Visualización de datos de las publicaciones científicas del CERN

Trabajo de fin de máster

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

VNiVERSiDAD D SALAMANCA

CAMPUS DE EXCELENCIA INTERNACIONAL

Junio de 2016

Autora
Alicia Boya García

Tutor
Rodrigo Santamaría

Dr. Rodrigo Santamaría, profesor del Departamento de Informática y Automática de la Universidad de Salamanca.

CERTIFICA:

Que el trabajo titulado "*Visualización de datos de las publicaciones científicas del CERN*" ha sido realizado por D. Alicia Boya García, con DNI 71038815A y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Máster de la Titulación Master Universitario en Ingeniería Informática de esta Universidad.

Y para que así conste a todos los efectos oportunos.

En Salamanca, a 02 de enero de 2021.

Dr. Rodrigo Santamaría
Dpto. de Informática y Automática
Universidad de Salamanca

Resumen

Se describe el desarrollo de *HEPData Explore*, una herramienta de búsqueda y visualización de datos de física de partículas desarrollada en colaboración con el *CERN*.

Esta herramienta se alimenta de los artículos registrados en *HEPData*, un repositorio abierto de datos científicos donde se registran los resultados de las investigaciones realizadas con el *LHC* y otros aceleradores de partículas. A partir de estos datos se construye un índice que permite hacer recuperación de información eficiente.

Se ha desarrollado una interfaz mediante la cual un usuario puede construir filtros complejos con los que localizar tablas de datos dentro de todos los artículos almacenados en el índice anterior.

La aplicación sugiere gráficos de forma automática a partir de los conjuntos de datos encontrados en cada búsqueda a la vez que es suficientemente flexible para permitir al usuario modificarlos de acuerdo a sus necesidades. De forma automática se generan visualizaciones compuestas con datos de varios artículos superpuestos.

Es posible examinar la lista de publicaciones que originan los datos visualizados, así como descargar los datos representados en formatos fáciles de procesar desde un lenguaje de programación sin perder las referencias a las fuentes originales.

Palabras clave: *visualización de datos, recuperación de información, repositorios abiertos, física de partículas.*

Abstract

In this document *HEPData Explore* is introduced, a new data retrieval and visualization tool for high energy physics developed in a collaboration between *USAL* and *CERN*.

This tool feeds from data of scholarly publishing recorded in *HEPData*, an open repository for high energy physics where research results from *LHC* and other particle accelerators are stored. From this data an index that powers efficient information retrieval is generated and maintained.

The application features a user interface where complex filters can be built in order to find sets of matching data tables from all of the articles stored in the index.

The application automatically suggests composite plots from the retrieved data sets and allows the user to modify them.

The application allows querying the list of publications where the plotted data came from. Also, plotted data can be downloaded in readily-parsable formats suitable for offline analysis without losing the references to the original data.

Keywords: *data visualization, information retrieval, open repositories, high energy physics.*

Índice general

CAPÍTULO 1

Introducción

La Organización Europea para la Investigación Nuclear (CERN, por sus siglas en francés) es responsable de la operación del mayor laboratorio de física de partículas del mundo. Este laboratorio contiene un complejo de ocho aceleradores de partículas en funcionamiento [1], incluyendo el Gran Colisionador de Hadrones (LHC), el más grande del mundo. El laboratorio cuenta con un personal de 2.500 personas y 15.000 usuarios de universidades de todo el mundo. [2]

Desde su fundación en 1954, este instituto ha realizado un gran número de experimentos. Los datos de estos experimentos aparecen reflejados en tablas y anexos de los artículos científicos que motivaron su realización.

Es muy frecuente que varios investigadores requieran los mismos datos de experimentos para trabajos diferentes. Por este motivo ha habido un esfuerzo en indexar toda esta información de forma que sea públicamente accesible y localizable.

Por un lado, el CERN ha establecido normas para asegurar que todos los artículos que contengan datos de experimentos de física de partículas sean publicados en *acceso abierto*. [3]

Por otro lado, el proyecto HEPData (*High Energy Physics Data*), iniciado por la Universidad de Durham (Reino Unido) en 1975 ha construido un repositorio centralizado en línea que indexa todas las tablas de datos de los artículos accesibles públicamente relativos a los experimentos realizados en el LHC y otros aceleradores de partículas.

[1] The Accelerator Complex
http://home.cern/about/accelerators

[2] CERN Personnel Statistics 2012
http://council.web.cern.ch/council/en/governance/TREF-PersonnelStatistics2012.pdf

[3] Open Access Policy for CERN Physics Publications
http://opendata.cern.ch/record/411/files/CMS-Data-Policy.pdf

En este momento HEPData cuenta con 64.000 tablas de datos provenientes de más de 8.000 artículos científicos. Existen formatos de datos estandarizados para que los científicos puedan subir a la plataforma los datos de un nuevo artículo como un simple fichero.

Actualmente HEPData cuenta con dos sitios web que permiten consultar el repositorio: http://hepdata.cedar.ac.uk/ contiene la versión estable pero antigua, mientras que https://hepdata.net/ contiene la versión actual, aún en desarrollo.

Ambas interfaces permiten al usuario buscar una publicación por su título o por etiquetas, ver los datos en formato textual, organizados en tablas y descargarlos en formatos reconocidos por varias herramientas. La versión nueva incluye además una pequeña representación de los datos de cada tabla en forma de diagrama de dispersión de manera que es posible hacerse con una idea general de los datos sin necesidad de tratarlos con una herramienta externa.

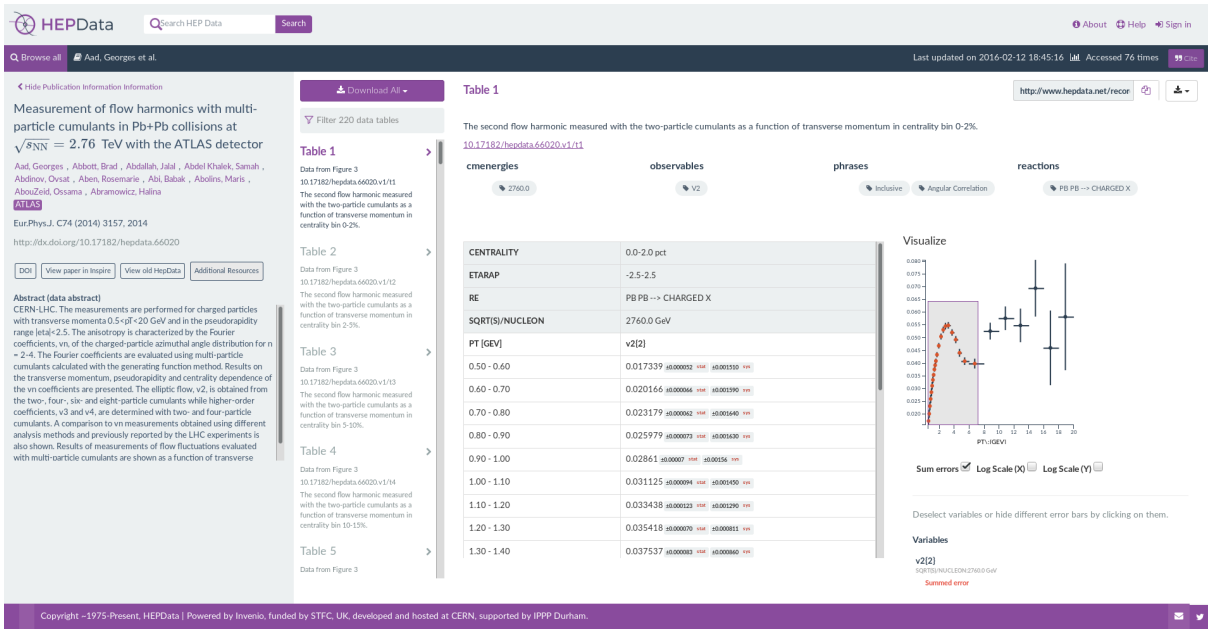

Figura 1.1: La interfaz de HEPData nueva, incluyendo un gráfico de dispersión al lado de la tabla de datos.

1.1 Este trabajo

A veces los científicos necesitan en sus investigaciones consultar varias tablas de datos relacionados, que pueden pertenecer a diferentes publicaciones.

Actualmente para cubrir esta necesidad es necesario buscar individualmente cada publicación

que pueda tener relación con la investigación, descargar las tablas relevantes y combinarlas manualmente para obtener una visualización de datos conjunta. Buscar tablas con diferentes parámetros puede ser una tarea muy tediosa.

Este trabajo tiene por objetivo el desarrollo de una nueva herramienta para consultar y visualizar los datos de los artículos alojados en HEPData. Esta herramienta permitirá localizar datos de acuerdo a una serie de criterios (por ejemplo, el tipo de reacción o un determinado conjunto de variables dependiente e independiente) y visualizarlos de forma conjunta.

El usuario puede modificar los criterios en cualquier momento, tras lo que obtiene una visualización actualizada. Finalmente, el usuario puede descargarse los datos conjuntos para darles un tratamiento adicional y obtener una lista de los artículos de los que provienen.

El resultado final de este proyecto es una aplicación de visualización completa y funcional accesible públicamente desde la siguiente URL:

http://hepdata.rufian.eu/

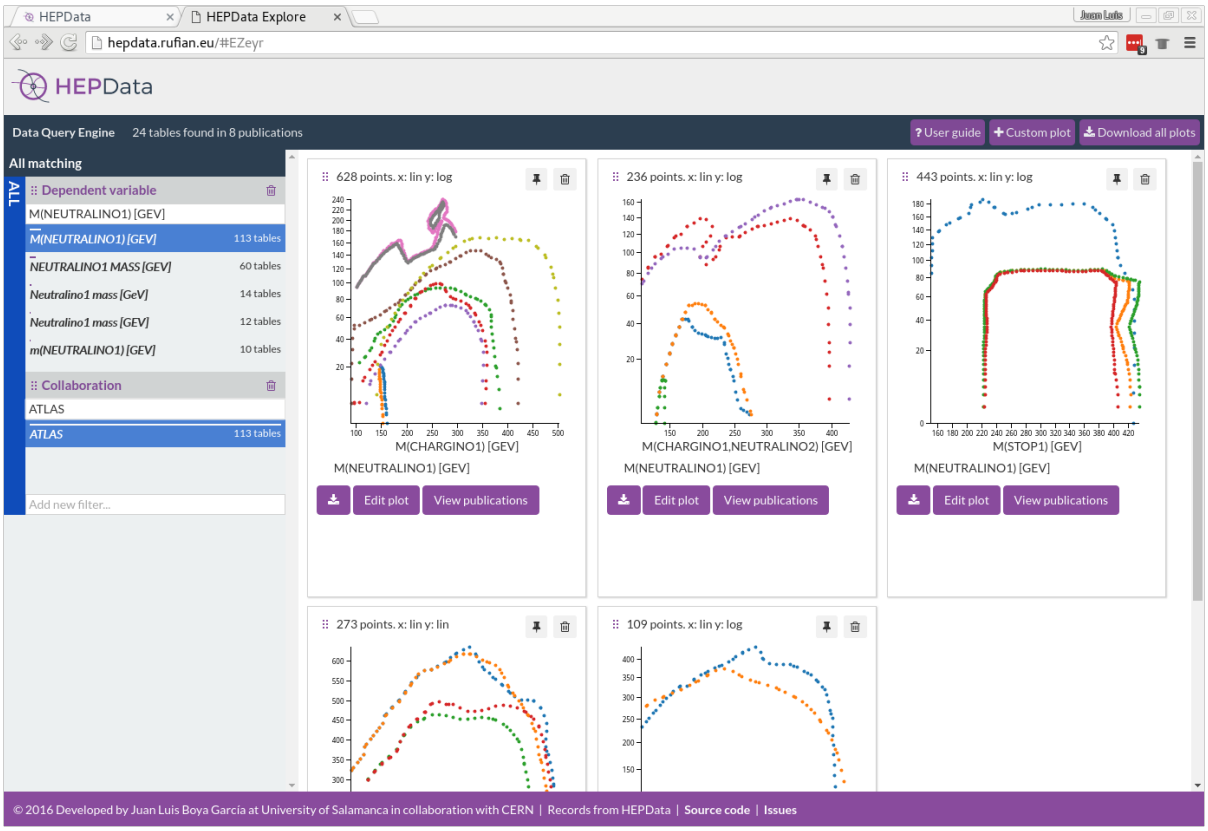

Figura 1.2: Interfaz definitiva de HEPData Explore.

El resto de este documento se estructura de la siguiente forma:

En los dos siguientes capítulos, Personas implicadas y Proceso de desarrollo, se enumeran las personas que han colaborado en el proyecto y se describe el proceso mediante el que se ha desarrollado el trabajo.

A continuación, el capítulo Extracción de datos explica cómo se obtuvieron y decodificaron los registros de HEPData.

El siguiente capítulo, El primer prototipo describe el desarrollo del primer prototipo de la aplicación y como a partir de esta experiencia se orientó el desarrollo posterior.

El capítulo Transformación e indexado introduce ElasticSearch y describe el formato con el que los datos se indexan en el sistema. Para continuar, La nueva interfaz de visualización y consulta describe los aspectos relevantes de la interfaz de usuario de la aplicación. El siguiente capítulo, Persistencia de estados, describe el sistema cliente-servidor empleado para que los usuarios puedan enlazar y compartir búsquedas a través de URLs cortas.

Los dos siguientes temas, Herramientas y técnicas de programación empleadas y Uso de ElasticSearch en HEPData Explore exponen una introducción más avanzada a varias herramientas utilizadas en el desarrollo de la aplicación y son de interés particular de cara a entender mejor el código fuente de la aplicación o crear proyectos similares.

Finalmente, se exponen las Conclusiones y líneas de trabajo futuras y Bibliografía. También se ha elaborado un Glosario de términos donde se explican términos técnicos – tanto informáticos como físicos – utilizados en este trabajo.

En el CD adjunto se incluye una copia en formato digital de esta memoria, el manual de la aplicación, el código fuente de todos los componentes desarrollados y el conjunto de datos utilizado.

CAPÍTULO 2

Personas implicadas

Este proyecto no podría haber tenido lugar sin la cooperación de expertos del dominio, que colaboraron revisando los prototipos, explicando la estructura de la plataforma HEPData y sus formatos de datos y detallando la funcionalidad deseada.

- Eamonn Maguire[4], *CERN*, desarrollador líder de la nueva versión de HEPData.
- Graeme Watt[5], *Universidad de Durham*, administrador de base de datos en HEPData.
- Kyle Cranmer[6], *Universidad de Nueva York*, profesor y físico experimental.
- Lukas Heinrich[7], *Universidad de Nueva York*, estudiante de doctorando de física experimental.
- Mike Whalley[8], *Universidad de Durham*, antiguo administrador de base de datos en HEPData.

Para comunicar a todas las partes implicadas se hizo uso de la plataforma Slack[9] y se establecieron reuniones ocasionales de conferencia de voz por Skype.

[4]https://www.linkedin.com/in/eamonnmaguire1986
[5]http://www.ippp.dur.ac.uk/profile/watt
[6]http://theoryandpractice.org/
[7]http://www.lukasheinrich.com/
[8]http://www.ippp.dur.ac.uk/profile/whalley
[9]https://slack.com/

CAPÍTULO 3

Proceso de desarrollo

En el desarrollo de HEPData Explore se han tenido que atajar muchos problemas.

Uno de los mayores problemas ha sido la carencia de una especificación de requisitos clara, concisa y estática. La idea general del proyecto es engañosamente simple:

> HEPData.net permite a los investigadores encontrar artículos y ver sus datos. Queremos una aplicación que nos permita invertir ese proceso: trabajar a partir de los datos, poder visualizarlos y encontrar los artículos en los que se publicaron.

Detrás de este planteamiento sencillo se esconden muchos problemas. ¿Con qué tipo de condiciones se filtrará? ¿Con qué granularidad debería trabajar la aplicación? ¿Qué entradas introducirá el usuario y qué mostrará la aplicación como respuesta? Son ejemplos de algunas de preguntas sencillas que parece imprescindible responder antes de comenzar un proyecto de este tipo.

Sin embargo, muchas de esas preguntas no tuvieron una respuesta clara hasta casi acabado el trabajo y las respuestas provisionales a las mismas cambiaron a lo largo del desarrollo del proyecto.

La falta de requisitos claros se unió a otros problemas. Uno de los más importantes fue el desconocimiento inicial del dominio. El resto de miembros del grupo habían trabajado con HEPData, bien como desarrolladores, mantenedores o usuarios durante mucho tiempo. Sin ese conocimiento extensivo previo del tipo de información que maneja la plataforma y cómo se estructura, entender las funcionalidades solicitadas y su motivación se complica.

Otro problema fue la latencia de comunicación. La mayoría de miembros del equipo pasan poco tiempo en el canal de chat, por lo que una pregunta cuya respuesta tendría repercusiones

en el desarrollo podía tardar varios días en ser contestada.

Todo esto sin olvidar que detrás de tablas y artículos están datos de un campo muy concreto como es la física de partículas, completamente ajeno para el autor de este trabajo y cuya aprendizaje requiere años de estudio, escapándose totalmente del ámbito del proyecto. Sin entender los problemas a los que se enfrentan los usuarios es difícil plantear una solución.

La combinación de todas estas dificultades hizo mella en el desarrollo del proyecto, que sufrió numerosas pausas y llegaron a afectar a la motivación para continuarlo en varias ocasiones.

3.1 Afrontando los problemas...

Afortunadamente, poco a poco pudieron salvarse los problemas del proyecto.

Después de afrontar el problema de muchas maneras, el punto de inflexión se produjo al afrontarlo desde la perspectiva de un físico.

La mayor aportación en este área se debe a Lukas Heinrich, el más activo de los dos físicos del canal. Lukas tuvo suficiente paciencia para responder a todas las preguntas y explicar la clase de consultas que él necesitaría de la herramienta y cada uno de los conceptos físicos que las motivan.

De las conversaciones con Lukas se extrajeron importantes conclusiones que marcaron la transición desde el prototipo inicial de HEPData Explore a la aplicación final.

3.2 El desarrollo

En este proyecto se ha seguido un proceso de desarrollo iterativo e incremental. Al inicio del proyecto se plantearon unos objetivos generales y se trabajó para producir un entregable que los cumpliera.

Cada versión del trabajo creado se subía a un servidor de desarrollo, http://ntrrgc.rufian.eu/hepdata-explore/, y posteriormente http://hepdata.rufian.eu/. Los miembros del grupo accedían a la aplicación desde estas URLs y comentaban sus impresiones, preguntaban dudas sobre su funcionamiento y sugerían nuevas funcionalidades.

Todo el desarrollo del proyecto se ha hecho en abierto a través del GitHub de HEPData, donde se puede acceder a toda la historia del proyecto:

https://github.com/HEPData/hepdata-explore/

En ocasiones durante el desarrollo se encontraron dificultades no previstas, como información importante que no estaba en los ficheros de datos. En estos casos se comentaba el problema para que los administradores de la plataforma proveyeran soluciones. Debido a la latencia de comunicación en muchos casos fue necesario implementar soluciones provisionales para continuar el desarrollo de la aplicación.

A la hora de hacer el indexado también se encontraron numerosos errores y problemas de homogeneización en los archivos. El programa de indexación incorpora numerosas reglas de homogeneización para tratar con los diferentes formatos usados para algunos campos y muestra un registro de las tablas que tienen problemas que hacen que no puedan ser indexadas.

Este programa también se ha aprovechado para informar a los desarrolladores de problemas de formato hasta ese momento desconocido, pudiendo comunicarles no solo una descripción del problema, sino también listas exactas de publicaciones y tablas afectadas.

3.3 Diseño de interfaces

Durante el ciclo de desarrollo de HEPData se utilizaron prototipos en papel para poder desarrollar una imagen mental de la futura interfaz y evaluar tanto su viabilidad (si es posible implementarla), su completitud (si es posible incorporar las funcionalidades deseadas) como usabilidad (si la interfaz es clara y fácil de comunicar).

