited on 23/09/2019).

[9] Roy Thomas Fielding. 'Architectural Styles and the Design of Network-based Software Architectures'. PhD thesis. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (visited on 12/04/2020).

[10] *How does CMS measure luminosity?* URL: https://cms.cern/news/how-does-cms-measure-luminosity (visited on 29/03/2020).

[11] *Jackson (GitHub page).* URL: https://github.com/FasterXML/jackson (visited on 23/09/2019).

[12] *Java SE Technologies - Database.* URL: https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html (visited on 23/03/2020).

[13] *Jdbi 3 Developer Guide.* URL: jdbi.org (visited on 23/09/2019).

[14] *Jersey.* URL: https://eclipse-ee4j.github.io/jersey/ (visited on 23/09/2019).

[15] *Jetty (GitHub page).* URL: https://github.com/eclipse/jetty.project (visited on 23/09/2019).

[16] *JSON API specification version 1.0.* URL: https://jsonapi.org/format/1.0/ (visited on 28/03/2020).

[17] *Katharsis (GitHub page).* URL: https://github.com/katharsis-project/katharsis-framework (visited on 23/09/2019).

[18] *LHC progress report, week 1.* URL: https://lhc-first-beam.web.cern.ch/News/lhc_080918.html (visited on 03/05/2020).

[19] *LHC: the guide.* URL: https://cds.cern.ch/record/2255762/files/CERN-Brochure-2017-002-Eng.pdf (visited on 29/03/2020).

[20] Audrius Mecionis. *Upgrade by example: WbM to OMS.* URL: https://indico.cern.ch/event/667725/contributions/2743453/attachments/1542448/2419504/WBM_to_OMS_Audrius_Mecionis.pdf (visited on 15/05/2020).

[21] *Oracle Java SE Support Roadmap.* URL: https://www.oracle.com/java/technologies/java-se-support-roadmap.html (visited on 19/04/2020).

[22] *Reconstructing a multitude of particle tracks within CMS.* URL: https://cms.cern/news/reconstructing-multitude-particle-tracks-within-cms (visited on 11/04/2020).

[23] Avi Silberschatz, Henry F. Korth and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 2019.

[24] Rende Steerenberg. *LHC Report: Another run is over and LS2 has just begun...* URL: https://home.cern/news/news/accelerators/lhc-report-another-run-over-and-ls2-has-just-begun (visited on 28/03/2020).

[25] *The Large Hadron Collider.* URL: https://home.cern/science/accelerators/large-hadron-collider (visited on 28/03/2020).

[26] *Where did it all begin?* URL: https://home.cern/about/who-we-are/our-history (visited on 15/05/2020).

Appendix A

Aggregation API reference manual

A.1 Query builder

A query builder is a Java class, subclass of `QueryBuilderV2`, that receives a `QuerySpec` (an object with the information of the user's request) and generates an SQL query to retrieve the adequate objects. The query builder is initialized once per endpoint with all the information about tables and attributes, which is known at compilation time, and then is queried with each request.

What can a query builder do?

Available options:

- Query simple tables.
- Query joined tables (`INNER JOIN` and `LEFT JOIN`).
  - Normal and custom joins.
- Use `GROUP BY` clause and aggregated functions.
  - Special grouping: groups of certain size.
  - Special grouping: specific number of groups.
- Use analytic functions.
- Nest subqueries in the `FROM` clause, as deep as necessary.
- Completely customize attributes.
- Convert request filters to the adequate SQL type.
- Include only the attributes and tables requested.
- `SELECT DISTINCT`.
- Use of `CONNECT BY`, `START WITH`, `PRIOR`.
- Use of `MODEL` clause.

Examples of possible extensions, without modifying the core query builder:

- Support for all kind of aggregated functions (e.g., `LISTAGG`, which has special syntax).
- Support for all kind of filtering modifications (e.g., conversions before filtering).
- Filtering by subquery (e.g., a special filter value `latest` that uses a subquery to return only the latest value).
- Custom sorting (e.g., if sort in a certain attribute is requested, sort by a function or a different attribute).
- Filter operator translation.