Para este propósito se utilizaron tanto wireframes en papel como creados por ordenador.

3.4 Diseño de objetos

Algunas funcionalidades, como los filtros, requirieron la creación de jerarquías de clases. En estos casos se ha hecho uso del lenguaje de modelado UML para poder explorar alternativas y razonar sobre la solución planteada – especialmente su reparto de responsabilidades, de forma más rápida que si se partiera solo del código.

CAPÍTULO 4

Extracción de datos

Antes de poder comenzar a desarrollar el software fue necesario obtener un volcado de las publicaciones de HEPData para disponer de datos con los que alimentar a la aplicación y poder estudiar su estructura.

Al principio del proyecto Eamonn creó una pequeña biblioteca Python para descargar publicaciones de HEPData de forma programática, publicada en GitHub:

https://github.com/HEPData/hepdata-retriever

Como parte del proyecto, a esta biblioteca se le añadió el script `retriever.py` que utiliza la biblioteca anterior para consultar todos los IDs de publicaciones y descargarlas a disco.

En el momento en el que se descargaron estos datos en el ordenador de desarrollo, el conjunto de datos consistía de más de 64.000 tablas pertenecientes a más de 8.000 publicaciones.

Cada publicación se descarga de los servidores de HEPData como un archivo `ins0000.zip` (donde `0000` es un código numérico de publicación) con la siguiente estructura:

- *submission.yaml*: Este archivo declara qué tablas contiene esta publicación y para cada una de ellas indica una serie de metadatos como el resumen y variables de interés físico, como observables, niveles de energía y reacciones.

- `Table1.yaml`, `Table2.yaml`...: Cada uno de estos archivos contiene los datos de una tabla del artículo. En ellos se indican sus variables dependientes e independientes, indicadores (`qualifiers`) con las condiciones del experimento y filas de datos.

Estos ficheros se guardan en un directorio único para cada publicación con el patrón `ins0000`.

El código numérico de publicación se corresponde con el número de registro en INSPIRE[10], una base de datos pública de artículos científicos sobre física de partículas. Actualmente es un pre-requisito que un artículo esté dado de alta en esta base de datos antes de que pueda entrar a HEPData, por lo que se puede asumir que todas las publicaciones tienen este código y que es único.

4.1 Anidando subdirectorios para mejorar el rendimiento

Aun guardando cada publicación en un directorio diferente, tener más de 8000 subdirectorios en una ruta causa serios problemas de rendimiento con la mayoría de sistemas de ficheros actuales.

Por ejemplo, hacer un `ls` en la consola con un sistema de ficheros EXT4 en Linux requiere internamente no solo listar el contenido del directorio sino también acceder a los más de 8000 inodos de cada uno de los archivos enumerados para poder conocer su tipo, tamaño, fecha de modificación, etc. Puesto que estos inodos pueden residir en ubicaciones aleatorias, la operación puede tardar fácilmente medio minuto en completarse en un disco duro tradicional, durante el cual la consola permanece bloqueada.

Para evitar esto se hace uso de una agrupación mediante una función hash. Los directorios se guardan en dos niveles, donde el nivel más profundo tiene el nombre del directorio original, mientras el nivel más alto representa la salida de una función hash que recibe el nombre del directorio original como entrada.

Concretamente en esta aplicación la función hash consiste en extraer los dos últimos caracteres numéricos del nombre del fichero, de manera que el directorio `ins917921` se guarda en `21/ins917921`. Puesto que hay 100 posibles valores para esta función hash, el número de ficheros por directorio se reduce (de media) a una centésima parte.

Esta simple estrategia logró reducir los tiempos de las operaciones de listado drásticamente, haciéndolos casi instantáneos.

4.2 Indicador de progreso

Al script de carga se le añadió una barra de progreso para que el usuario que carga los datos pueda saber que el script no se ha quedado atascado. Esta barra, además del porcentaje

[10] https://inspirehep.net/

de progreso, indica el código de publicación que se está procesando, de manera que si una publicación por alguna razón se atasca, es posible saber cuál.

Esta funcionalidad resultó ser útil, puesto que durante el desarrollo se encontraron dos publicaciones que por alguna razón, el servidor nunca terminaba de retornarlas. En ese momento se modificó el script para ignorar ambas y la anomalía fue reportada para su corrección.

```
Warning: Rejected table. ins283744, Table 12. Reason: No valid independent variables.
Warning: Excluded independent variable "N(P=4)" on ins283744, Table13.yaml. Reason: NotNumeric: >= 20
.0
Warning: Rejected table. ins283744, Table 13. Reason: No valid independent variables.
Warning: Excluded independent variable "N(P=4)" on ins283744, Table14.yaml. Reason: NotNumeric: >= 20
.0
Warning: Rejected table. ins283744, Table 14. Reason: No valid independent variables.
Warning: Excluded independent variable "N(P=4)" on ins283744, Table15.yaml. Reason: NotNumeric: >= 20
.0
Warning: Rejected table. ins283744, Table 15. Reason: No valid independent variables.
Warning: Excluded independent variable "N(P=4)" on ins283744, Table16.yaml. Reason: NotNumeric: >= 20
.0
Warning: Rejected table. ins283744, Table 16. Reason: No valid independent variables.
Warning: Excluded independent variable "PT(P=3)**2 (GEV**2)" on ins376444, Table1.yaml. Reason: NotNu
meric: < 0.075
Warning: Excluded dependent variable "SIG(NAME=COHERENT DIFFRACTIVE PRODUCTION) (NB)" on ins376444, T
able1.yaml. Reason: NotNumeric: <220.0
Warning: Rejected table. ins376444, Table 1. Reason: No valid independent variables.
Warning: Excluded dependent variable "SIG (NB)" on ins376444, Table2.yaml. Reason: NotNumeric: <120.0
Warning: Rejected table. ins376444, Table 2. Reason: No valid dependent variables.
42% ins417044  [##################################                                    ]
```

Figura 4.1: `server-aggregator` haciendo su trabajo. Además de la barra de progreso, el programa muestra los errores que encuentra en las publicaciones cuando estos impiden indexarlas total o parcialmente.

En la figura 4.1 se puede ver el aspecto del programa en ejecución:

4.3 Metadatos de la publicación

Algunos datos importantes de los artículos como su título, resumen, autores y la colaboración (grupo de investigación) que lo ha publicado no aparecen en el fichero `submission.yaml`.

Esto se debe a que en HEPData, la información básica de los artículos y las tablas de datos se procesa y almacena de forma separada.

Eventualmente esto se convirtió en un problema porque algunas personas querían ser capaces de filtrar por datos de la publicación, especialmente por el nombre de la colaboración.

Para poder obtener también esta información, Eamonn Maguire habilitó un endpoint en

hepdata.net que permite consultar estos datos.

https://hepdata.net/record/ins1442359?format=json&light=true

Se creó otro script para descargar esta información en un archivo con nombre `publication.json` dentro del directorio de cada artículo.

Se hizo uso de paralelismo, con 64 tareas simultáneas, puesto que el endpoint es lento (600 ms) pero muy paralelizable. De esta manera se redujo el tiempo de carga de una hora y media estimada en el caso secuencial a tan solo cinco minutos en el caso paralelo.

4.4 Porcentaje de éxito

A pesar de todos los esfuerzos de normalización, no es posible incorporar todas las tablas de HEPData en HEPData Explore. Hay muchas que no tienen un formato numérico, o tienen un formato difícil de representar de forma general (ej. rango de un número a infinito).

En estos casos `server-aggregator` ignora en la medida de lo posible las tablas afectadas y sigue buscando tablas con datos indexables.

Al final de su ejecución, el programa muestra el porcentaje de tablas cuyos datos pudieron ser normalizados y extraídos. Con el conjunto de publicaciones utilizado en este trabajo solo se descarta un 5.93 % de las tablas procesadas.

```
100% ins98899    [##################################################]
Done
Indexed 8194 submissions.
Scanned 64817 tables, rejected 3845 tables (5.93%).
```

CAPÍTULO 5

El primer prototipo

5.1 Propuesta

Eamonn Maguire propuso un mockup del aspecto que podría tener HEPData Explore.

En él se distinguían tres filtros:

- Filtro por variable independiente.
- Filtro por tipo de reacción.
- Filtros por rango de valores de varias variables dependientes.

En la interfaz se distingue un gráfico circular mostrando los tipos de reacción encontrados y un histograma para cada variable dependiente mostrando su distribución de valores.

Eamonn recomendó usar las biblioteca crossfilter.js[11] para el filtrado de datos en el lado del cliente y dc.js[12] para la creación de gráficos.

El mockup no pretendía de forma alguna ser definitivo, pero sirvió como guía para marcar el aspecto y tipo de funcionalidad que tendría el proyecto.

[11] https://github.com/square/crossfilter
[12] https://dc-js.github.io/dc.js/

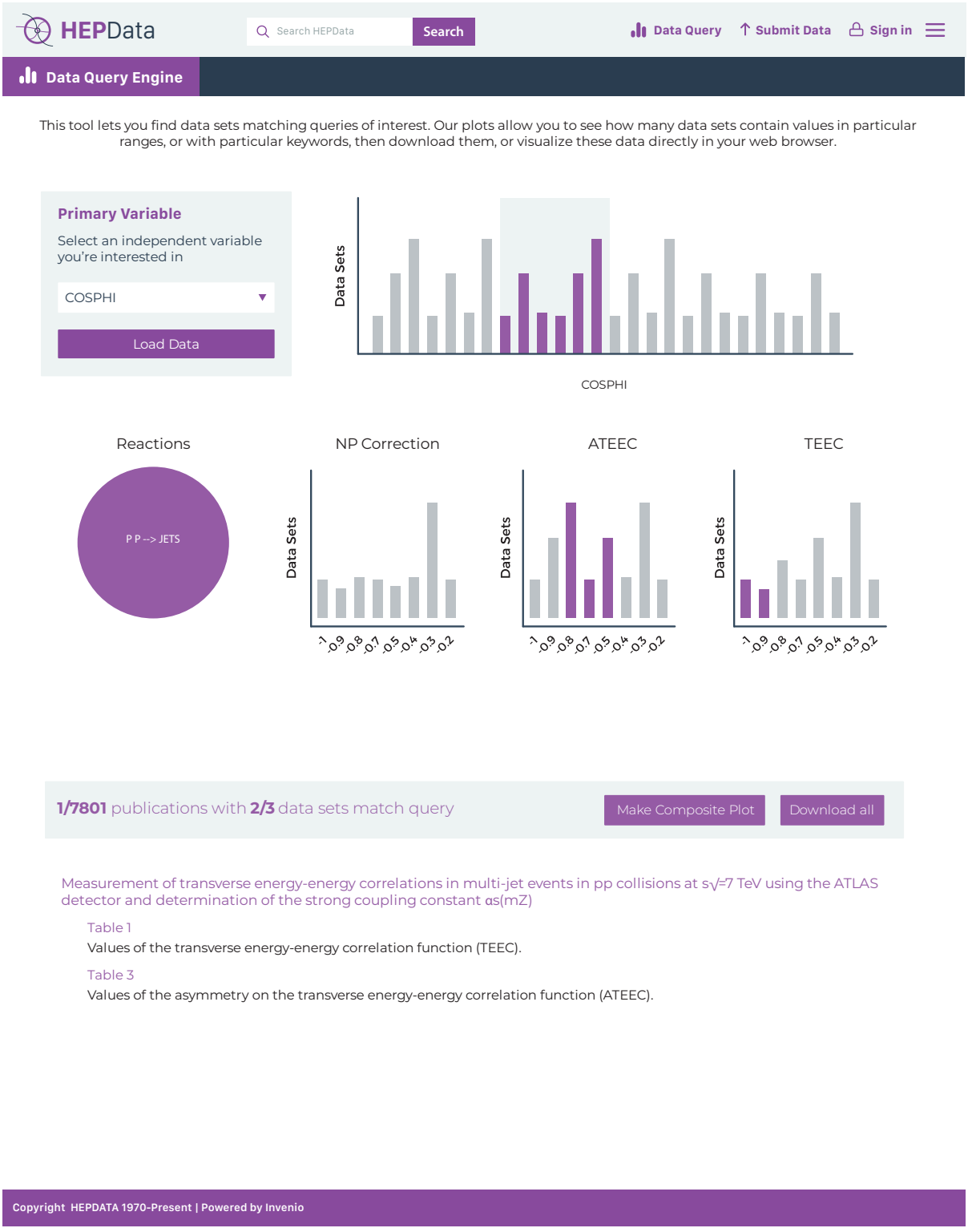

Figura 5.1: El primer mockup de HEPData Explore.

5.2 Primer paso: Diseño de índices

Para poder mostrar los datos de varios artículos filtrados por sus atributos primero es necesario indexarlos de acuerdo a cómo van a ser consultados.

La indexación es el proceso consistente en almacenar información en estructuras de datos que permitan la recuperación de parte de los datos de acuerdo a unos determinados criterios.

En una aplicación web la indexación puede hacerse en el lado del cliente o en el lado del servidor.

Cuando se hace en el lado del servidor, una máquina remota aloja los índices y los datos y atiende a peticiones remotas, leyendo sus criterios de filtrado y enviando al cliente datos que los cumplan.

Cuando la indexación se hace en el lado del cliente, el navegador del propio usuario descarga todos los datos, crea índices y los utiliza para atender las consultas del usuario. En consecuencia, la interacción es mucho más fluida, puesto que no es necesario esperar el tiempo de retorno (*roundtrip*) desde que se establecen los criterios de filtrado hasta que se recibe la respuesta de un servidor con los datos filtrados. Por contra, es necesario descargar la base de datos completa al ordenador del usuario, lo cual no es práctico cuando ésta alcanza un cierto tamaño.

En este prototipo se estableció una solución de compromiso, con un indexado en dos niveles:

- **Primer nivel**, en el lado del servidor: Los datos se indexan agrupados por variable independiente.
- **Segundo nivel**, en el lado del cliente: Los datos se filtran por rango de energía y rango de valores.

En un sistema híbrido de este tipo, modificar el filtro del primer nivel es una tarea relativamente lenta puesto que requiere una consulta al servidor y descargar un conjunto de datos mayor que si el indexado se hiciera completamente en el lado del servidor. Por otro lado, modificar los filtros del lado del cliente es instantáneo, puesto que la búsqueda y filtrado se hace en el mismo ordenador del usuario, y debido al primer nivel de filtro el conjunto de datos es relativamente pequeño.

Los filtros con menor interacción son más adecuados para aparecer en el lado del servidor, y viceversa. En este caso la variable independiente es un filtro de selección de uno entre múltiples valores y será modificada de forma poco frecuente, mientras que los otros dos son

de rango y se espera un mayor número de interacciones con ellos.

5.3 Segundo paso: aplanamiento

Al intentar llevar a la práctica este prototipo con las herramientas propuestas se llegó a una limitación muy importante de crossfilter.js: solo es capaz de indexar estructuras de datos planas, con un número de columnas predefinido, similar a una estructura tabular.

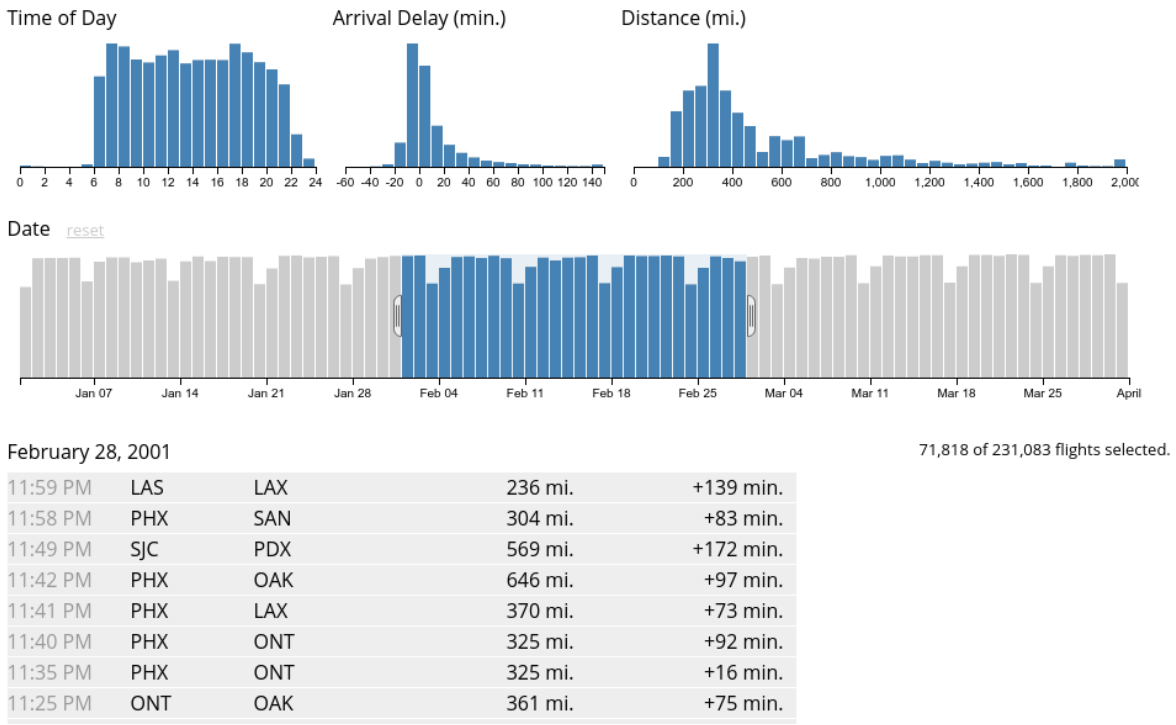

Figura 5.2: Captura de la página de demostración de crossfilter.js y dc.js, con un conjunto de datos tabular.

Esto es un problema porque los datos con los que trabaja el prototipo pertenecen a diferentes tablas de muchos artículos, y cada una puede tener un conjunto de variables distinto. También se dan con relativa frecuencia casos de tablas donde hay varias variables independientes.

No parece una solución adecuada crear una columna de crossfilter.js para cada variable de cada tabla, puesto que puede haber un gran número de ellas y la mayoría de ellas no se usarán. Internamente crossfilter.js crea un índice en memoria para cada columna, por lo que el número de columnas declaradas tiene un gran impacto en el rendimiento y en el consumo de memoria de la aplicación. Además, hay un límite máximo de 64 columnas a partir del cual no acepta

más porque se queda sin bits en las estructuras internas de sus índices.

Por otro lado, el diseño de este prototipo trabaja con pares de valores x e y, donde x siempre es el valor de una variable independiente e y siempre es el valor de una variable dependiente.

Se aprovechó esta circunstancia para sortear la limitación de crossfilter.js transformando los datos a un formato aplanado.

La siguiente tabla muestra lo que podría ser una tabla original sin aplanar, con una sola fila.

Pub. 150, Table 10			
Variables independientes		Variables dependientes	
A	B	C	D
1	2	3	4

Después de aplicar la transformación de aplanado, la tabla anterior se convertiría en la siguiente:

Publicación	Tabla	var_x	var_y	x	y
Pub. 150	10	A	C	1	3
Pub. 150	10	A	D	1	4
Pub. 150	10	B	C	2	3
Pub. 150	10	B	D	2	4

Este diseño tiene importantes consecuencias:

- La estructura resultante es tabular, por lo que puede ser introducida en crossfilter.js sin ningún problema.
- Los nombres de variables pasan de ser columnas a ser valores de un campo.
- La transformación se hace con pérdida. Puesto que la transformada devuelve pares, es fácil saber qué pares de valores aparecen juntos en dos variables distintas, pero no se puede hacer el mismo cálculo con tres o más variables.
- La transformación plana contiene mucha redundancia. Los metadatos a nivel de tabla y publicación tienen que ser introducidos en cada fila.
- El número de filas es mucho mayor que en la tabla original. Concretamente es $filas' = filas \cdot indeps \cdot deps$.
- Puesto que está definido que los pares (x, y) deben ser de una variable independiente y

una variable dependiente respectivamente, no es posible hacer asociaciones entre varias variables del mismo tipo – por ejemplo, ver qué pares de valores de dos variables independientes aparecen juntos.

El proceso de transformación podría modificarse para soportar estas asociaciones, aunque el nivel de redundancia aumentaría significativamente. Siendo $N = indeps \cdot deps$, serían necesarias $filas' = filas \cdot N \cdot (N-1)$ para cubrir los mismos datos.

5.4 Tercer paso: Almacenamiento y consulta de datos

A partir de la estructura plana definida en el punto anterior y los requisitos de consulta especificados se diseñó una estructura de datos para almacenar e indexar la información.

Puesto que la consulta en el lado del servidor solo abarcaba un nivel de filtro, se optó por un índice simple basado en archivos: Antes del primer uso del sistema, y cada vez que se agreguen datos se ejecutaría un programa, `flatten.py` que escanearía todas las tablas de HEPData.

Para cada tabla, este programa la aplana y escribe sus filas en un archivo determinado según la variable independiente a la que hace referencia. Al mismo tiempo se mantiene un registro de cuántas filas se han escrito para cada variable independiente de manera que se pueden ordenar en la interfaz por número de registros.

Como optimización, todas las columnas de la tabla que en la tabla original no son exclusivas de una fila (ej. datos de la publicación, número de tabla, variable dependiente..., todo excepto los valores de esa fila) se almacena de forma en una estructura de *grupo*, con lo que se evita redundancia y el archivo resultante es significativamente más pequeño. El grupo consiste de una cabecera con los valores para número de tabla, variable dependiente etc. que son comunes a todo el grupo y una lista de filas con la información tabular sin las columnas redundantes.

Una vez descargado, el cliente recompone la tabla original plana en memoria. Sin embargo, esta optimización reduce significativamente el tamaño de la descarga, lo cual se traduce en una mejora de notable del rendimiento puesto que el uso de la red suele ser es más lento y costoso que el preprocesado en la CPU.

Una vez generados los ficheros de índice, la consulta desde el cliente es muy sencilla: basta con determinar la URL del fichero que contiene las tablas con la variable independiente, descargarlo y decodificarlo.

Un aspecto a destacar de este diseño es que no requiere software servidor especializado,

cualquier servidor de archivos estático sirve. Esto tiene consecuencias muy interesantes:

- El rendimiento es el más alto posible, puesto que los resultados de todas las consultas posibles ya están precalculados y solo hay que enviarlos al cliente.
- Al no haber código web dinámico, la posibilidad de encontrar vulnerabilidades de seguridad se reduce a la mínima expresión.
- La escalabilidad es prácticamente lineal. Añadir un segundo servidor sólo requiere copiar los archivos y añadir una entrada en la zona DNS.
- El despliegue es muy simple y se puede hacer en prácticamente cualquier servidor.

5.5 Cuarto paso: interfaz

Una vez indexados los datos por el filtro primario (variable independiente) y accesibles por HTTP es posible implementar una interfaz de usuario para visualizarlos.

La interfaz implementada se muestra en las figuras 5.3 y 5.4 y es accesible a través de la siguiente URL:

http://ntrrgc.rufian.eu/hepdata-explore [13]

Esta interfaz se basa en el mockup de Eamonn, con algunas modificaciones:

- El gráfico de tarta se ha sustituido por un gráfico de barras para aprovechar mejor el espacio. Es difícil colocar cadenas de texto tan largas como las utilizadas en la aplicación en un gráfico de tarta.
- Los diagramas de variables dependientes eran histogramas en el mockup. Aquí se han sustituido por diagramas de dispersión de manera que es posible ver no solo cuantos datos hay, sino los propios datos.

La colocación de los gráficos se hace de manera automática: se agrupan los datos filtrados por variable dependiente y ordenan por número de registros. Las diez variables dependientes con mayor número de registros se llevan un gráfico cada una.

Los colores de los gráficos de dispersión sirven para distinguir datos de varias tablas con las

[13] Esta URL no contiene la versión final de la aplicación, sino solo el prototipo del que se habla en este capítulo. La versión final es accesible desde http://hepdata.rufian.eu/.

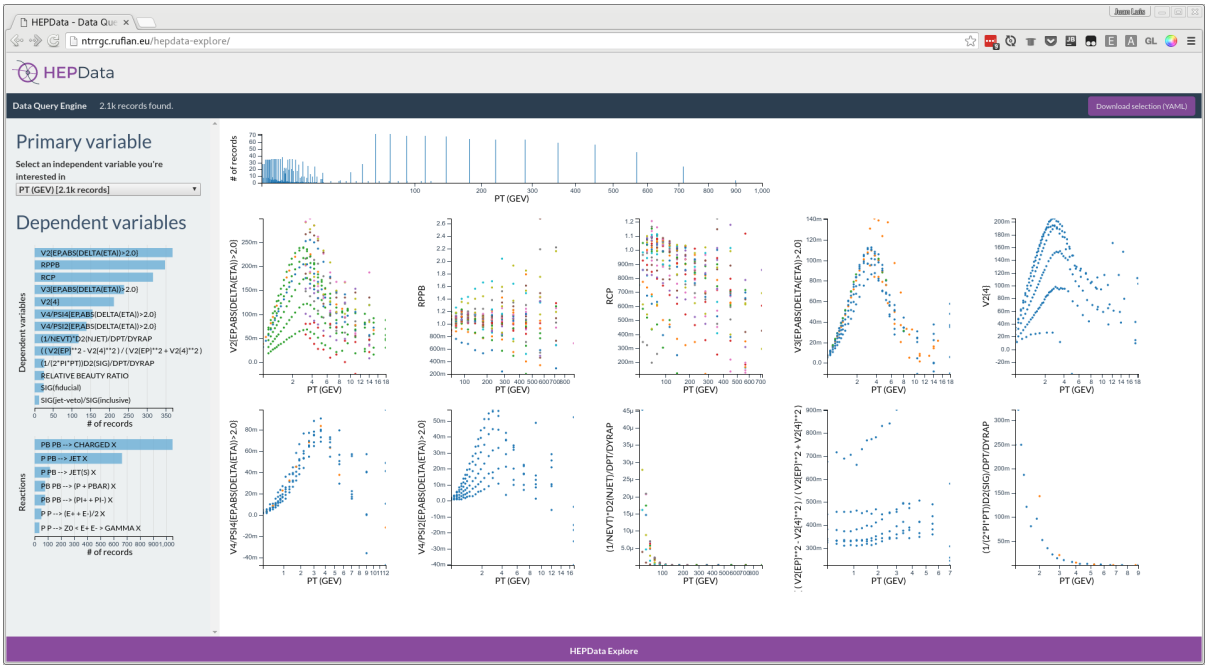

Figura 5.3: La interfaz del prototipo de HEPData Explore.

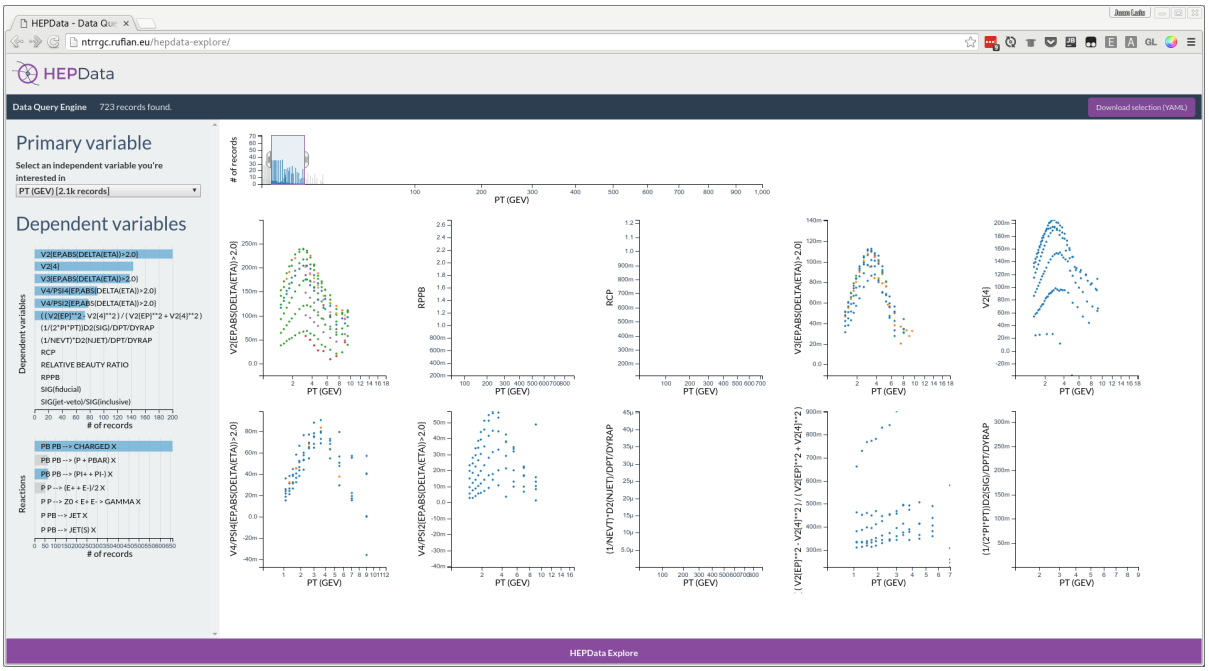

Figura 5.4: Filtrado en el lado del cliente. Los datos que no pasan el filtro desaparecen de los diagramas de dispersión.

mismas variables.

Es posible utilizar filtros en el lado del cliente para ver datos filtrados. Por ejemplo, haciendo clic y arrastrando con el ratón en el histograma superior se pueden ver registros cuyo valor de la variable independiente seleccionada está en un determinado rango. También es posible seleccionar una o más barras en el diagrama de reacciones para filtrar los datos con datos sobre las mismas.

5.6 Problemas y evolución hacia la versión actual

La aplicación mostrada hasta ahora cumplía la mayoría de funcionalidad del mockup de Eamonn, con las notable excepción de la lista de publicaciones, que aún estaba por implementar.

Sin embargo, después de una reunión de proyecto quedó una impresión de que la aplicación era más atractiva que útil para las necesidades de los físicos nucleares.

En particular, de forma contraria a lo planteado en el mockup inicial, se pedía que el filtro primario pudiera ser no solo una variable independiente sino también una variable dependiente, un observable, una colaboración, etc.

La primera reacción fue intentar adaptar el sistema creado para soportar nuevos filtros primarios. El número de posibles filtros primarios tiene un impacto en el tamaño del índice, el tiempo de indexado y la complejidad general del sistema, por lo que debían ser escogidos cuidadosamente.

No se consiguió una respuesta clara a la pregunta de qué tipo de filtros sería deseable soportar y cuáles podrían ser descartados, pero quedó claro que existía un deseo de que el sistema fuera *lo más flexible posible*.

Para hacer frente a esta demanda, el sistema fue repensado desde la base.

5.6.1 Ideas iniciales

En primer lugar, el índice que construía el script `flatten.py` era simple, eficiente y muy fácil de desplegar, pero no era suficiente para cumplir las demandas de los usuarios.

Para soportar otros tipos de filtros primarios sería necesario crear muchos más índices. Incluso

si se crearan, si un usuario quisiera hacer consultas compuestas – por ejemplo, obtener las tablas donde se cumplan no una sino dos condiciones) requeriría implementar un servidor de búsqueda, lo que eliminaría la facilidad de despliegue conseguida con el índice simple y sería difícil implementar desde cero.

La solución encontrada para este problema consta de dos partes.

5.6.2 Un servidor de búsqueda

En primer lugar, se decidió utilizar un servidor de búsqueda genérico, de manera que no hubiera que implementar uno de cero, lo que alargaría el proyecto demasiado, pero que fuera lo suficientemente flexible para tratar con los datos de HEPData de manera eficiente.

El servidor de búsqueda escogido fue ElasticSearch[14].

En realidad, la posibilidad de usar ElasticSearch ya se había contemplado antes de escribir el script `flatten.py`, pero en aquel momento fue descartado por ser demasiado compleja para los requisitos planteados inicialmente. Con este cambio de requisitos, esta premisa dejó de ser cierta.

La migración del sistema inicial hacia ElasticSearch requirió un esfuerzo notable. El capítulo siguiente describe cómo se lograron indexar los datos de las publicaciones y hacer las consultas con el nuevo motor de búsqueda. A pesar de la dificultad inicial, la migración mereció la pena.

El nuevo sistema no solo soporta filtros arbitrariamente complejos y responde con rapidez suficiente, sino que gracias a ElasticSearch la gestión de actualizaciones es mucho más potente. Actualizar una publicación en el nuevo sistema solo requiere hacer una petición al servidor de búsqueda que se resuelve en milisegundos, mientras que en el sistema anterior se requería reindexar todas las publicaciones.

El siguiente capítulo, Transformación e indexado, explica como se indexaron los datos con el nuevo servidor de búsqueda.

5.6.3 Una nueva interfaz

La segunda parte de la solución consiste en una nueva interfaz de filtrado que sea lo suficientemente flexible como para expresar todas las condiciones que los usuarios puedan

[14] http://elastic.co/

demandar. El capítulo La nueva interfaz de visualización y consulta explica cómo se diseñó y construyó esa interfaz.

CAPÍTULO 6

Transformación e indexado

En el capítulo Extracción de datos se consiguió cargar toda la información de las publicaciones de HEPData en el disco duro local del ordenador de desarrollo. Disponer de una copia de esta información en disco local es imprescindible para poder experimentar con diferentes formatos de indexación de forma simple y eficiente.

Los usuarios de la aplicación van a querer consultar subconjuntos de datos que cumplan ciertos criterios. Estos criterios son enviados a un *servidor de búsqueda*, el cual utiliza *índices* para localizar los artículos que los cumplen de forma eficiente.

6.1 El servidor de búsqueda: ElasticSearch

ElasticSearch[15] es un servidor de búsqueda basado en Apache Lucene[16], distribuido como software libre bajo la licencia Apache.

ElasticSearch funciona como una base de datos con una interfaz REST. ElasticSearch define los siguientes conceptos, cada uno de los cuales tiene una asociación con conceptos de bases de datos:

- *Índice*, se corresponde con una base datos. Una misma instancia de ElasticSearch puede almacenar varios índices.
- *Tipo*, se corresponde con una colección o tabla.

[15] https://www.elastic.co/
[16] http://lucene.apache.org/

- *Documento*, se corresponde con un registro de una tabla. Los documentos en ElasticSearch se codifican en JSON y soportan estructuras de datos anidadas.
- *Campo*, se corresponde con una columna de una tabla. Por defecto en ElasticSearch todos los campos tienen índices (entendidos por la definición de las bases de datos tradicionales: estructuras de datos que permiten la recuperación de información sin necesidad de recorrer una tabla entera de principio a fin).
- *Mapeo* o *mapping*, define la estructura de un tipo, qué campos acepta y cómo se indizarán. En una base de datos se corresponde con el esquema.

La API admite endpoints REST como `POST /hepdata/publication/1234` para crear una nueva publicación con `id` 1234.

6.2 El mapeo

Definir mapeos en ElasticSearch es opcional y se puede evitar en tipos simples. Sin embargo, algunas funcionalidades como los documentos anidados o la posibilidad de deshabilitar el análisis textual para ciertos campos no se pueden usar sin mapeo explícito, por lo que suele ser necesario cuando avanza el proyecto. Este ha sido el caso con HEPData Explore.

En HEPData Explore hay solo un tipo (colección): `publications`. Este objeto contiene todos los datos de una publicación que pueden ser consultados o filtrados en el sistema, desde su título hasta cada uno de los errores estadísticos de un dato de de una tabla.

El mapeo se carga en ElasticSearch como un objeto en formato JSON con una llamada `POST /hepdata/publication/_mapping`.

En el archivo `server-aggregator/aggregator/record_aggregator.py` se puede ver el contenido completo del mapeo, en el método `init_mapping()` de `RecordAggregator`.

6.2.1 Tipos de indexamiento

Analizado o no analizado

El mapeo también es el lugar donde se puede especificar de manera opcional cómo debe indizarse cada campo.

En particular, los campos de tipo texto (`string`) por defecto son *analizados* como texto natural en tiempo de indexación. Esto significa que se dividen en palabras, se extraen los lexemas y se filtran símbolos y *stop words* del inglés.

Si bien en algunos campos, como el título o el resumen, este comportamiento es útil, en otros muchos, como los nombres de variables, es indeseable. En estos campos es posible especificar `"index": "not_analyzed"` para que ElasticSearch los indexe como entidades completas. Es decir, `P (GEV)` sólo será devuelto si se busca exactamente `P (GEV)`.

Indexado deshabilitado

En algunos casos realmente no nos interesa que un campo esté indexado. Es el caso del contenido de las tablas (filas): Se filtra por las cabeceras de las tablas, no por los valores de sus filas. Por otro lado, las filas tienen muchos valores para cada publicación.

Incluso si quisiéramos hacer consultas sobre los datos de las filas en ElasticSearch, es muy poco probable que pudiéramos hacer consultas útiles sin hacer grandes cambios en el mapeo.

En conclusión, si permitimos que se indexen estaremos gastando espacio de disco y tiempo de indexado sin ganar realmente nada. Afortunadamente ElasticSearch permite deshabilitar el indexado para estos casos especificando en la configuración del campo `"enabled": false`.

Nótese que esta propiedad solo deshabilita el indexado, lo cual impide las búsquedas por el contenido de ese campo, pero los datos todavía son almacenados y recuperados como resultado de una consulta que filtre por otros campos.

6.2.2 Contenido del mapeo

El mapeo utilizado tuvo varios cambios durante el desarrollo de la aplicación, según el conocimiento del problema y los requisitos fueron variando.

El cambio más notable fue la eliminación del aplanado en tiempo de indexado, que dejó de ser necesario en el nuevo sistema, simplificando el mapeo y las consultas.

A continuación se define de forma textual el contenido de la última versión del mapeo.

- `title`: Título de la publicación, indexado como texto natural.
- `title_not_analyzed`: Título de la publicación, convertido a minúsculas e indexado sin

analizar. Este campo se utiliza para búsquedas con expresiones regulares.

- `comment`. Comentario de los datos que el autor puede añadir en el momento de enviar la publicación a HEPData.
- `comment_not_analyzed`. Comentario de los datos, convertido a minúsculas e indexado sin analizar.
- `collaborations`. Lista de colaboraciones que participaron en el artículo. Indexado como texto sin analizar. En los artículos nuevos suele tener uno solo de los siguientes valores: `ATLAS`, `ALICE`, `CMS` o `LHCb` – que se corresponden a los mayores grupos de investigación que están trabajando actualmente en el LHC, aunque hay muchos valores, muchos pertenecientes a grupos de investigación antiguos.
- `publication_date`. Fecha en la que se publicó el artículo.
- `inspire_record`. Código único del artículo, indexado como un número entero.
- `hepdata_doi`. Código DOI de la publicación. Estos códigos son utilizados para citar las tablas en los trabajos de investigación de los físicos.
- `version`. Número de versión de la publicación. Este número se incrementa cuando el personal de HEPData hace cambios en los datos de una publicación, por ejemplo, para corregir errores de formato.
- `tables`. Tipo anidado. Cada una de las tablas del artículo.
  - `table_num`. Número secuencial que identifica la tabla dentro del artículo.
  - `description`. Explicación textual de qué está codificado en la tabla, indexada como texto natural.
  - `description_not_analyzed`. Explicación textual de qué está codificado en la tabla, convertida a minúsculas e indexada sin analizar. Utilizado en búsquedas con expresiones regulares.
  - `cmenergies_min`. Nivel de energía, en GeV. Si en una misma tabla se han hecho mediciones con diferentes niveles de energía, este campo indica el menor de ellos.
  - `cmenergies_mmax`. Nivel de energía, en GeV. Si en una misma tabla se han hecho mediciones con diferentes niveles de energía, este campo indica el mayor de ellos.