A.1.1 Interface

There is only one method that has to be implemented in subclasses: a constructor building up the table and attribute structure (see next sections). However, query builders inherit other methods that can be of interest and that can, sometimes, be overridden:

- `createQueries()`: generates non-paginated queries.
- `createPaginatedQueries()`: generates paginated queries.
- `generateCounterQuery()`: generates a query that counts the number of matches for the original query. It can be overridden if, for this particular endpoint, there is a faster way to count than `SELECT COUNT(*) FROM (<original query>)`.
- `addsCustomPaginationCounter()`: returns a Boolean indicating whether a pagination counter is added manually (`false` by default). A pagination counter is an attribute whose name is stored in the constant `ATTRIBUTE_COUNTER` which counts incrementally for each row, starting from 1. This method should be overridden and return `true` if a custom counter is added.

Lifecycle

When a query builder is used (typically by the repository implementation), two different methods can be called: `createQueries()` and `createPaginatedQueries()`. The first one generates the usual query, and the second one generates a query that only returns the results requested taking into account the `page[limit]` and `page[offset]` query parameters. This second method is usually faster, but it has a problem: as filtering is done in the database side, we don't know the total resource count. That is why these methods also returns a "counter query". This is a special query that calculates the number of total items matching the request.

A.1.2 Table creation for complex table structure

Grouping and subquery behavior must also be specified in the table structure. The following steps show the table creation process, with examples.

1. Create the master table. If your query has a subquery, then this master table should be the table nested at the innermost level.

   ```
   QueryTableMaster master = new QueryTableMaster("schema.name",
   ↪  "abbreviation");
   ```

2. If necessary, create the joined or detail tables at this level:

```
QueryTable tableJoined = master.innerJoin("schema.name", "abbreviation")
        .onColumns("COL_DETAIL", "COL_MASTER");
QueryTable tableDetail = master.joinDetail("schema.name", "abbreviation")
        .onColumns("COL_DETAIL", "COL_MASTER")
        .onColumns("COL_DETAIL_2", "COL_MASTER_2");
QueryTable tableDetailCustomJoin = master.joinDetail("schema.name", "abbr")
        .customOn("A = B");
```

Note that the examples before show different usages of the `onColumns()` and `customOn()` methods. These methods can be used alone or together in any joined or detail table, to specify the joining details.

3. If relevant, specify that the previous tables are nested into a subquery. For this, we retrieve our master table and call to the `asSubquery()` method:

```
QueryTableMaster higherLevelTable =
       master.asSubquery("SUBQUERY_ABBREVIATION");
```

4. If relevant, specify that the previous tables are grouped. For this, we have two different options in our master table (which should be the highest level table declared, that is, if we did not skip step 3, then we should use the table created there):

```
QueryTableMaster groupedTable = higherLevelTable
        .groupBy(attribute1, attribute2);
QueryTableMaster groupedByGranularity = higherLevelTable
        .groupByGranularity(attribute1, attribute2)
        .addGranularity("gran1", attribute1)
        .addGranularity("gran2", attribute1, attribute2)
        .defaultGranularity("gran1");
```

Note that `attribute1` and `attribute2` are `QueryAttributes` (see below) that must be declared as attributes in the nested tables, if a subquery is present, or in our master table if it is not.

The `groupBy()` function is used for default grouping. However, special methods for specific grouping are provided, such as `groupByGranularity()`, which is used when we wish to allow queries with the `group[granularity]` parameter. The value of this parameter specifies which one of the groupings defined above will apply, while the `defaultGranularity()` method is useful for specifying which one of the granularities should be applied by default.

If we want to add more depth (i.e., nest the subquery in another level), just go back to step 2 and use the current top-level table as the master table.