- `reactions`. Describe las reacciones producidas en el experimento, es decir, qué partículas iniciales son reemplazadas por qué partículas de destino durante la interacción en el acelerador. Se indexa como un objeto con las siguientes propiedades, todas de tipo cadena no analizada:
  - `particles_in`. Lista de partículas iniciales, definidas como cadenas. Por ejemplo, `['GAMMA', 'P']`.
  - `particles_out`. Lista de partículas finales, definidas como cadenas. Por ejemplo, `['P', 'PI+', 'PI-']`
  - `string_in`. Partículas iniciales, definidas como una cadena para poder ser filtrada con expresiones regulares. Por ejemplo, `GAMMA P`
  - `string_out`. Partículas finales, definidas como una cadena para poder ser filtrada con expresiones regulares. Por ejemplo, `PI+ PI+ PI- PI- P`
  - `string_full`. Cadena con la reacción completa, tal como se almacena en HEPData. Por ejemplo, `GAMMA P --> PI+ PI+ PI- PI- P`.
- `observables`: Definen fenómenos físicos observados en el experimento.
- `phrases`: Etiquetas normalizadas que describen el tema estudiado en una tabla.
- `indep_vars`: Lista de variables independientes. Cada variable se define como un objeto con una única propiedad `name` que indica el nombre de la variable. Este campo se introduce como objeto por similaridad con `dep_vars` (explicado abajo) y para que en versiones posteriores sea fácil añadir nuevos campos sin necesidad de cambiar todas las consultas existentes. Posibles nuevos campos podrían ser las unidades o un nombre de variable homogéneo.
- `dep_vars`: Lista de variables dependientes. Cada variable se define como un objeto, con las siguientes propiedades:
  - `name`: Nombre de la variable.
  - `qualifiers`: Representan condiciones de la variable medida. Estos calificadores son lo que distingue dos variables distintas que utilizan las mismas unidades. Se almacena como un array de objetos, cada uno con una propiedad `name` y `value`.
- `data_points`. Lista con cada una de las filas de la tabla. No se indexa. Cada fila se

representa como una lista de objetos. Cada objeto está asociado con una columna. Estos objetos están ordenados de manera que si las variables independientes son A y B y las variables dependientes son C y D, cada fila de datos estará compuesta por [A, B, C, D].

- `value`: Valor de la columna en esta fila, definido como un número real o `null` en caso de que no haya valor.
- `low` y `high`: Algunas columnas definen rangos en vez de valores puntuales. En ese caso se utilizan estos campos y `value` no se especifica.
- `errors`: Lista de errores. Cada uno se define como un objeto con las siguientes propiedades:
  - `label`: Etiqueta de error, indica su naturaleza. `sys` se refiere a error sistemático, `stat` se refiere a error estadístico. Si un error en el conjunto de datos original no estaba etiquetado, recibe la etiqueta automática `main`.
  - `type`: Puede ser `symerror` (error simétrico) o `asymerror` (error asimétrico).
  - `value`: Para los errores simétricos, define la magnitud del error.
  - `plus` y `minus`: En los errores asimétricos, define la desviación positiva y negativa respectivamente.

6.3 Las consultas de búsqueda

ElasticSearch utiliza un lenguaje específico de dominio para definir consultas de búsqueda.

Un punto muy fuerte de este lenguaje es que está basado en JSON, un lenguaje de serialización jerárquico fácil de manipular y que permite anidar elementos fácilmente. Este punto fue clave para la implementación de los filtros.

Gracias a la estructura jerárquica de JSON basada en objetos, cada filtro de la interfaz tiene su propia función de transformación que devuelve una consulta de ElasticSearch con los resultados que le corresponden. Esta salida, puede ser a su vez tratada por filtros compuestos (*all* y *some*) para implementar operaciones lógicas.

Cada filtro es independiente de los demás y debido a su estructura objetual, toda una clase entera de errores de escapado y validación es eliminada de forma automática.

6.3.1 Herramientas de soporte

Por otro lado, ElasticSearch tiene una curva de aprendizaje no despreciable, en especial a la hora de trabajar con estructuras anidadas, como es el caso en HEPData, donde hay unas estructuras padres (publicaciones) y otras hijas (tablas) y es necesario filtrar por los datos de ambas.

Además, la sintaxis del lenguaje de consultas, aunque potente y ventajosa a la hora de implementarla en una aplicación, es realmente verbosa. Esto hace que experimentar con ElasticSearch a través de clientes simples como cURL[17] se vuelva incómodo, especialmente a la hora de identificar errores.

En este aspecto, durante la experimentación con ElasticSearch fue de gran ayuda Sense, un cliente de ElasticSearch construido precisamente para facilitar la realización de consultas de prueba durante el desarrollo de aplicaciones de búsqueda.

Sense conoce la sintaxis de ElasticSearch y es capaz de ofrecer autocompletado y validación antes de ejecutar la consulta. Una pulsación de teclas es suficiente para obtener en el panel adyacente el resultado de la consulta en JSON automáticamente sangrado.

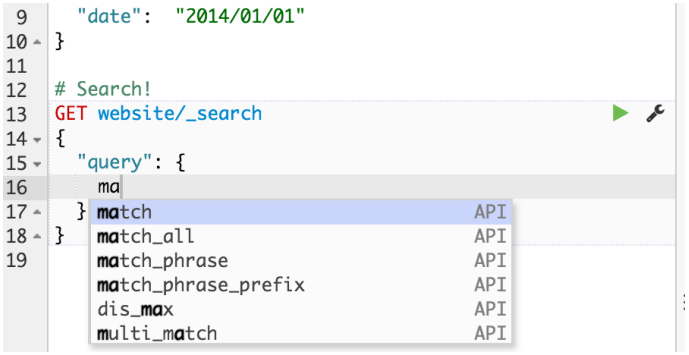

Figura 6.1: Autocompletado de Sense.

También fue útil crear una base de datos en miniatura con datos de prueba, con tan solo dos publicaciones con dos artículos cada una, para probar que las consultas efectivamente devuelven los resultados que coinciden con los filtros y que no devuelven aquellos que no corresponden.

[17] https://curl.haxx.se/

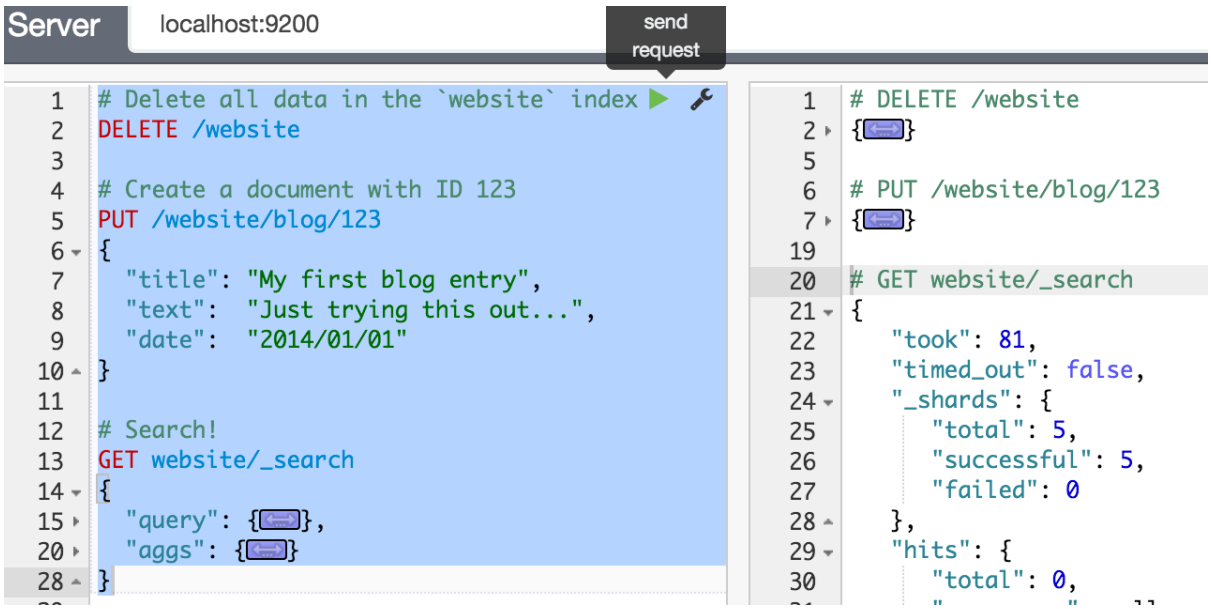

Figura 6.2: Haciendo peticiones a ElasticSearch y obteniendo respuestas con Sense.

6.3.2 Construcción de consultas

La aplicación transforma los filtros de la interfaz de usuario en consultas con el formato del DSL de ElasticSearch, las envía al servidor y procesa su respuesta para extraer el contenido de las tablas a partir de las que genera los gráficos.

La construcción de estas consultas no es trivial. El capítulo Uso de ElasticSearch en HEPData Explore explica de forma detallada cómo se pueden construir consultas de ElasticSearch para crear una aplicación como HEPData Explore.

CAPÍTULO 7

La nueva interfaz de visualización y consulta

Solucionado el indexado de los datos, el siguiente paso para tener una aplicación usable es crear una interfaz de usuario. La interfaz de usuario suele ser la parte más compleja de muchas aplicaciones y este caso, tratándose específicamente de una aplicación de visualización de datos, no es ni mucho menos la excepción.

7.1 Elección de tecnología

El prototipo se desarrolló como una aplicación web simple, con componentes HTML estándar, gráficos predefinidos de dc.js, gestión de eventos directamente con la API del DOM y todo el código de la aplicación cabía en un único archivo JavaScript de 530 líneas.

Para hacer frente a la nueva demanda de flexibilidad de la aplicación, era predecible que su complejidad crecería de forma muy significativa, por lo que desde el principio de la migración a la nueva aplicación se buscó utilizar un framework que simplificara el manejo de eventos y la sincronización de la interfaz con el modelo interno del código.

La primera opción barajada fue Angular[18], pero después de varias pruebas se descartó por la dificultad de cargar componentes como datos, una funcionalidad crítica para ofrecer una interfaz de filtros compuestos.

Finalmente se escogió Knockout[19], por soportar con gran facilidad la funcionalidad mencionada anteriormente y por familiaridad debida a proyectos previos.

[18] https://angularjs.org/
[19] http://knockoutjs.com/

También se optó por migrar el código de JavaScript a TypeScript[20].

El capítulo Herramientas y técnicas de programación empleadas explica con mayor detalle cómo estas y otras herramientas contribuyeron a hacer el desarrollo de HEPData Explore mucho más manejable.

7.2 La interfaz de filtros

En el prototipo inicial, la interfaz de filtrado no era un problema muy complicado. Los filtros soportados eran un conjunto cerrado y cada uno de ellos tenía un espacio reservado en la interfaz.

En la interfaz del prototipo inicial solo era necesario permitir al usuario seleccionar una variable independiente y ver la lista completa de variables.

En la nueva versión de la aplicación, no obstante, se busca la máxima flexibilidad. El usuario puede o no querer un determinado filtro, por lo que debe ser capaz de agregarlos y quitarlos a voluntad.

Para permitir aún mayor flexibilidad, los filtros deben poder ser combinados para formar operaciones lógicas AND y OR. Los usuarios pueden querer además reordenar estas expresiones: por ejemplo, un usuario puede tener en un momento dado un filtro A. Más tarde el usuario quiere hacer OR con un nuevo filtro. La interfaz sería poco cómoda si requiriera al usuario borrar el filtro A, crear el filtro OR y después volver a crear el filtro A dentro del OR. Sería mucho mejor, por ejemplo, si el usuario de alguna manera pudiera mover el filtro A dentro del OR en vez de destruirlo y crearlo de nuevo.

Por último, la interfaz debe ser compacta para que los filtros puedan manipularse cómodamente en un ordenador con una pantalla normal.

7.2.1 El diseño de interfaz (a alto nivel)

Para cumplir todos los requisitos establecidos en el punto anterior se buscó que la interfaz fuera componible: Es decir, que unos elementos puedan vivir dentro de otros.

Así, el diseño debe partir de los filtros compuestos. Unos filtros compuestos pueden contener otros filtros, simples o compuestos. Los filtros compuestos son recursivos por definición.

[20] https://www.typescriptlang.org/

Diseñando un filtro compuesto componible habremos diseñado la mayor parte de la interfaz de filtros.

Otro aspecto a tener en cuenta para crear el diseño de los componentes de filtros es su ubicación, puesto que ésta marcará el espacio que pueden ocupar.

Se decidió asignar un panel lateral para colocar los filtros, situado en la parte izquierda de la pantalla. Esta decisión no se tomó por continuidad con el prototipo inicial, sino porque en la inmensa mayoría de pantallas de PC el espacio horizontal es abundante mientras que el espacio vertical es muy escaso. Este es especialmente el caso de la relación de aspecto más popular hoy en día, 16:9. Utilizar un panel lateral vertical permite reservar un área relativamente espaciosa de la interfaz para los filtros sin obstruir la vista principal aunque la pantalla sea pequeña y tenga pocas líneas horizontales.

Las figuras 7.1 y 7.2 muestran el diseño al que se llegó.

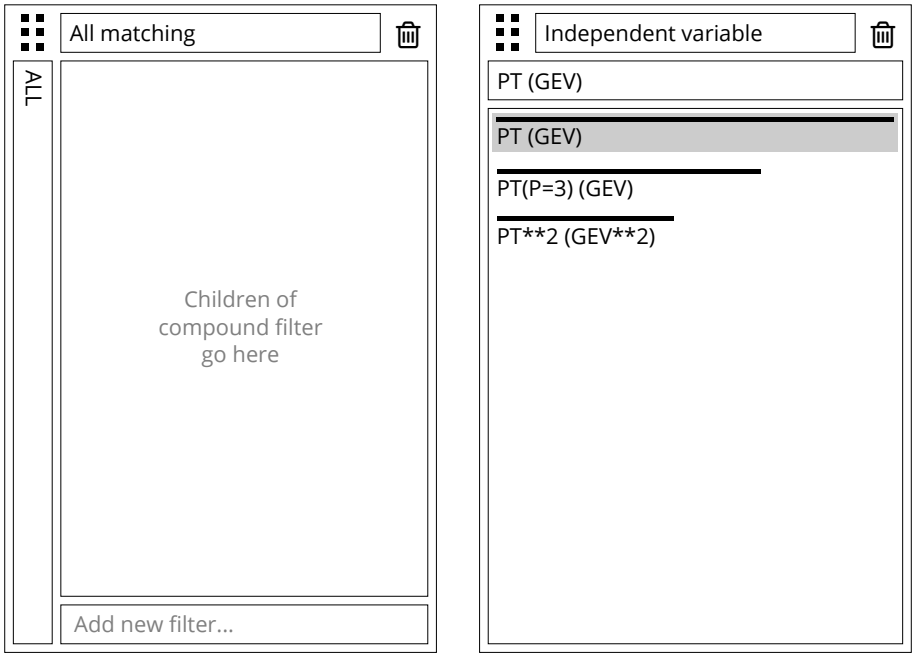

Figura 7.1: Wireframes mostrando el diseño de la interfaz de filtros. De izquierda a derecha: (a) Filtro compuesto *All* (AND) vacío, (b) filtro de variable independiente.

En la parte superior aparecen tres elementos. A la izquierda, el icono de seis puntitos es un ancla de arrastre. Haciendo clic y arrastrando sobre él un usuario puede mover el filtro a otro contenedor. A su derecha aparecen el nombre del filtro y un botón para eliminarlo del contenedor.

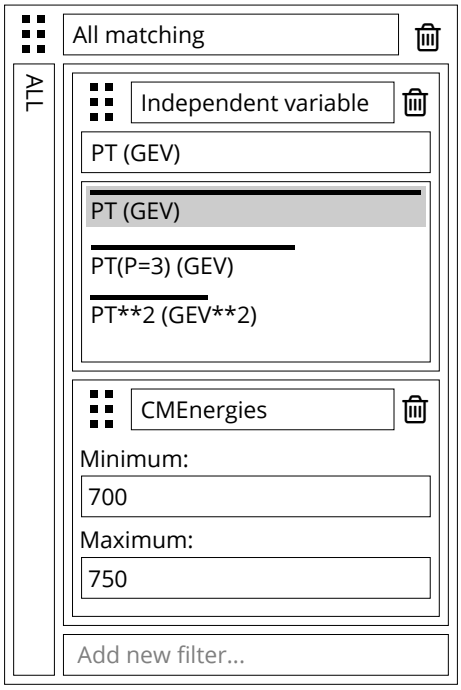

Figura 7.2: Wireframe mostrando un filtro *All* con otros dos filtros anidados.

El filtro compuesto muestra a la izquierda una bandera donde muestra *ALL* o *SOME* dependiendo de la operación lógica que aplique. El propósito de esta bandera es añadir un nivel de sangrado en cada filtro compuesto anidado, de manera que sea mucho más fácil distinguir qué filtros están dentro de qué otros.

La mayor parte de la interfaz del filtro compuesto la conforma el espacio reservado para los filtros hijos. Este área tiene un tamaño mínimo para que sea fácil arrastrar elementos dentro de un filtro compuesto recién creado. Si este área no tuviera tamaño mínimo, en un filtro nuevo sería invisible y no se podrían arrastrar filtros a ella.

Por último, en la parte inferior del filtro compuesto hay un campo para crear un nuevo filtro y añadirlo a su descendencia. El usuario haría clic en ese campo y escribiría el nombre del filtro que quisiera, recibiendo de manera automático sugerencias de autocompletado y mensajes de ayuda.

7.2.2 Diseño de elementos

A partir del diseño visual se dio un paso más hacia la implementación y se pasó al diseño de elementos HTML. Es decir, colocar los diferentes elementos visuales del diseño en grupos (`div`) de manera que formen una jerarquía y puedan ser manipulados fácilmente desde

código. Algunos de estos grupos marcan también la separación de responsabilidad entre varios componentes. Por ejemplo, dentro del filtro compuesto *All* habrá un `div` con los elementos hijos. Sin embargo, para el filtro compuesto el contenido de cada elemento hijo le es indiferente: su responsabilidad termina en permitir añadir el elemento hijo, pero no gestionar su contenido ni funcionalidad.

La figura 7.3 muestra el diseño de elementos poniendo el filtro compuesto *All* como ejemplo.

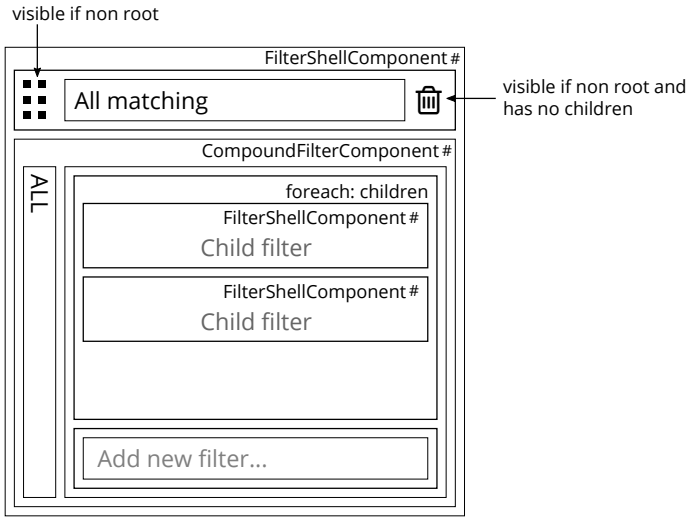

Figura 7.3: Diseño de elementos HTML. Al wireframe anterior se le han añadido cajas adicionales para mostrar los límites de los elementos HTML con los que se implementarían los elementos visuales originales.

En la misma figura también se han indicado qué elementos HTML son *componentes* (marcados con un signo #). Los componentes son clases que tienen asociada una plantilla HTML. La plantilla puede incluir datos de la clase e invocar sus métodos en respuesta a eventos. La plantilla de un componente también puede tener puntos de extensión para integrar otros componentes, como se da en este caso con los hijos del filtro compuesto.

Los componentes están diseñados para usar composición en vez de herencia, de manera que unos contienen a otros en puntos de extensión definidos en vez de reemplazar sus partes de forma arbitraria.

En el primer nivel está `FilterShellComponent`. Contiene dos elementos. En la parte superior está la cabecera, con el ancla de arrastre, el nombre del componente y el botón de borrado. Para que el botón de borrado funcione `FilterShellComponent` necesita tener una referencia al filtro padre, si existe. En otro caso no se muestra.

El filtro raíz no tiene padre, por lo que no puede ser eliminado. Tampoco puede ser movido,

puesto que no hay otro contenedor en el que pudiera introducirse sin provocar ciclos.

Nota: Se provocaría un ciclo, por ejemplo, si un usuario arrastrara un filtro compuesto al interior de sí mismo.

La implementación debe impedir que se puedan producir estos ciclos.

Debajo de la cabecera, `FilterShellComponent` contiene un punto de extensión para el cuerpo del componente. Este cuerpo es un componente específico para cada filtro. Algunos filtros muy similares comparten un mismo componente. Por ejemplo, los filtros *All* y *Some* utilizan ambos *CompoundFilterBody*. *CompoundFilterBody* accede a las propiedades del filtro para determinar qué debe pintar en la bandera, pero por lo demás la funcionalidad es exactamente la misma.

De forma similar, muchos filtros tienen la misma forma de interacción: el usuario ve una lista de posibles valores de un campo y teclea uno, recibiendo autocompletado. Es el caso de los filtros de variable independiente, variable dependiente, reacción, observable y frase. Todos estos filtros usan el componente `ChoiceFilterComponent`.

En el ejemplo de la figura 7.3, el componente utilizado para el cuerpo del filtro es `CompoundFilterBody`. En su interior se puede ver a la izquierda la bandera y a la derecha la lista de nodos hijo y el cuadro para añadir un nuevo filtro.

Cada filtro anidado se inserta de nuevo como un `FilterShellComponent`, de manera que el componente es recursivo. El usuario puede utilizar el ancla de arrastre y el botón de borrar mencionados al principio para mover o borrar el filtro hijo respectivamente. Los filtros compuestos también se pueden utilizar como filtros hijos.

Nota: Durante el diseño de elementos HTML se tuvo en cuenta la consistencia y ortogonalidad de la interfaz. En la medida de lo posible sin sacrificar la usabilidad, elementos con la misma funcionalidad deberían tener la misma jerarquía para poderse implementar con el mismo código.

Así, en el primer diseño, solo los filtros simples tenían la barra superior. Los filtros compuestos en cambio tenían solo la barra vertical a la izquierda y el ancla de arrastre y el botón de borrar estaban situados en ella.

Añadiendo también una barra superior a los filtros compuestos y moviendo la barra vertical al contenido – dejándola solo como pista visual de anidamiento, se logró no solo hacer el

diseño de elementos más simple y fácil de implementar, sino también se logró una interfaz más consistente, pues ahora todos los filtros tienen la misma estructura visual.

7.2.3 Diseño de objetos

Una vez se encontró la manera de agrupar los elementos de interfaz en elementos HTML, y definidos los límites componentes, el siguiente paso fue desarrollar una jerarquía de clases capaz de soportar todas las interacciones del sistema.

En este diseño se hizo hincapié en los siguientes principios:

- **Objetos valor vs objetos de funcionalidad**: Los filtros son datos. Modelarlos como clases puras independientes de la interfaz de usuario ofrece ventajas a la hora de implementar varias funcionalidades de la aplicación como son la serialización y persistencia de estados, el cálculo de filtros complementarios, la construcción de objetos de búsqueda de ElasticSearch y la aplicación de los aplicación de filtros en el lado del cliente.

  Tratar a los filtros como objetos de datos permite implementar todas las funcionalidades anteriores sin acoplar a ellas entidades de la interfaz de usuario. La creación, modificación, reordenación y destrucción de filtros es una operación ligera que por si sola no altera el estado de la interfaz ni requiere de ella.

- **Separación de responsabilidades**: Dado que el número de componentes es elevado, es importante que sus límites estén definidos para que la implementación sea limpia y no existan dependencias implícitas que causen, por ejemplo, que un filtro compuesto falle dependiendo del tipo de hijos que tenga.

7.2.4 Objetos filtro

En el diseño creado, existe por un lado una jerarquía de clases que parte de `Filter` con los datos del filtro. Existe una subclase para cada tipo de filtro. Algunos ejemplos: `DepVarFilter`, `IndepVarFilter`, `CMEnergiesFilter`, `AllFilter`, `SomeFilter`.

Dentro de la jerarquía existen clases intermedias para aquellos filtros que comparten similitudes. Así, `AllFilter` y `SomeFilter` heredan de la subclase `CompoundFilter`, lo que permite tratar con ambas clases de filtros con una misma interfaz capaz de enumerar y manipular sus hijos.

Otro ejemplo de reutilización es `ChoiceFilter`, una clase intermedia de la que heredan `DepVarFilter`, `IndepVarFilter`, `ObservableFilter`, `ReactionFilter` y `PhraseFilter`. Todos estos filtros tienen en común que filtran tablas de acuerdo al valor de un campo. La subclase define qué campo se filtrará durante su inicialización. Así, la implementación de la mayoría de estos filtros es tan simple como el código a continuación:

```
@registerFilterClass
export class PhraseFilter extends ChoiceFilter {
    constructor(value: string = '') {
        super('phrases', value);
    }

    static getLongName() {
        return 'Phrase'
    }
}
```

La cadena `'phrases'` en el código anterior especifica el campo de la tabla por el que se hará el filtrado.

La jerarquía de `Filter` tiene varias responsabilidades, que toda subclase hoja debe cumplir:

- **Almacenar los criterios de filtrado**. Para ello los filtros utilizan propiedades específicas según el tipo de filtro.

  Estas propiedades están marcadas como observables de manera que la interfaz de edición de filtros siempre está sincronizada con los objetos de filtro. Otras suscripciones a estos observables también son utilizadas para detectar cambios en los criterios de filtrado establecidos por el usuario y ejecutar una búsqueda actualizada.

  Las suscripciones a los observables son completamente independientes de la instanciación de los filtros, de manera que esta sincronización no introduce acoplamiento entre la clase `Filter` y ninguna de las otras funcionalidades.

- **Generar objetos de consulta**. Para ello los filtros deben implementar el método `toElasticQuery()`, que devuelve un objeto filtro de ElasticSearch.

  Las implementaciones de este método de `AllFilter` y `SomeFilter` llaman recursivamente a los métodos `toElasticQuery()` de sus hijos para traducir los filtros compuestos al lenguaje de ElasticSearch.

- **Serialización y deserialización**. Los filtros implementan un esquema de serialización simple y seguro basado en POJOs (Plain Old Javascript Objects).

Cada filtro se serializa como un objeto con dos propiedades, `type`, que contiene el nombre de la clase filtro y `params`, que contiene un objeto con un par clave-valor para cada propiedad del filtro que almacena criterios de filtrado.

Toda la lógica de la serialización y deserialización está implementada en la clase base `Filter`, la cual es expuesta públicamente a través de los métodos `load()` y `dump()`.

Para hacer funcionar la serialización las clases derivadas solo necesitan llamar al método protegido `registerSerializableFields()` pasando como parámetro una lista con los nombres de las propiedades que contengan criterios de filtrado. Este método puede ser llamado varias veces, en el caso que clases intermedias en la jerarquía declaren propiedades nuevas.

Para que una propiedad pueda ser registrada como serializable debe tener un tipo admitido. Son admisibles los valores serializables a JSON (nulos, números, cadenas, arrays, POJOs). Adicionalmente, también se admiten propiedades cuyo tipo sea otro filtro o un array de filtros. En este caso, los filtros anidados son serializados de forma recursiva llamando a sus métodos `dump()`.

Para que la deserialización funcione es necesario que todas las clases hoja se registren durante la inicialización del programa. Este registro es mantenido por el objeto `filterRegistry`, que mantiene una asociación entre los nombres de las clases de filtros y su constructor, de manera que sea posible saber qué clase se debe instanciar durante la deserialización.

La forma más simple de hacer este registro es añadir el decorador `@registerFilterClass` a la declaración de la clase. Este método es utilizado por todas las clases hoja.

7.2.5 Componentes de filtro

Los componentes de filtro son el puente entre las clases filtro y la interfaz de usuario.

Cada componente de filtro contiene, además de una asociación con la clase filtro, propiedades adicionales para almacenar el estado de la interfaz y métodos para manejar eventos (por ejemplo, *tecla Enter presionada en el campo de texto*).

Algunos ejemplos de datos que forman parte del estado de la interfaz y que son guardados por el componente de filtro son:

- **Contenido de los campos de texto de la interfaz de filtro.** Los objetos filtro solo son

editados con valores consistentes. Todos los valores temporales usados como parte de la interacción con el usuario se almacenan en el componente.

De esta manera, `ChoiceFilterComponent` tiene una propiedad `valueTyped` que se inicializa con el valor de la propiedad `value` de `ChoiceFilter`, pero que después tiene una vida independiente, volviendo a sincronizarse con `ChoiceFilter` solo cuando el usuario confirma el valor escrito.

- **Sugerencias de autocompletado:** Algunos componentes ofrecen autocompletado de los posibles valores de sus filtros. Las sugerencias extraídas de base de datos se almacenan en el componente.

- **Estado volátil de la interfaz:** Cualquier cantidad de estado de la interfaz de un componente de filtro que sea necesario tener en cuenta para ciertas funcionalidades del sistema pero no sea necesario persistir ni reflejar en el objeto filtro; por ejemplo, un booleano indicando si un mensaje de ayuda está activado en un determinado momento.

Cada clase filtro tiene asociada una clase componente, pero varias clases filtro pueden compartir la misma clase componente. Así, por ejemplo, `CompoundFilterComponent` es utilizado por `AllFilter` y `SomeFilter`. Al componente le es indiferente que el filtro sea uno u otro, lo que le importa es la interacción del usuario con el mismo, que en este caso es la misma: añadir y quitar filtros hijos.

Lo mismo ocurre con `ChoiceFilterComponent`, que es utilizado para todas las subclases de `ChoiceFilter`.

En la figura 7.4 se puede ver como queda la jerarquía de clases de filtro y como se relaciona con sus componentes asociados.

7.3 Autocompletado

Muchos componentes de HEPData Explore dependen de una función de autocompletado para ayudar al usuario a introducir valores sin errores, descubrir las funcionalidades de la interfaz y descubrir los conjuntos de datos que puede explorar. Concretamente el autocompletado se usa en varias áreas:

- En todos los filtros de elección se utiliza el autocompletado no solo para ayudar a introducir las variables o valores, sino también para ver las opciones que tienen mayor número de registros.

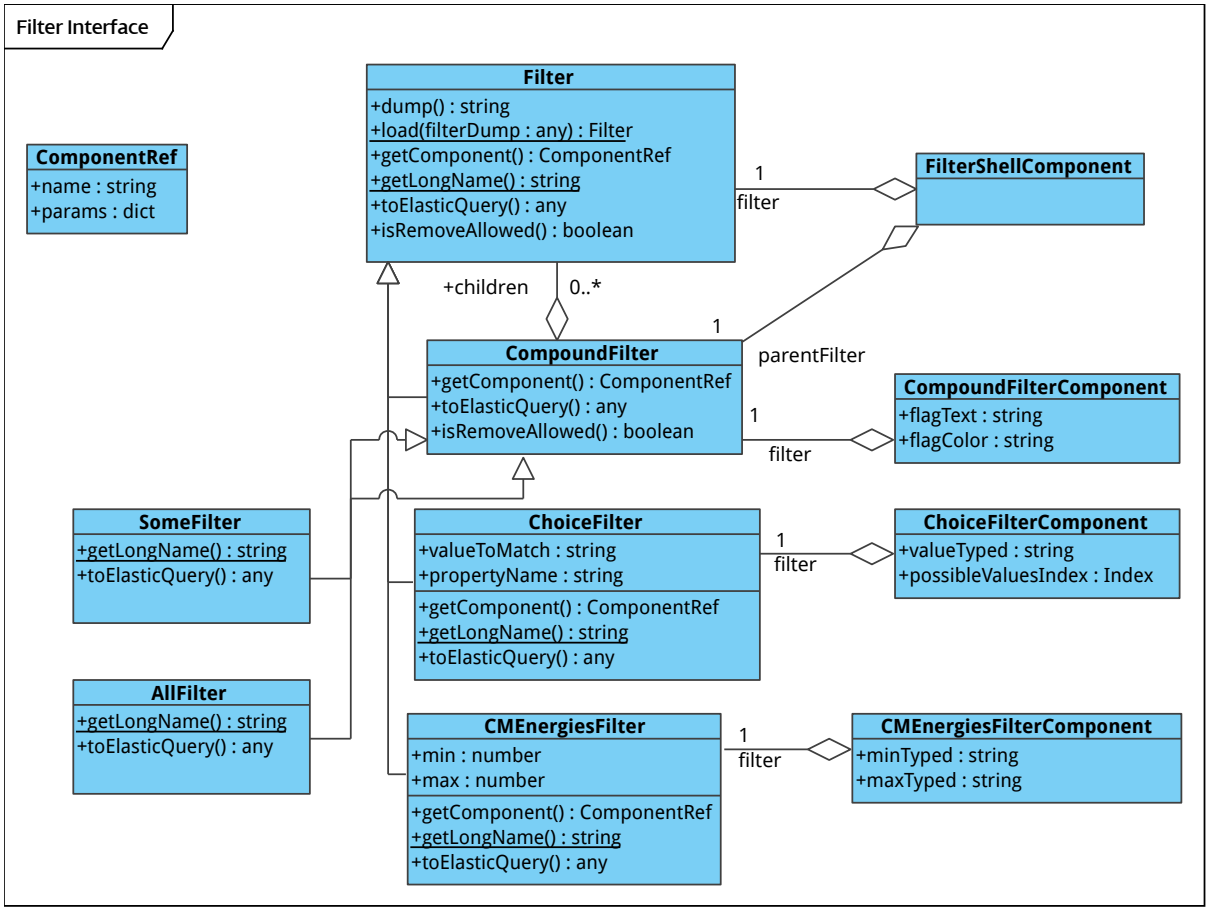

Figura 7.4: Diagrama de clases simplificado de la interfaz de filtros.

- La creación de un nuevo filtro se hace con un campo de texto con autocompletado.
- En la vista de edición de gráficos también se hace uso de autocompletado para listar y permitir escoger las posibles variables a visualizar.

7.3.1 El servicio de autocompletado

Dado el potencial de reutilización y la variedad de usos de caso, se creó una clase `AutocompleteService` con la funcionalidad común a todos los casos de autocompletado. Cada instancia de `AutocompleteService` se configura con los siguientes parámetros:

- `koQuery`: Observable que en todo momento contiene la cadena de texto introducida por el usuario. Cada vez que el usuario pulsa una tecla, esta cadena se modifica y `AutocompleteService` recibe una notificación.
- `searchFn`: Función de búsqueda. Recibe una cadena de búsqueda procedente de `koQuery` y debe devolver una lista de elementos *sugerencia* con formato libre. Cuando en las funciones siguientes se hable de sugerencias, se entiende que utilizan el formato utilizado en esta función.

  De esta manera, el `AutocompleteService` puede autocompletar datos de distinta naturaleza, y sus clases cliente pueden incorporar en estos objetos *sugerencia* cualquier información que consideren relevante para su caso de uso.
- `rankingFn`: Función de calificación. Permite ordenar los elementos devueltos por `searchFn` de acuerdo a alguna métrica de relevancia.

  La puntuación se devuelve como uno o más números. Las sugerencias con valores más bajos aparecen primero. Si la función devuelve varios valores para cada sugerencia, el primero tiene prioridad y el resto se usan para desempatar. Esto permite construir con precisión métricas basadas en múltiples criterios de forma muy fácil.

  A modo de ejemplo, el siguiente es un extracto del código de puntuación para el autocompletado de un campo de variable en la vista de edición de gráficos. La variable usada actualmente en el gráfico siempre aparece en primer lugar, después todas aquellas que, con la configuración actual, tienen datos para ser mostradas y después todas las demás. Dentro de los últimos dos grupos, se ordenan por longitud del nombre de la variable (por ejemplo, `PT` aparece antes que `PT(P=3)`):

```
    rankingFn: (s: VariableChoice) => [
        // The currently selected variable always appears first,
        // then all the rest
        s.name == this.cleanValue ? 0 : 1,
        // Cross matches appear first
        s.isCrossMatch ? 0 : 1,
        // Finally, the suggestions are sorted by string length
        // (simpler matches first)
        s.name.length,
    ],
```

- `keyFn`: Función de identificación. Recibe una sugerencia y devuelve un atributo único de la misma. Si dos objetos *sugerencia* devuelven el mismo resultado al introducirlos en esta función, se consideran la misma sugerencia aunque provengan de dos búsquedas distintas.

  Esta función se utiliza con fines de optimización. Después de cada búsqueda solo se crean elementos de interfaz para aquellas sugerencias que, de acuerdo a esta función, sean nuevas; el resto se reutilizan.

- `suggestionAcceptedFn`: Manejador de evento para la selección de una sugerencia. Cuando se hace clic en una sugerencia o se acepta con el teclado, esta función es invocada, recibiendo la sugerencia como parámetro.

- `maxSuggestions`: Máximo número de sugerencias que serán presentadas en la interfaz. La función de búsqueda puede devolver un número de sugerencias mayor, pero serán recortadas a esta cantidad después de ser ordenadas.

- `acceptWithTabKey`: Booleano indicando si se acepta la tecla Tabulador para aceptar una sugerencia. Si está a `false`, solo se aceptará la tecla Enter para aceptar una una sugerencia y Tab se dejará para mover el foco entre controles. Si está a `true`, adicionalmente se registrará la tecla Tab para aceptar la sugerencia antes de mover el foco.

`AutocompleteService` no define ninguna plantilla HTML para los elementos del menú de autocompletado, sino que los deja a la implementación y en lugar de eso ofrece manejadores para sincronizar los componentes de cualquier conjunto de elementos de autocompletado con el servicio.

En particular, `AutocompleteService` exporta los siguientes métodos y propiedades:

- `suggestions`: Lista de sugerencias a mostrar (inicialmente vacía).

- `selectedSuggestionIx`: Índice de la sugerencia seleccionada actualmente, relativo a `suggestions`.
- `acceptSelected()`: Acepta la sugerencia actualmente seleccionada con los cursores del teclado.
- `nextSuggestion()`: Selecciona la sugerencia siguiente a la actualmente seleccionada, o en su defecto la última.
- `prevSuggestion()`: Selecciona sugerencia anterior, o en su defecto la primera.
- `pageDown()`: mueve el cursor una página abajo.
- `pageUp()`: mueve el cursor una página arriba.
- `keyPressed`: Manejador de evento reutilizable que las plantillas conectan al evento de pulsación de teclado del cuadro de texto donde escribe el usuario. Maneja las flechas de cursor e incluye algoritmos de paginación para *AvPag* y *RePag*.
- `koMouseDownHandler`: Manejador de evento para clics sobre elementos de sugerencia. El prefijo `ko` indica que este es un manejador estilo knockout, por lo que debe recibir como primer parámetro el objeto sugerencia.
- `leakScrollPaneHandler()`: permite pasar el control HTML que contiene barras de desplazamiento, si hay alguno. Esta función es necesaria para hacer funcionar *AvPag* y *RePag* en vistas que tengan estas barras.

7.3.2 Los índices con lunr

Cuando se explicó el servicio de autocompletado en el punto anterior un aspecto que se obvió fue la búsqueda de coincidencias. Este punto está dedicado a estas búsquedas.

La búsqueda de sugerencias de autocompletado se hace en el lado del cliente con el fin de que la interacción sea lo más fluida posible. Para ello se hace uso de lunr.js[21], un pequeño motor de búsqueda diseñado para este tipo de aplicaciones.

El diseño de lunr imita aquel de los motores de búsqueda con capacidad de tratamiento de texto natural como Solr o ElasticSearch, pero muy simplificado para caber en una biblioteca de menos de 5 KiB.