5. We save the top-level table in the attribute `masterTable`.

A.1.3 Query attributes

Query attributes are the core of the query builder. Each query attribute roughly represents a attribute in the response object. All query attributes are subclasses of `QueryAttribute`. They are implemented using the decorator design pattern: the subclasses of `QueryAttributeBase` store the core information about the attribute, and can be decorated with subclasses of `QueryAttribute`

`Decorator`. These subclasses modify or extend the attribute behavior with respect to its definition in the `SELECT` clause, filtering, sorting...

After creating an attribute, it has to be added to the query builder via either the `addIdentifyingAttribute()` or `addAttribute()` functions. These two methods do the same thing, but `addIdentifyingAttribute()` tells the query builder that this attribute should always appear in the `SELECT` clause, independently of whether the user requested them. For this reason, it should be used with attributes which are part of the identifier.

Attributes are created by decorating a base class with zero or more decorators, which alter the default behaviour on different ways (for example, they can change the filter translation, wrap the attribute into a function...). The following is a list of the base attributes:

- **Column attributes**: for data coming from a table column.
- **Literal attributes**: for literal data, they are initialized with a string that has to be valid SQL, since it is written directly into the query and not validated.
- **Dynamic attributes**: sometimes, an attribute needs to be calculated from other attributes, but this is more easily done on the Java code. In that case, a dynamic attribute is created. It is a dummy attribute that is created with information about the attributes that it depends on. When this attribute is requested, the query builder adds all the underlying attributes to the query, even if those are not requested, since they will be needed later to compute the dynamic attribute.
- **Multi-argument function**: used for attributes that are completely based on two or more other attributes, combined with a function or an operator. It is useful to use different attributes as building blocks, reducing significantly the complexity of the final attribute and enabling reuse of the parts. This is used, for example, when an attribute is calculated as the sum of two other attributes.

Furthermore, we provide a fifth type of base attribute. This is one special attribute used to rank the rows in groups of a specified size, or in a given number of equally-sized groups. All the members of a group will receive the same rank, and this value can be used later in a `GROUP BY` clause.

For the decorators, here are some examples:

- **Function**: one of the simplest decorators, it wraps the attribute `SELECT` expression into a given function. Other (literal) arguments can be added to the function.
- **Analytic function**: similar to the previous one, but adds the possibility of specifying an analytic clause, and the desired `ORDER BY` and `PARTITION BY`.
- **Filter conversions**: maybe the most used of all the decorators, different conversions can be provided. A conversion defines the generation of the where condition when the attribute appears on a filter. Numbers, strings, dates, booleans, etc. are provided by default, while a generic class is provided to be subclassed, thus enabling custom filter conversion.
- **Sort literals**: a literal passed as an argument replaces the default expression for this attribute in the `ORDER BY` clause.
- **Subquery attributes**: used to generate an attribute from another attribute that was selected in a nested subquery. The query builder uses these attributes to determine what should and should not be shown in the subquery.

- **Subselect attributes**: specifies that an attribute on a detail table should be selected with a subquery in the `SELECT` clause, instead of joining the table in the `FROM` clause. Useful for performance in some cases.

A.2 Annotations for endpoint tuning

As part of all this modernization process, we found several endpoint features that were better expressed as annotations. Most of them have to do with the deserialization and serialization process: we want to modify how certain fields are deserialized or serialized but we want to keep control on which ones and be able to customize this. Annotations are made for this kind of use case, and we created several custom annotations that apply to one single field or to a whole endpoint resource class. Then, we customized the serializer and deserializer in Katharsis to include support for these annotations.

We support several annotations that can be used by a developer to tune the input (query specification deserialization) or the output (JSON serialization) of a given endpoint.

A.2.1 Query specification deserialization

We also allow 5 annotations that modify the default deserialization configuration. As opposed to the annotations in the previous section, these already existed: they are provided by the Jackson library. However, they were not supported by Katharsis. This was one of the two changes that had to be made in Katharsis's source code and couldn't be done in our API, since Katharsis directly ignored all annotations for serialization.