[21] http://lunrjs.com/

Al igual que en ElasticSearch, en primer lugar se define un mapeo.

```
// Searches for text, ignoring a defined set of stopwords
this.textIndex = lunr(function() {
    this.field('name', {boost: 10});
    this.field('description');
    this.ref('id');
});
```

El campo `boost` indica que las coincidencias en `name` tendrán una mayor repercusión en los resultados que las de `description`.

Cada documento indexado en lunr debe tener un identificador (campo `id`). Este identificador es almacenado en el índice junto a los valores transformados de los demás campos.

Nota: A diferencia de como ocurre en ElasticSearch, en lunr no hay `_source`. Para acceder a las propiedades del objeto original indexado, éste se debe guardar en otro lugar y debe ser recuperable a partir del identificador especificado en `id`.

Una vez definido el mapeo, se usa el método `add` para añadir documentos.

```
// For each filter
this.database.push(filter);
this.indexTags.add({
    id: i++,
    name: this.database[i].name,
    description: this.database[i].description,
});
```

Finalmente, se usa el método `search()` para buscar en el índice, pasando como parámetro la cadena de búsqueda del usuario.

```
const rawResults = this.textIndex.search(query);
const results = resultsText.map((result) => {
    return {
        match: this.database[result.ref],
        score: result.score,
    };
})
```

Nótese que es necesario consultar la base de datos local para extraer los datos de los documentos encontrados en la búsqueda. Cada coincidencia devuelta por lunr solo contiene

`ref` (el identificador utilizado) y `score` (una métrica de relevancia).

En lunr, la búsqueda de una cadena vacía devuelve cero resultados, por lo que para ofrecer sugerencias de autocompletado antes de que el usuario escriba ninguna letra es necesario capturar ese caso y devolver la base de datos completa o un subconjunto relevante.

7.3.3 Los índices en la interfaz de nuevo filtro

El principal caso de uso con el que HEPData Explore amortiza más lunr es el autocompletado de nuevo filtro. En la figura 7.5 se puede ver esta interfaz.

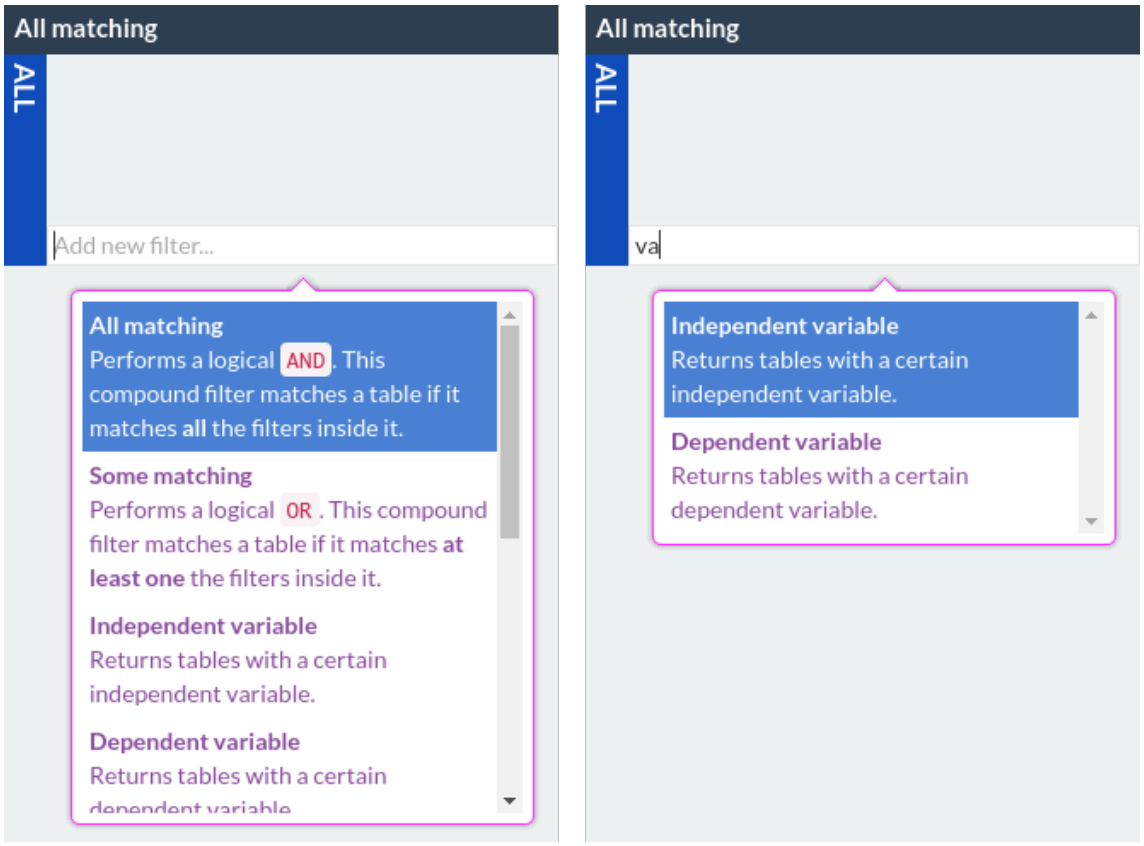

Figura 7.5: El usuario hace clic sobre el campo de texto *Add new filter* y automáticamente aparece un menú de burbuja con todos los filtros existentes y una breve explicación de cada uno. El usuario puede navegar por la lista utilizando el ratón o el teclado, o puede teclear parte del nombre, descripción o palabras relacionadas para encontrar el filtro deseado muy rápidamente.

Este índice se puede consultar a través del objeto singleton `FilterIndex`.

La principal diferencia entre el código de ejemplo del punto anterior y el código de `FilterIndex` es que se han añadido *etiquetas* para poder encontrar filtros con palabras que de otro modo serían filtradas como stopwords. Este problema es particularmente notable en el caso de los filtros compuestos, pues todas estas son stopwords: *all*, *some*, *or*, *and*.

lunr permite modificar el cauce de procesamiento de texto de un índice, lo que hace posible desactivar el filtro de stopwords:

```
myIndex.pipeline.remove(lunr.stopWordFilter);
```

Desafortunadamente, el cambio afecta a todo el índice y no es posible aplicarlo solo a un campo específico.

Para poder localizar estos filtros utilizando estas palabras comunes sin eliminar el filtrado de stopwords en general se creó un índice adicional, `tagsIndex`, solo con un campo `tags` cuyo valor es una lista de etiquetas asociadas a cada filtro.

A la hora de hacer una búsqueda, se buscan los términos del usuario en los dos filtros y se hace la combinación de ambos.

Se aprovechó este diseño para añadir etiquetas adicionales para sinónimos y palabras clave que es probable que un usuario escriba para encontrar un filtro pero que no necesariamente están en la descripción. En la figura 7.6 se puede ver un ejemplo de esta asociación.

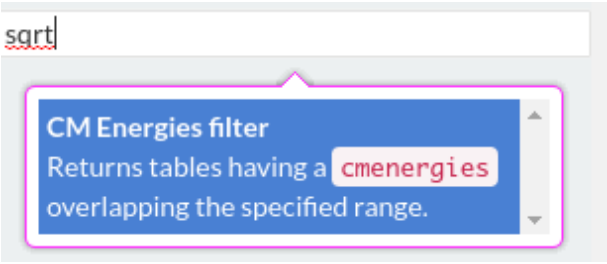

Figura 7.6: Aunque el texto *sqrt* no aparece en la descripción del filtro, el filtro aparece como resultado de la búsqueda por palabras clave ya que `SQRT(S)` es un nombre alternativo de `cmenergies`.

A continuación se muestra un extracto de la base de datos de filtros:

```
filterIndex.populate([
    {
        filterClass: AllFilter,
        description: `Performs a logical <code>AND</code>.
        This compound filter matches a table if it matches <b>all</b>
```

```
        the filters inside it.`,
        tags: ['all', 'and', 'compound'],
    },
    {
        filterClass: SomeFilter,
        description: `Performs a logical <code>OR</code>. This
        compound filter matches a table if it matches <b>at least one
        </b> the filters inside it.`,
        tags: ['some', 'or', 'any', 'compound'],
    },
    {
        filterClass: IndepVarFilter,
        description: `Returns tables with a certain independent
        variable.`,
        tags: ['x', 'indep', 'independent', 'variable'],
    },
    {
        filterClass: CMEnergiesFilter,
        description: `Returns tables having a <code>cmenergies</code>
        overlapping the specified range.`,
        tags: ['sqrt', 'sqrts', 'cmenergies'],
    },
]);
```

7.4 Los gráficos

Una vez descargados los datos filtrados desde ElasticSearch el siguiente paso es mostrarlos al usuario para que los visualice. La forma en la que esto se hace en HEPData Explore es mediante gráficos de dispersión.

Los gráficos se alimentan de las tablas filtradas. Son capaces de representar pares de variables x e y, donde todos los pares de un mismo gráfico tienen la misma x pero distinta y. Se admiten de forma indistinta variables dependientes e independientes tanto para x como para y.

Además de representar el valor de cada fila de datos, los puntos de los gráficos de dispersión también muestran la dimensión del error, tanto para la variable x como para la variable y.

En caso de que una variable tenga varios errores, el error a representar se calcula como la

magnitud del vector donde cada error es una dimensión:

$$e' = \sqrt{\sum_{i=1}^{n} e_i^2}$$

El error puede ser asimétrico, en cuyo caso la magnitud del error se calcula de forma separada para el lado positivo y para el lado negativo.

7.4.1 Implementación de los gráficos

Durante el desarrollo de HEPData Explore se ha cambiado en varias ocasiones la implementación del componente de gráficos:

- Inicialmente se usó dc.js[22] (basado en d3.js[23]), pero su rendimiento no era adecuado para la cantidad de datos que se manejaban.
- Para afrontar el problema de rendimiento del componente anterior, se reescribieron los gráficos en WebGL. El rendimiento mejoró de forma significativa y abrió puertas a nuevos tipos de visualizaciones, pero también introdujo mayor complejidad.
- En un momento del desarrollo se introdujo un componente alternativo de gráficos basado en canvas 2D[24] para dar funcionalidad básica a aquellos equipos sin un buen soporte de WebGL. El rendimiento resultó ser bastante bueno y la implementación mucho más sencilla, y puesto que hasta no se había usado hasta entonces ninguna funcionalidad que requiriera WebGL, este componente acabó sustituyendo al componente WebGL para todos los casos.

La implementación actual está basada a su vez en componentes:

- `Plot` es la clase principal. Contiene la lista de capas y referencias a otros componentes. También es la clase que contiene métodos para el pintado y la actualización de los datos.
- `PlotConfig` almacena aquellas propiedades que distinguen a un gráfico de otro, en particular:
  - Su variable x.
  - Sus variables y.

[22] https://dc-js.github.io/dc.js/
[23] https://d3js.org/
[24] https://developer.mozilla.org/en/docs/Web/API/CanvasRenderingContext2D

  - Si ese gráfico está anclado (se explicará más adelante).
  - Su política de color: Se puede elegir entre que cada color represente una variable o una tabla.

- `AxesLayer` se encarga de pintar los ejes.
- `ScatterLayer` se encarga de pintar el gráfico de dispersión sobre los ejes.
- `PlotComponent` une el gráfico con el resto de la aplicación. Es el responsable de asignarle un espacio en la página para que se aloje y de hacer funcionar los botones de gestión para para modificar, eliminar o mover el gráfico.

Este diseño por capas hace más fácil incorporar y mezclar otros tipos de gráficos en el futuro.

7.4.2 Gráficos automáticos

Una funcionalidad importante de HEPData Explore es que es capaz de sugerir gráficos al usuario a partir del conjunto de datos.

Un gráfico en HEPData Explore puede estar en dos estados: anclado o automático. Los gráficos automáticos son modificados, eliminados y creados por la aplicación cuando se actualizan los filtros, mientras que los gráficos anclados solo son alterados como resultado de una interacción explícita del usuario con los mismos.

El algoritmo de gráficos automáticos se ejecuta tras cada modificación de los filtros hecha por el usuario. El procedimiento en detalle se puede ver en el fichero `frontend/app/services/autoPlots.ts`. A continuación se describe de forma simplificada:

1. Una vez recibidos los datos filtrados, para cada gráfico existente:

   a) Se actualizan los datos del gráfico: Se escogen aquellas tablas de los datos filtrados que tengan pares de variables que aparezcan en la configuración del gráfico.

   b) Si el gráfico no está anclado, se eliminan aquellas variables y para las que no se hayan encontrado datos en las tablas.

   c) Si el gráfico no está anclado y se queda sin variables y, se elimina.

   d) Se lleva un registro de aquellos pares de variables que al final de este paso están representadas en algún gráfico, con independencia de que éste sea anclado o

automático.

2. Se determina el número de gráficos libres hasta llegar al límite de gráficos. El algoritmo puede crear gráficos hasta alcanzar un máximo (contando tanto los anclados como los automáticos).

   El usuario puede superar este límite creando gráficos manualmente.

3. Se buscan todos los pares de variable independiente con variable dependiente dentro de los datos filtrados y se registra cuántas filas existen de cada par entre todas las tablas.

   De esta lista se filtran aquellos pares de variables que ya existan en un gráfico (utilizando el registro del punto 1.d).

   Finalmente se ordenan los resultados de mayor a menor número de filas.

4. Se crea una lista de *plantillas* de gráficos (`PlotBlueprint`) a partir de los gráficos automáticos existentes hasta este momento. Cada uno de estos elementos contiene:

   - `xVar`: Define la variable x del gráfico.
   - `yVars`: Define las variables y del gráfico.
   - `existingPlot`: Define el objeto gráfico real (`Plot`) a partir del cual se creó esta plantilla.

   El objetivo de estas plantillas es tener unas estructuras ligeras a las que poder añadir y quitar variables de forma eficiente durante la ejecución del algoritmo sin riesgo de causar efectos secundarios. Al final del algoritmo se actualizarán los gráficos reales a partir de estas plantillas.

5. Para cada par de variables obtenido en el paso 3, empezando por aquellos con mayor número de filas:

   a) Se comprueba si están en algún gráfico con el registro del paso 1.d. En caso afirmativo, se salta al siguiente par de variables.

   b) Si el par no está representado en ningún gráfico, se busca una plantilla de gráfico que tenga la variable independiente del par como variable x y tenga espacio para una nueva variable y.

   Cada gráfico automático puede tener hasta cinco variables y. Dos gráficos pueden

compartir la misma variable x con diferentes variables y.

c) Si no se pudo encontrar ningún gráfico con espacio en el paso anterior pero aun no se ha superado el límite de gráficos, se crea una nueva plantilla de gráfico con `xVar` establecido a la variable independiente del par y `existingPlot` establecido a nulo.

d) Si en cualquiera de los pasos anteriores se encontró un gráfico con espacio, se añade la variable dependiente del par como una variable y del gráfico y se registra el nuevo par de variables en el registro del punto 1.d.

6. Para cada plantilla de gráfico:

a) Si `existingPlot` no es nulo, se copian los datos de la plantilla en el gráfico real.

b) Si `existingPlot` es nulo, se crea un nuevo gráfico inicializado con los datos de la plantilla y se añade a la interfaz.

c) Si el gráfico contiene más de una variable y, se establece una política de color por variable. De lo contrario se establece una política de color por tabla.

7.4.3 Gráficos manuales

HEPData Explore permite a los usuarios editar los gráficos generados de acorde a sus necesidades, así como crear sus propios gráficos con las variables que deseen. Esto puede hacerse haciendo clic en el botón *Edit plot* de uno de los gráficos, o bien utilizando el botón *Custom plot*, en cuyo caso se parte de un gráfico en blanco.

La figura 7.7 muestra la interfaz de edición de gráficos.

El usuario puede modificar las variables de los ejes x e y mediante los campos de texto de la interfaz. Al final de la lista de variables y siempre aparece un campo de texto en blanco que permite añadir una nueva variable. Las variables existentes pueden ser borradas con el botón del cubo de basura o borrando su contenido y cambiando de campo (ej. con la tecla Tab).

Los campos de variable tienen autocompletado contextual. En la figura 7.8 se puede ver un ejemplo de esta interfaz en funcionamiento.

Si estamos editando una variable y, en morado aparecen aquellas sugerencias que con la variable x actualmente seleccionada tendrían datos representables. En blanco aparecen aquellas que tendrían datos si se escogiera otra variable x.

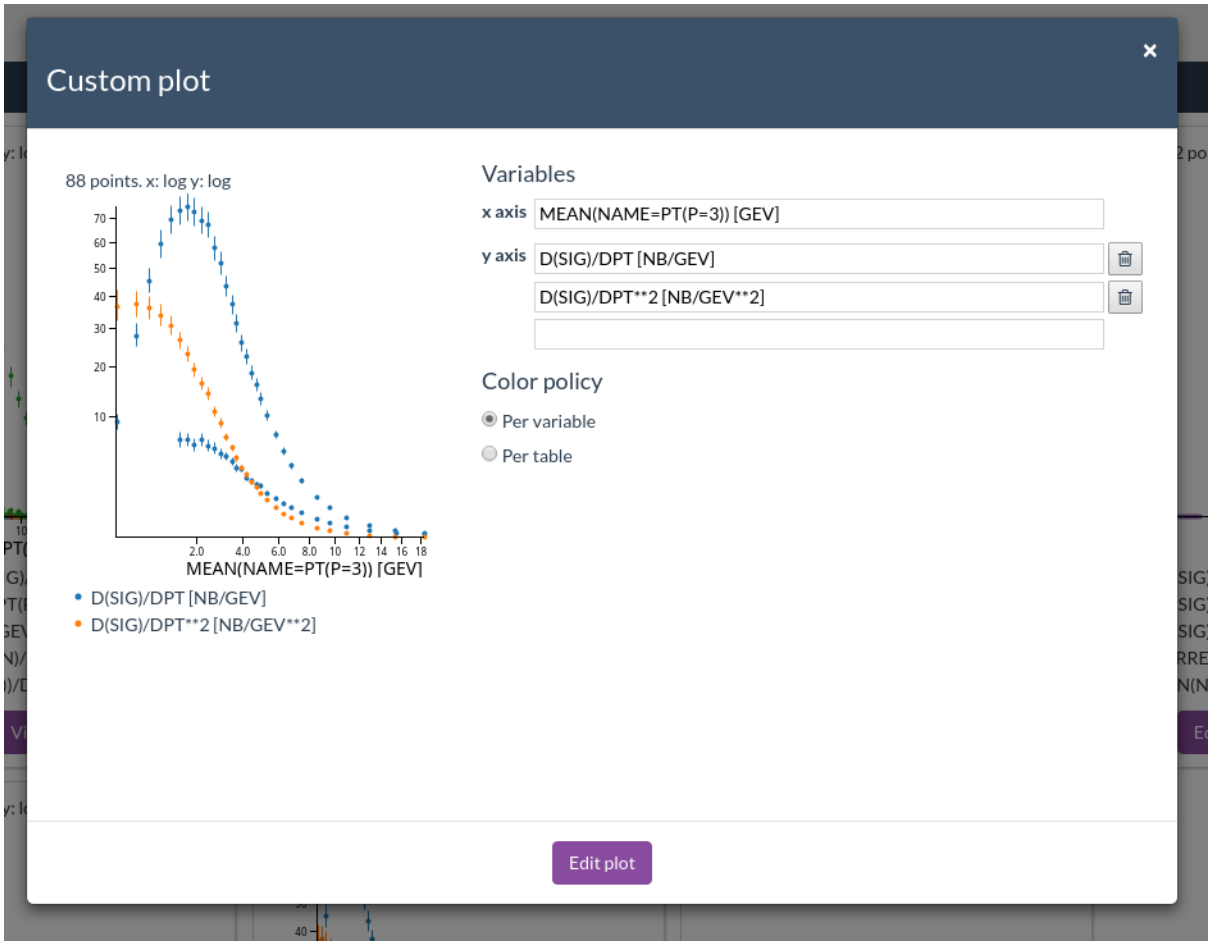

Figura 7.7: Interfaz de edición de gráficos.

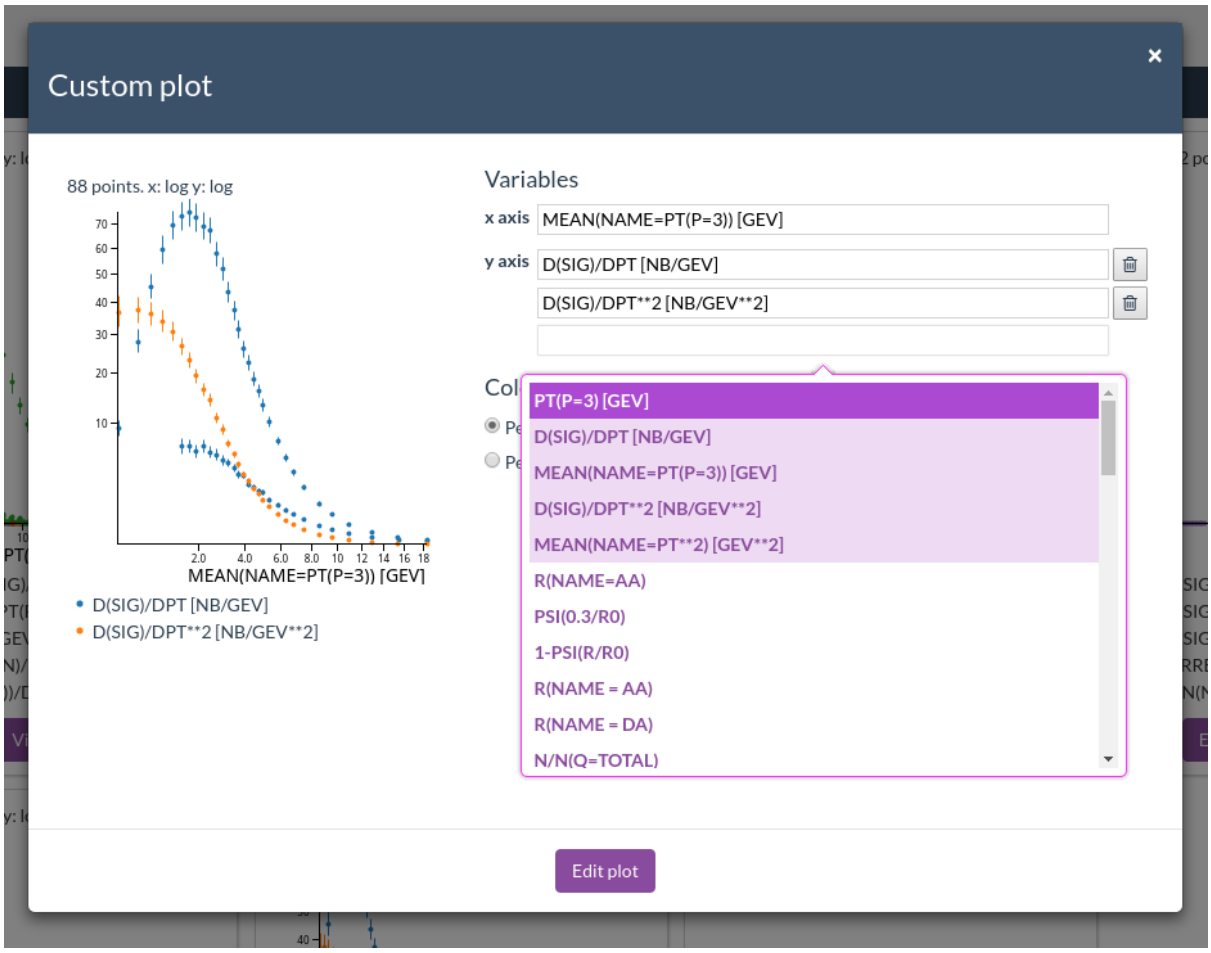

Figura 7.8: Autocompletado contextual de variables.

La misma funcionalidad existe en el campo de la variable x. El autocompletado de la variable x muestra en morado aquellas variables para las cuales al menos una de las variables y escogidas tendría datos.

El botón *Edit plot* aplica los cambios al gráfico y cierra el cuadro de edición. De forma similar, el botón *Add plot*, en el caso de un gráfico nuevo, añade un gráfico nuevo a la visualización con los parámetros elegidos.

En cualquiera de los dos casos el gráfico se marca como anclado (representado con el botón de la chincheta en amarillo y con un pequeño borde) para evitar que se pierda al cambiar los filtros. En la figura 7.9 se puede ver un gráfico anclado y uno en estado automático.

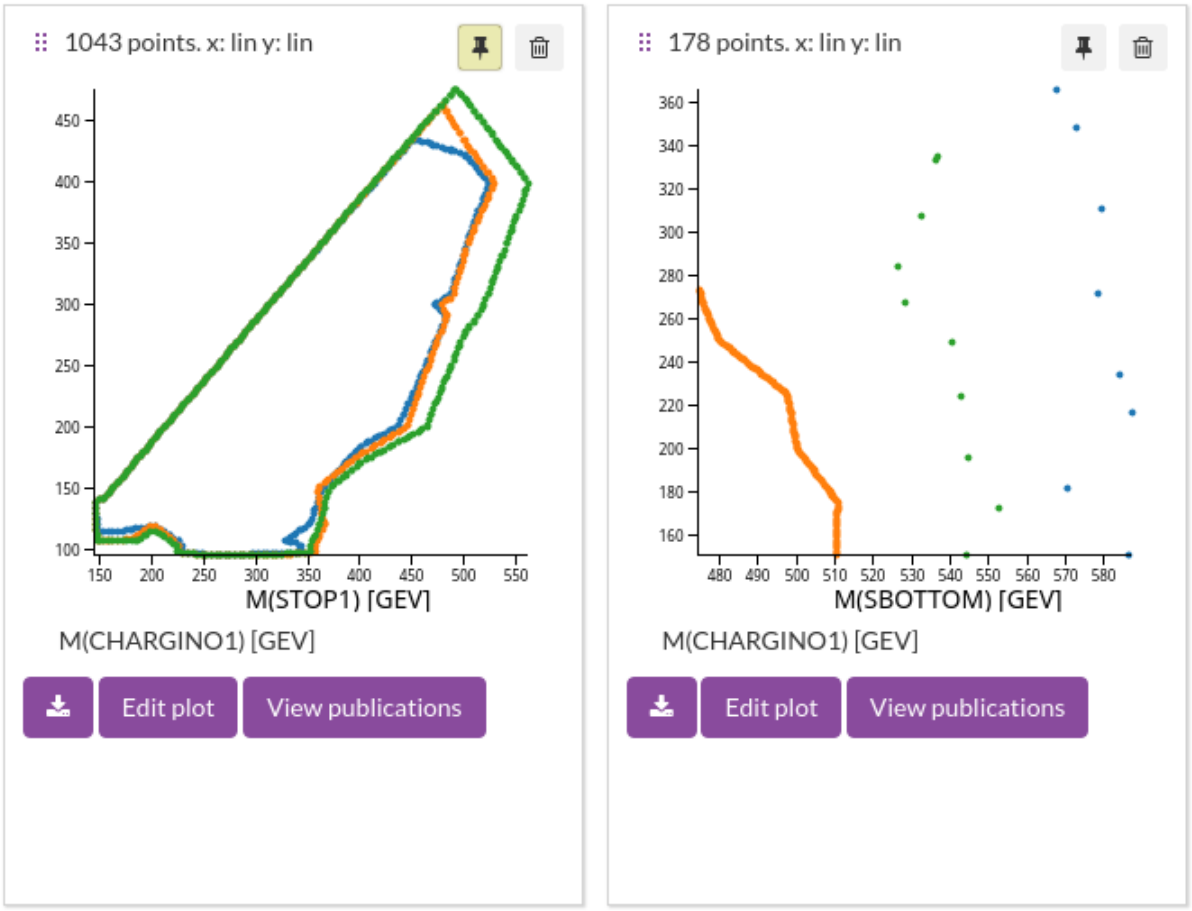

Figura 7.9: El gráfico de la izquierda está anclado. El gráfico de la derecha no lo está, por lo que desaparecerá si el usuario busca datos que no incluyan el par de variables `M(CHARGINO1)` y `M(SBOTTOM)`.

7.5 Lista de publicaciones

Los gráficos disponen de un botón *View publications* que permite ver la lista de artículos y tablas de las que salen los datos de ese gráfico.

En la figura 7.10 se puede ver un ejemplo de la vista de publicaciones.

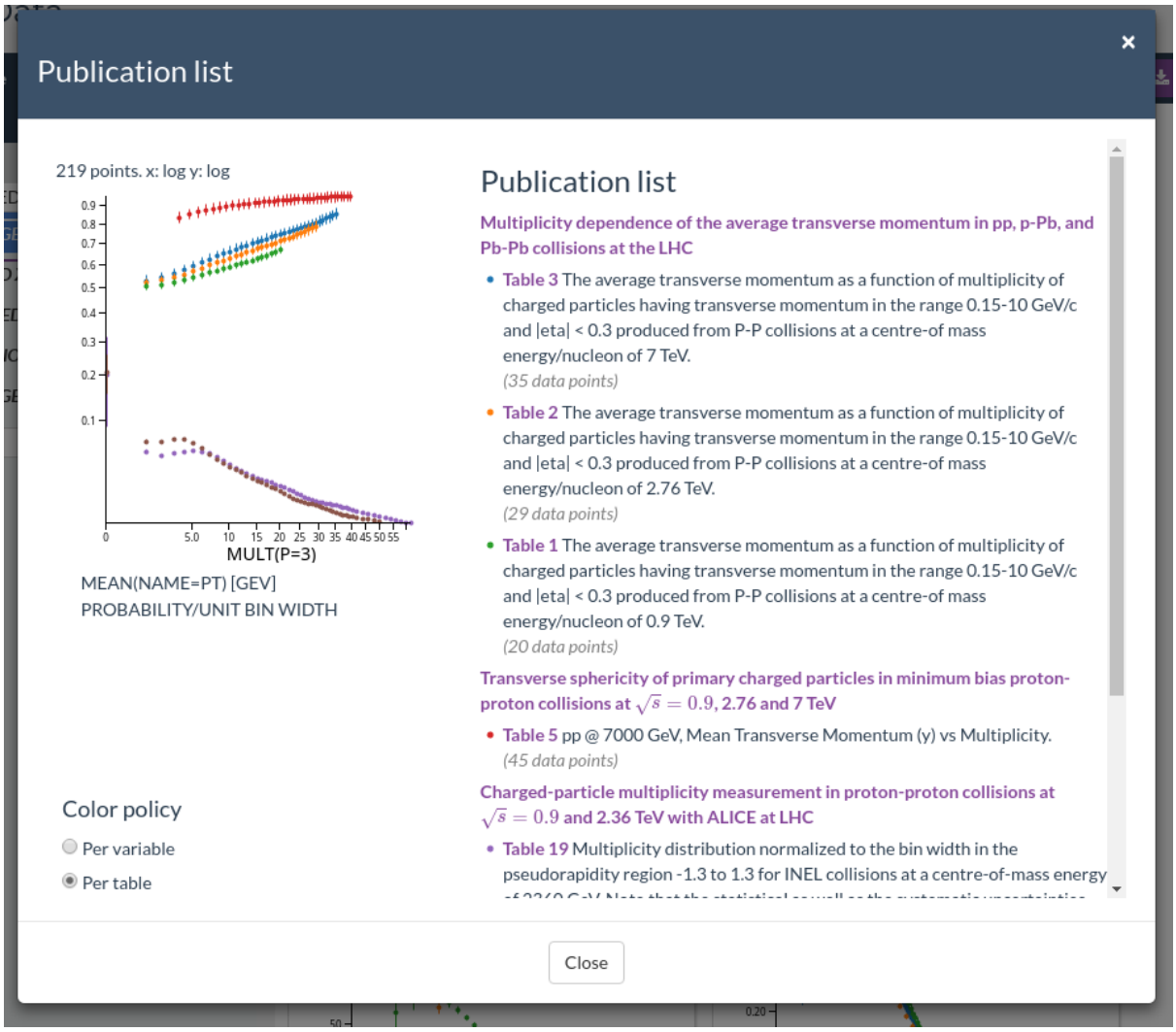

Figura 7.10: Vista de lista de publicaciones

Por defecto en esta vista los gráficos se representan con un color por tabla.

La lista incluye enlaces a los artículos en el sitio principal de HEPData donde se pueden ver los detalles del contenido, las tablas con los datos crudos, extraer citas, etc.

7.6 Exportación de gráficos

Cada gráfico incluye un botón de descarga que exporta el contenido del gráfico en formato YAML apto para que se le pueda dar un tratamiento posterior con un programa informático.

El documento exportado es una estructura jerárquica con los siguientes datos:

- Tablas de origen. Aparecen listadas las tablas de las que salen los datos de este gráfico. De cada una de ellas se incluye:
  - El título de su artículo, su código de registro en INSPIRE y su DOI.
  - El número de tabla dentro del artículo.
  - La descripción adjunta a la tabla.
  - El DOI de la tabla
- Nombre de la variable x.
- Para cada variable y:
  - Nombre de la variable y.
  - Datos, codificados como puntos (x,y). Cada punto incluye además la magnitud de su error superior e inferior, tanto para x como para y y el DOI de la tabla de la que procede ese dato.

También es posible hacer uso de la función *Download all plots* situada en la barra superior para descargar en un solo archivo todos los gráficos. Esta función añade además una cabecera en la que incluye un volcado del filtro con el que se obtuvieron los datos y la URL a la aplicación donde pueden verse de nuevo los datos con los mismos filtros y gráficos.

```
tables:
  - publication:
      title: 'Multiplicity dependence of the average transverse momentum in pp,
Pb-Pb collisions at the LHC'
      inspire_record: 1241423
      hepdata_doi: 10.17182/hepdata.62298.v1
    description: >
      The average transverse momentum as a function of multiplicity of charged
      particles     having transverse momentum in the range 0.15-10 GeV/c and
      |eta| < 0.3 produced from P-P collisions      at a centre-of mass
      energy/nucleon of 7 TeV.
    table_num: 3
    hepdata_doi: 10.17182/hepdata.62298.v1/t3
  - publication:
      title: 'Multiplicity dependence of the average transverse momentum in pp,
Pb-Pb collisions at the LHC'
      inspire_record: 1241423
      hepdata_doi: 10.17182/hepdata.62298.v1
    description: >
      The average transverse momentum as a function of multiplicity of charged
      particles     having transverse momentum in the range 0.15-10 GeV/c and
      |eta| < 0.3 produced from P-P collisions      at a centre-of mass
      energy/nucleon of 2.76 TeV.
    table_num: 2
    hepdata_doi: 10.17182/hepdata.62298.v1/t2
x_var:
  name: MULT(P=3)
y_vars:
  - name: 'MEAN(NAME=PT) [GEV]'
    data_points:
      - x: 1
        x_low: 0.5
        x_high: 1.5
        'y': 0.528
        y_low: 0.5043997436454601
        y_high: 0.5516002563545399
        hepdata_doi: 10.17182/hepdata.62298.v1/t3
      - x: 2
        x_low: 1.5
        x_high: 2.5
        'y': 0.541
```

Figura 7.11: Aspecto de un gráfico exportado desde HEPData Explore.

```yaml
hepdata-explore-uri: 'http://hepdata.rufian.eu/#FVu8s'
filter:
  type: AllFilter
  params:
    children:
      - type: IndepVarFilter
        params:
          field: indep_vars.name
          value: 'M(NEUTRALINO1) [GEV]'
      - type: DepVarFilter
        params:
          field: dep_vars.name
          value: EFFICIENCY
plots:
  - tables:
      - publication:
          title: "Search for direct pair production of a chargino and a
neutralino decaying to the 125\_GeV Higgs boson in $\\sqrt{s} = 8$ \_TeV ${pp}$
collisions with the ATLAS detector"
          inspire_record: 1341609
          hepdata_doi: 10.17182/hepdata.68567.v1
        description: "Efficiency for the same-sign $\\mu\\mu$ channel with two
or three jets.\n"
        table_num: 62
        hepdata_doi: 10.17182/hepdata.68567.v1/t62
      - publication:
          title: "Search for direct pair production of a chargino and a
neutralino decaying to the 125\_GeV Higgs boson in $\\sqrt{s} = 8$ \_TeV ${pp}$
collisions with the ATLAS detector"
```

Figura 7.12: Aspecto de la cabecera de un volcado de todos los gráficos, que incluye la URL de la aplicación.

CAPÍTULO 8

Persistencia de estados

El estado en HEPData Explore es transferible: Un usuario puede compartir sus filtros y gráficos con otros usuarios simplemente copiando y pegando la URL de la aplicación.

Cada vez que un usuario realiza una acción, como añadir un nuevo filtro o modificar un gráfico, se crea un volcado del estado de la interfaz en formato JSON. Ese volcado de estado se envía a un servidor, donde se enlaza con una URL corta como la siguiente:

http://hepdata.rufian.eu/#FVu8s

El código después del carácter # hace referencia a un registro en la base de datos del servidor que contiene el estado de la aplicación en el momento de ser compartida.

Este tema explica brevemente cómo se ha diseñado e implementado este sistema, así como sus implicaciones.

8.1 Serialización de estado

El estado de la aplicación se calcula en una propiedad observable, `app.appState`, que lo devuelve como un objeto POJO. En particular contiene el volcado del árbol de filtros (hecho con `app.rootFilter.dump()`) y para cada gráfico, el volcado de su configuración `plot.dump()`.

```
> JSON.stringify(app.appState, null, 2)
< "{
    "version": 2,
    "filter": {
      "type": "AllFilter",
      "params": {
        "children": [
          {
            "type": "IndepVarFilter",
            "params": {
              "field": "indep_vars.name",
              "value": "M(NEUTRALINO1) [GEV]"
            }
          },
          {
            "type": "DepVarFilter",
            "params": {
              "field": "dep_vars.name",
              "value": "EFFICIENCY"
            }
          }
        ]
      }
    },
    "plots": [
      {
        "pinned": false,
        "xVar": "M(NEUTRALINO1) [GEV]",
        "yVars": [
          {
            "name": "EFFICIENCY",
            "color": "#d62728"
          }
        ],
        "colorPolicy": "per-table"
      },
```

Figura 8.1: Volcado del estado de la aplicación en formato legible.

8.2 Deduplicación de estados

La serialización de estados se ha diseñado de forma que n interacciones desde un mismo estado inicial s_0 hechas por distintos usuarios siempre produzcan los mismos estados $s_1, s_2, ..., s_n$, con exactamente el mismo volcado, byte a byte.

Para ello, la exportación a JSON se hace con un generador específico que emite las claves de los objetos en orden alfabético. Concretamente se usa la biblioteca stable-stringify.js[25] del proyecto Snorky.

No solo los volcados son exactamente iguales, sino que los identificadores con los que se enlazan se calculan mediante una función hash del volcado. Así pues, un mismo volcado siempre recibe el mismo identificador.

De esta manera las interacciones más comunes, como añadir un filtro muy popular nada más abrir la aplicación, solo ocupan espacio en la base de datos la primera vez que se realizan, reutilizándose en todos los demás casos.

8.3 Software del lado del servidor

Los estados se almacenan en un pequeño servicio web propio llamado `kv-server`. Su nombre se debe a que guarda fundamentalmente pares clave-valor, sin realizar un análisis profundo del sentido de esos datos. La interpretación del volcado de los estados se deja íntegramente a la aplicación del lado del cliente.

El servidor `kv-server` está implementado con el framework web para Python Flask[26] y el ORM SQLAlchemy[27]. Su código fuente está alojado en el repositorio de HEPData Explore.

La principal ventaja de haber utilizado SQL Alchemy es que el sistema gestor de base de datos a utilizar se convierte en un parámetro de configuración [28] en vez de una decisión de diseño irrevocable.

De todos los sistemas de bases de datos soportados, el más simple es SQLite, puesto que no requiere instalar software adicional. Además, tiene un rendimiento aceptable para esta aplicación y migrar la base de datos de un servidor a otro es tan simple como copiar un archivo.