`@Aggregating`

This annotation is used on endpoints that use grouping (typically endpoints that use the `group[count]`, `group[size]` or `group[granularity]` query parameters). It marks an attribute that is obtained by aggregating another attribute. The annotation enables filtering by the aggregated parameter, even if this is not a field in the response.

The typical use case is an endpoint that returns objects for each *range* of run numbers. This means that the response objects only has `first_run_number` and `last_run_number` attributes indicating the endpoints of the range, but we would like to filter the results by run number. The attributes, then, would look like this:

```
@JSONApiResource(type = "Resource")
class Resource {
    // ...

    @Aggregating(field = "run_number", function = AggregationFunction.MIN)
    private Integer first_run_number;

    @Aggregating(field = "run_number", function = AggregationFunction.MAX)
    private Integer last_run_number;

    // ...
}
```

Adding the annotations will enable filtering by `run_number` with certain operators. In the previous example, filtering by `run_number` with operator GT or GE will be translated to a filter by `first_run_number` with the same operator. Similarly, filtering by `run_number` with operator LT or LE will be translated to a filter by `last_run_number` with the same operator. Trying to filter by `run_number` with a different operator will fail.

Note that if aggregation functions other than `MIN` or `MAX` are chosen, any filter on `run_number` will fail.

`@AllowFilter`

Normally, the deserialization process throws an error if the user tries to filter by a field that is not present in the return object. However, sometimes there is a reason to enable this kind of filters. That is what the `@AllowFilter` annotation is for.

A resource using this annotation will look like this:

```
@JSONApiResource(type = "Resource")
@AllowFilter(fields = "strange_filter")
class Resource {
    // ...
}
```

If an endpoint annotated in this way is filtered by `strange_filter`, this filter will be deserialized with no checks at all. In this case, the value for the filter will be stored as a String.

This annotation is meant to be used jointly with the `mapSpecialFilter(FilterSpec,QuerySpec)` method present on `BasicResourceRepository` and its children. All filters that are only allowed due to the annotation will be removed from the query specification and used as an argument to this method. The method, then, should modify the query specification with one or several *normal* filters.

`@AllowUnknownAttributes`

This annotation is a wildcard version of the previous one. It just allows every filter. Example code using this annotation would look like this:

```
@JSONApiResource(type = "Resource")
@AllowUnknownAttributes
class Resource {
    // ...
}
```

This is very rarely used, and its usage should be considered carefully. It is rarely necessary to skip filter checking entirely, and typically stricter and type-safer solutions can be found.

`@BypassTypeChecking`

Normally, the deserialization process tries to convert the value of every filter to the type of the corresponding field. However, sometimes we want to accept different values. For example, we could accept the character ‘%’ as part of a number for autocompleting features. Or we want to accept the string “latest” instead of a number to just get the latest value for the given field.

The resource class would look like this:

```
@JSONApiResource(type = "Resource")
class Resource {
    // ...

    @BypassTypeChecking(allowedValues = {"last", "latest"})
    private Integer lumisection;

    // ...
}
```

If we try to filter by lumisection with value "last" or "latest", the filter will be propagated with string values. If we try to do it with a normal integer value, it will be converted to integer normally. If we try any other value, it will throw an exception.

`@SupportsGranularities`

Since most endpoints do not support grouping by granularity, the default implementation throws an error if the `group[granularity]` is specified. The `@SupportsGranularities` annotation allows to specify the granularities that will be allowed through the deserialization process, as well as to select a default granularity to be used when no granularity is specified.

The resource class would look like this:

```
@SupportsGranularities(
        granularities = { ResourceMeta.GRANULARITY_A,
                          ResourceMeta.GRANULARITY_B },
        defaultGranularity = ResourceMeta.DEFAULT_GRANULARITY)
@JSONApiResource(type = "Resource")
class Resource {
        // ...
}
```

A.2.2 JSON serialization

Several Jackson annotations are supported to control the JSON serialization process.

`@JsonInclude`

`@JsonInclude` is supported partially, with values `Include.ALWAYS`, `Include.NON_NULL`, `Include.NON_ABSENT` and `Include.NON_EMPTY`. A field would be annotated as follows:

```
@JsonApiResource(type = "Resource")
    class Resource {
    // ...

    @JsonInclude(value = JsonInclude.Include.NON_NULL)
    private Integer lumisection;

    // ...
}
```

and it would only appear in the return object if its value was not null. For `Include.ALWAYS`, the field will be always included (default); `Include.NON_ABSENT` will work like `Include.NON_NULL`; and `Include.NON_EMPTY` will not include `null` nor empty collections.

`@JsonUnwrapped`

A map can be annotated with `@JsonUnwrapped` to unwrap its contents into the level above. This is used for creating highly dynamic endpoints, where the attributes change for each call. Filtering, of course, is not working out of the box with this kind of attributes, so typically this annotation is used jointly with `@AllowUnknownAttributes`. Using this annotation is not recommended in general, because typical endpoints should have a static structure.

`@JsonIgnore`

A field annotated with `@JsonIgnore` is never present in the JSON serialization.

`@JsonProperty`

`@JsonProperty` is partially supported to rename fields in the JSON file. Only the `value` variable will be considered, all other elements in the annotation will be ignored. A resource field annotated with `@JsonProperty(value = "abc")` will appear on the JSON file with name `abc`.

A.3 Extensions to the JSON API specification

This page describes the additions made to the JSON:API specification, and the deviations from it. The Aggregation API conforms to the 1.0 version of the JSON API specification [16], which explains our generic URL format. This section only describes the deviations from the standard.

A.3.1 General guidelines

The following rules are followed project-wide but for very specific exceptions.

- Field names always use `lower_snake_case`.
- The response object for a given endpoint is static, that is, the response format stays the same for different request parameters in the same endpoint.
- Only fields appearing in the response object can be filtered on or sorted by.
  - Exception: a few endpoints allow filtering on certain attributes even if it is not a part of the return object.
  - Exception: endpoints which aggregate ranges and present them with attributes of type `first_attr`, `last_attr` can typically be filtered and sorted by `attr` too.
- If a field that does not exist appears on the request in whatever manner (`filter`, `sort`, `fields`...) the API will return an HTTP 400 (bad request) error.
- All dates must be specified in ISO 8601 format, although `yyyy-MM-dd HH:mm:ss` is also accepted. The dates will always be output as a string with the same ISO 8601 format, unless `include=presentation_timestamp` is specified as part of the request, in which case the second format will be used.

A.3.2 `page` request parameter

Aggregation API uses `page[offset]` and `page[limit]` to paginate. Note that the specified offset must be a multiple of the limit or else an error will be thrown.

A.3.3 `filter` request parameter

Aggregation API enables filtering with the format `filter[field][operator]=value`. `field` is the name of the field to filter by. `operator` is one of the following:

- `EQ`: equals.
- `NEQ`: not equals.
- `GT`: greater than.
- `GE`: greater or equal than.
- `LT`: less than.
- `LE`: less or equal than.
- `LIKE`: filter on similarity. Similar to SQL `LIKE` operator.
- `CT`: contains. This is rarely allowed, and only for array values.

If the operator is omitted, `EQ` is assumed.

However, we have a special use case which is not covered by the normal filtering: lumisections are uniquely identified with a run number and lumisection number. However, the filtering structure described above doesn't allow to get the range between two lumisections, with filters like:

```
?filter[run_number][GE]=320000&filter[lumisection][GE]=20
        &filter[run_number][LE]=320005&filter[lumisection][GE]=30
```

which will return lumisections that have both lumisection number between 20 and 30 and run number between 320000 and 320005 (and it will not return, for example, lumisection with number 5 in run 320002).

Tuple filtering solves this problem. We can write the following filter:

```
?filter[run_number,lumisection][GE]=320000,20
        &filter[run_number,lumisection][LE]=320005,30
```

and this will return the values with `(320000,20) <= (run_number,lumisection) <= (320005,30)` (sorted lexicographically). That is, it will return the lumisections for run 320000 that have number greater than 20, all the lumisections for runs strictly between 320000 and 320005, and the lumisections for run 320005 that have number less than 30.

This filtering mode is not supported for all endpoints or for all pairs of attributes.

A.3.4 `sort` request parameter

Sorting is fully supported as suggested in the JSON API specification.

A.3.5 `fields` request parameter

Sparse fieldsets are fully supported as suggested in the JSON API specification.

A.3.6 `group` request parameter

Certain endpoints support grouping specification in the query. This has two different modes:

- `group[count]` and `group[size]`: these two parameters should never be used at the same time (and using them will throw an HTTP 400 error). `group[count]` specifies the number of groups returned, and these groups will be of the same size (or off by 1). `group[size]` specifies the size of each group, and the last group may be of a different size.
- `group[granularity]` specifies which attribute will the API group on. The valid attributes depend on the endpoint, and may not be attributes in the returned object.

A.3.7 `include` request parameter

In addition to the default behavior suggested in the JSON API specification, we use the `include` parameter to specify application flags:

- `meta`: adds the metadata to the default response.
- `presentation_timestamp`: specifies that returned resources should present the timestamps in `yyyy-MM-dd hh:mm:ss` format, instead of the default ISO 8601 date format.

A.4 Query verification

Query verifying is another problem that needs to be solved in almost every endpoint, but at the same time needs to be defined in a case-by-case basis. It is a consequence of the limitations of our data source: often, we do not wish to allow a user to retrieve all the entries in the database for a given endpoint, or maybe we want to force them to use specific columns for the filtering, usually because they are indexed.

However, the rules for allowing or disallowing specific filters depend completely on the endpoint and, ultimately, on the database structure. This means that the developer must be able to decide which filters to allow and which to block, but it is useful to have a common framework such that the filter list doesn't have to be inspected manually every time.

The query specification verifier does exactly this. It can be set up with a set of rules, and it will automatically check every query specification to verify that the endpoints conform to those rules. There are two kind of rules: accept and reject. If any accept rule is specified, the query specification has to conform to one of them to be accepted. For every reject rule specified, the query cannot contain this specific combination of filters.

An example query specification verifier would be built via:

```
verifier.accept().filterFor(ATTRIBUTE_A).withEquality()
        .and().filterFor(ATTRIBUTE_B).withEquality();

verifier.accept().filterFor(ATTRIBUTE_C).withRange();

verifier.reject().granularity(GRANULARITY_1)
        .and().filterFor(ATTRIBUTE_D);

verifier.reject().filterFor(ATTRIBUTE_E);
```

This query specification verifier would enforce the following rules:

- A query specification should either filter by both `ATTRIBUTE_A` and `ATTRIBUTE_B` with operator `EQ`, or filter by `ATTRIBUTE_C` with a range (specified with `LT` or `LE` and `GT` or `GE`). If it doesn't, an error will be thrown.
- A query specification cannot specify granularity `GRANULARITY_1` and filter by `ATTRIBUTE_D` at the same time, or it will be rejected (even if it complies with the previous rule).
- A query specification cannot filter by `ATTRIBUTE_E`, or it will be rejected (even if it complies with the previous rules).

Appendix B

Related code

The complete code of the Aggregation API and the related SQL code will be provided to the members of the tribunal through the shared folder enabled for such purpose. All rights to that code, and to any fragments of it published in the present text, are reserved by CERN, for the benefit of the CMS collaboration.