[25] https://github.com/ntrrgc/snorky/blob/6181bdf/snorky/frontend/stable-stringify.js
[26] http://flask.pocoo.org/
[27] http://www.sqlalchemy.org/
[28] Lista de sistemas de bases de datos soportados: http://docs.sqlalchemy.org/en/latest/dialects/index.html

Para tener un medio contra posibles abusos, además del identificador y el volcado, el servidor también almacena la IP del cliente que solicitó almacenar el estado y la fecha y hora en la que se solicitó.

El diseño del servidor hace fácil replicar y particionar la base de datos de manera que, si se deseara, se podría ofrecer el servicio con alta disponibilidad haciendo pocas modificaciones a la aplicación.

8.4 Historial del navegador

La actualización de la URL después de un cambio de estado se hace a través de la API de historial[29] que ofrecen los navegadores.

Como consecuencia, cada cambio de estado es interpretado como un cambio de página, de manera que un usuario puede usar los botones *atrás* y *adelante* de su navegador (y sus correspondientes atajos de teclado) para deshacer y rehacer acciones en la aplicación.

Además, también es posible hacer uso de los marcadores del navegador para guardar una búsqueda para otro día o recuperar un búsqueda de una pestaña cerrada recientemente utilizando las funciones de los navegadores como el atajo `Ctrl+Shift+T`.

Para que esta funcionalidad funcione de manera previsible ha sido necesario desarrollar la aplicación de manera que cada acción provoque un (y sólo un) cambio de estado persistible, sin que haya estados intermedios que puedan crear ciclos al intentar volver atrás.

8.5 Migraciones

Los estados incluyen un campo `version` que permite distinguir estados creados en versiones anteriores de la aplicación y aplicarles transformaciones hasta migrarlos (actualizarlos) a la versión actual.

En el archivo `migrateStateDump.ts` se puede ver un ejemplo de este tipo de migración para el cambio de la versión 1 a la 2, en la que se añadió al volcado el color de cada variable de los gráficos de manera que no cambiaran al compartir el estado.

La función definida en este archivo añade un color a cada variable y devuelve un volcado

[29] https://developer.mozilla.org/en/docs/Web/API/History

compatible con la versión 2. A partir de entonces la aplicación puede deserializarlo normalmente y proceder como si fuera un estado creado en la última versión.

CAPÍTULO 9

Herramientas y técnicas de programación empleadas

En este capítulo se explican algunas de las herramientas utilizadas en HEPData Explore para hacer el desarrollo más manejable y productivo.

Las técnicas y herramientas aquí tratadas son genéricas y afectan a un gran número de aplicaciones. El objetivo de este capítulo es explicar en líneas generales su motivación y las implicaciones que han tenido en este proyecto.

Este es un capítulo técnico. Incluye explicaciones y ejemplos que serán de gran ayuda para el lector que quiera entender el código de HEPData Explore.

9.1 TypeScript

Anticipando la magnitud del proyecto y la necesidad de cambios en el futuro, se apostó por escribir la nueva aplicación en TypeScript[30] para tener autocompletado y comprobación de tipos en la mayor parte del proyecto.

TypeScript es un lenguaje que extiende JavaScript con anotaciones de tipos, de manera que es capaz de detectar en tiempo de compilación en qué partes del código se está llamando a una función con argumentos incorrectos y mostrar un error al programador. Este tipo de ayuda es extremadamente valiosa a la hora de desarrollar código de interfaz, donde probar un componente requiere varios pasos.

Usar TypeScript fue un completo acierto para el proyecto:

[30] https://www.typescriptlang.org/

Por un lado, el compilador fue capaz de encontrar con rapidez una gran cantidad de errores que de otra manera habrían pasado desapercibidos. En particular, con el modo `strictNullChecks` introducido muy recientemente, TypeScript es capaz de detectar cuando un argumento puede ser `null` o `undefined` y dar error si la función no admite estos valores, eliminando una clase de errores muy difíciles de encontrar que en los sistemas de tipos de la mayoría de lenguajes pasa desapercibida. [31]

Por otro lado, el sistema de tipos permitió hacer refactorizaciones que de otro modo ni siquiera habrían sido planteadas debido al enorme riesgo de regresiones y la dificultad de encontrarlas. Al mismo tiempo, el autocompletado ahorró mucho tiempo a la hora de navegar por estructuras complejas.

El motor de inferencia de TypeScript es lo suficientemente potente para que anotar el código requiera muy poco esfuerzo. Habitualmente solo es necesario indicar tipos en los métodos y las propiedades de las clases – donde de otra forma deberían figurar como documentación. La figura 9.1 muestra un ejemplo de esta inferencia.

Figura 9.1: A pesar de que esta función no contiene ninguna anotación de tipos, TypeScript es capaz de inferir que `groups` contiene un mapa de pares `[Publication, PublicationTable[]]` y que `map` pasará estos pares a la función lambda de la imagen.

TypeScript también permitió acceder a nuevas funcionalidades como los módulos ES6, que

[31] Esta garantía recibe el nombre *null safety*. https://kotlinlang.org/docs/reference/null-safety.html

permitieron modularizar la aplicación de mejor manera que con otras alternativas, o los decoradores, que se introdujeron gradualmente como azúcar sintáctico pero que acabaron teniendo repercusiones muy positivas en la calidad interna de la aplicación y eliminaron clases de errores, como la inicialización tardía de observables.

9.2 Propiedades observables

Knockout define un objeto *observable* que funciona como una variable a la que se le pueden escribir y leer valores, pero que además permite suscribirse a él para recibir notificaciones cuando su valor es modificado.

Esta capacidad fue combinada en HEPData Explore con los decoradores de TypeScript para crear propiedades observables, definidas con el decorador `@observable`.

```
// PlotConfig es un clase con dos propiedades, xVar e yVars.
class PlotConfig {
    @observable()
    xVar: string|null = null;
    @observable()
    yVars: string[] = [];
}

var config = new PlotConfig();
// xVar funciona como una propiedad normal
config.xVar = 'PT';

// Sin embargo, tiene algo especial... podemos suscribirnos a ella
ko.getObservable(config, 'xVar').subscribe(newValue => {
    console.log('xVar ha cambiado: ' + newValue);
})
config.xVar = 'M(NEUTRALINO)'; // xVar ha cambiado: M(NEUTRALINO)
config.xVar = 'M(NEUTRALINO)'; // ningún mensaje (no hay cambio)
config.xVar = 'M(GLUINO)';     // xVar ha cambiado: M(GLUINO)
```

Estas propiedades son muy importantes porque permiten desacoplar la interfaz de usuario de los datos. Los componentes de interfaz pueden suscribirse a las propiedades observables de los datos de manera que siempre muestran el último valor.

Si no hubiera observables, en cada lugar del código en el que modificaran datos susceptibles de ser mostrados en la interfaz tendría que escribirse el código para hacer esas modificaciones *en cada uno* de los lugares de la interfaz que pudieran necesitar un refresco. Además, es fácil

que al añadir un componente de interfaz nos olvidemos de modificar manualmente el código de actualización, mientras que los observables no requerirían ningún cambio en este caso.

Las propiedades observables son una implementación del patrón Observer[32].

9.3 Propiedades calculadas y detección de dependencias

Además del observable básico visto en el punto anterior, Knockout ofrece otro tipo de observable muy útil, el observable calculado [33], que funciona como una propiedad calculada con detección de dependencias.

De forma similar, se creó un decorador `@computedObservable` para encapsular estos observables en la interfaz de una clase. Por ejemplo:

```
class Person {
    @observable()
    firstName: string = '';

    @observable()
    lastName: string = '';

    @computedObservable()
    get fullName() {
        return this.firstName + ' ' + this.lastName;
    }
}

var person = new Person();
person.firstName = 'Juan';
person.lastName = 'García';

// fullName funciona como una propiedad calculada
console.log(person.fullName); // 'Juan García'

// Pero además, es a su vez un observable
ko.getObservable(person, 'fullName').subscribe(newValue => {
    console.log(newValue);
});

// Al cambiar el valor de una dependencia, automáticamente
```

[32] https://en.wikipedia.org/wiki/Observer_pattern
[33] http://knockoutjs.com/documentation/computedObservables.html

```
// se recalcula la propiedad calculada y si su valor
// es distinto al de la ejecución anterior, se notifica
// a los suscriptores.
person.firstName = 'Antonio'; // 'Antonio García'
```

La detección de dependencias es dinámica. La propiedad solo se recalcula cuando alguno de los observables utilizados para generar su valor anterior es modificado. Se permite recursividad, por lo que unas propiedades calculadas pueden ser a su vez dependencias de otras.

Esta detección de dependencias resultó ser un gran punto fuerte para este proyecto. Los componentes de la interfaz declaran con estas propiedades qué datos necesitan mostrar y solo cuando realmente esos datos son modificados, se actualizan de forma automática.

Esta característica se usa intensivamente en HEPData Explore. Así, por ejemplo, `app.filterDump` es una propiedad calculada que contiene el resultado de serializar el filtro raíz. Cada vez que cualquier hijo, nieto, etc. de este filtro es creado, modificado o eliminado esta propiedad se recalcula y notifica todos sus suscriptores.

Esta notificación es recibida por otros componentes de la aplicación. No solo hay suscriptores que modifican la interfaz para reflejar los nuevos parámetros, sino que también hay suscriptores que envían la consulta para descargar las nuevas tablas y para descargar información de autocompletado actualizada.

Como consecuencia, la gestión de eventos en la aplicación se simplifica de forma notable. El código que necesita modificar filtros modifica filtros, sin acoplamiento con el código que necesita mostrar esos filtros en la interfaz o convertirlos en consultas de ElasticSearch.

El control programático de la aplicación también se simplifica. Así, por ejemplo, la siguiente línea, cuando se introduce desde las herramientas de desarrollador del navegador, añade al filtro raíz un nuevo hijo, un filtro por nivel de energía con el rango 450-550 GeV. Automáticamente la interfaz refleja el cambio y se hace la búsqueda.

```
app.rootFilter.children.push(new CMEnergiesFilter(450, 550))
```

Por otro lado, a veces es necesario codificar las actualizaciones de las propiedades observables con cuidado para evitar que estas reacciones en cadena no se den más veces de las necesarias. Por ejemplo, si se necesitara añadir varios filtros hijo simultáneamente, no se harían varias llamadas a `push` sino que se reemplazaría el array `children` con uno nuevo.

9.4 Flujos observables asíncronos

La gestión de dependencias explicada hasta ahora está pensada para código síncrono (que tiene efecto de forma inmediata), como es la actualización de componentes de la interfaz. Sin embargo, por sí sola se queda corta cuando la actualización requiere de datos que se obtendrán en el futuro como respuesta de una petición a un servidor.

Las acciones asíncronas como son las peticiones a servidores introducen un gran nivel de complejidad en la aplicación, por varios motivos:

- La petición puede funcionar o puede fallar. El fallo puede producirse en el lado cliente o en el lado servidor.

  Es necesario informar al usuario de esta situación para que la aplicación no se quede *colgada*. También es importante que el fallo sea controlado de manera que problemas en acciones pasadas no impidan acciones futuras.

- Los fallos pueden ser ocasionales y transitorios, por ejemplo, por un problema de conectividad de la red del usuario. Una funcionalidad automática de reintentos puede ser necesaria o deseable.

- El usuario puede modificar la interfaz mientras una petición está en curso, de manera que puede que antes de que llegue una respuesta a una petición sus datos ya sean irrelevantes.

  Bloquear la interfaz durante el trascurso de las peticiones simplificaría la interfaz, pero tendría un efecto muy negativo en la usabilidad de la aplicación.

- Las respuestas a dos peticiones seguidas pueden llegar desordenadas, pudiendo causar condiciones de carrera si el código no está preparado para soportarlo.

  Por ejemplo, el usuario selecciona en la interfaz el filtro A (causando la petición A), después se arrepiente y selecciona el filtro B (causando la petición B). Un programa sencillo puede tener un código que haga la petición y al recibir la respuesta actualice la interfaz. Sin embargo, si el servidor responde antes la petición B (por ejemplo, porque se calcula de forma más rápida) que la A, el usuario obtendría datos erróneos.

- Algunas operaciones requieren de otras para completarse, posiblemente varias de ellas.

  Por ejemplo, para mostrar sugerencias de autocompletado es necesario por un lado tener la entrada de texto del usuario (que se actualiza cada vez que pulsa una tecla) y por

otro lado una lista de valores aceptables, que se descarga cada vez que los demás filtros son modificados. Hasta que no se resuelvan ambas dependencias no pueden mostrarse sugerencias de forma correcta.

Además, la interfaz debe reaccionar de forma automática cada vez que una de las dos dependencias cambie; es decir, que el usuario teclee o se reciba una lista de sugerencias actualizada.

Estos requisitos no son obvios pero son imprescindibles para que la aplicación sea estable y funcional. Satisfacerlos con programación imperativa requiere introducir una gran cantidad de estado en el código de la aplicación en forma de variables con resultados temporales, banderas (*flags*) de progreso, eventos y código condicional. El resultado puede ser bastante lioso y omisiones difíciles de detectar pueden causar cualquiera de los problemas antes comentados.

9.4.1 RxJS

Alternativamente, para hacer frente a las dificultades de la programación asíncrona en interfaces de usuario de forma más limpia se ha hecho uso de RxJS[34].

RxJS implementa un estilo de programación basado en unas entidades llamadas *flujos observables* (*observable streams*).

Un flujo observable es un objeto que emite valores a lo largo del tiempo. El flujo puede ser finito (ej. una petición a un servidor termina cuando se recibe una respuesta) o infinito (un flujo de eventos puede vivir tanto como la aplicación). En caso de ser finito, un flujo puede finalizar con éxito o con error (por ejemplo, si la petición falló).

Los flujos observables contienen métodos (operadores) que se conectan al flujo origen para devolver un flujo nuevo que aplica una transformación. Por ejemplo, el nuevo flujo podría contener solo algunos valores del flujo original, haberlos convertido de cadena a entero, haber convertido errores en mensajes de error, etc. RxJS incorpora un repositorio con operadores para operaciones comunes al trabajar con datos asíncronos.

9.4.2 Ejemplo de flujo asíncrono con RxJS

Por ejemplo, supongamos un caso simplificado de autocompletado – en HEPData Explore no funciona exactamente así, se trata de un ejemplo didáctico:

[34]https://github.com/Reactive-Extensions/RxJS

Queremos que un usuario pueda escribir en un campo de texto y que según escriba reciba sugerencias, para lo que puede ser necesario hacer una petición a un servidor externo que recibe la cadena del usuario y devuelve la lista de sugerencias.

Para implementar este ejemplo con RxJS, podemos partir de un flujo inicial que emite un valor cada vez que el usuario modifica el campo de texto. A este flujo inicial queremos aplicarle operadores en cadena para eliminar los espacios al principio y al final, filtrar los valores con menos de 3 caracteres, filtrar aquellos valores que sean iguales al anterior (ej. porque el usuario tecleó un espacio), hacer una petición a un servicio web con cada uno de estos valores diferentes, soportar automáticamente hasta 3 reintentos, descartar resultados de peticiones resueltas fuera de orden, mostrar un icono de carga hasta que la petición se resuelva y finalmente mostrar el resultado de la petición en la interfaz.

RxJS permite hacer todo esto sin necesidad de almacenar estado intermedio en variables.

En primer lugar necesitamos el flujo de inicio. RxJS es compatible con los observables de knockout. Utilizando el método `toObservableWithLatest` de `KnockoutObservable` obtenemos un flujo observable que se inicializa con el valor de dicho observable y que emite un nuevo valor con cada actualización. En este caso supondremos que tenemos una propiedad observable llamada `userQuery`.

```
const userQueryStream = (<KnockoutObservable<string>>
    ko.getObservable(this, 'userQuery'))
    .toObservableWithReplyLatest();
```

También necesitamos una función que haga la petición, por ejemplo:

```
function requestSuggestions(query: string): Promise<string[]> {
    return jsonPOST('/autocomplete', {query: query});
}
```

Esta función crea un objeto promesa (`Promise`) que se devuelve instantáneamente y que emite un evento cuando se recibe una respuesta para la petición. Las promesas son un estándar de JavaScript [35]. `jsonPOST` es una función de la API de HEPData Explore que hace peticiones HTTP con contenido en formato JSON.

Este sería el código de nuestra aplicación hipotética:

```
userQueryStream
    // Eliminamos espacios al principio y al final
    .map(query => query.trim())
```

[35] El estándar se denomina *Promises/A+*. Los navegadores actuales lo implementan, aunque se pueden usar bibliotecas adicionales para obtener más funcionalidad. https://promisesaplus.com/

```
    // Omitimos las búsquedas con menos de tres caracteres
    .filter(query => query.length >= 3)
    // El usuario puede teclear muy rápido. Para no sobrecargar
    // el servidor en vano, esperaremos 50 ms antes de pasar
    // una consulta al siguiente flujo.
    // Si en esos 50 ms recibimos otra consulta, descartaremos
    // la anterior y volveremos a esperar otros 50 ms por esta.
    // Y así sucesivamente.
    .debounce(50)
    // Omitimos una búsqueda si es igual que la inmediatamente
    // anterior
    .distinctUntilChanged()
    // Esta parte es un poco más complicada...
    // Convertimos cada cadena de consulta en un nuevo flujo
    // que emite el resultado de una determinada búsqueda.
    // Este operador devuelve por tanto un flujo de flujos.
    .map(query =>
        // Convertimos cada consulta en un nuevo flujo
        // que emite el resultado de una petición.
        Rx.Observable.defer(() =>
            // Haz una petición a un servicio web...
            requestSuggestions(query)
        )
        // En caso de error, reintenta la petición tres veces
        // antes de propagar el error.
        .retry(3)
        // Al recibir una respuesta, la convertimos al formato
        // que necesitemos para nuestra interfaz.
        .map(response => ({
            suggestions: response,
            hasError: false,
        }))
        // Si por el contrario recibimos un error, incluso
        // después de los tres reintentos, transformamos el
        // error a un formato apto para nuestra interfaz.
        .catch(error => Rx.Observable.just({
            suggestions: [],
            hasError: true,
        }))
    )
    // Acabamos de lanzar una petición, activaremos un icono
    // de "cargando" en nuestra interfaz.
    .do(() => {
        this.loadingSuggestions = true;
    })
```

```
    // Si el último valor emitido fue un flujo que aún
    // no ha completado y recibimos un nuevo flujo,
    // cancela el antiguo. De esta manera evitamos procesar
    // respuestas fuera de orden.
    // switch() emite el resultado de la última petición,
    // solo si no existe otra petición activa.
    .switch()
    // Finalmente, procesamos cada objeto recibido
    .subscribe(formattedResponse => {
        this.suggestions = formattedResponse.suggestions;
        this.autocompletionFailed = formattedResponse.hasError;

        // Acabamos de terminar de cargar las sugerencias,
        // y .switch() nos garantiza que no hay otras
        // peticiones en curso, por lo que podemos quitar
        // el icono de "cargando".
        this.loadingSuggestions = false;
    });
```

RxJS encapsula el estado de la mayoría de operaciones asíncronas que necesitamos dentro de los operadores de flujo. Si no hubiéramos usado RxJS habríamos tenido que utilizar variables externas para implementar muchas de estas operaciones, por ejemplo:

- Tiempo de la última pulsación de teclado.
- Petición en curso, con método de cancelación.
- Última consulta enviada.
- Número de reintentos.

RxJS nos libera de tener que introducir estas variables, solo necesitando aquellas realmente relacionadas con la interfaz, como la lista de sugerencias, el icono de cargando, etc. Además, aspectos habituales de la lógica de peticiones asíncronas son manejados de forma reutilizable por los operadores.

9.4.3 En HEPData Explore

En HEPData Explore los flujos observables se usan para descargar datos de las tablas, gestionar las sugerencias del sistema de autocompletado, y persistir los estados de la interfaz en el servidor, de forma similar al ejemplo aquí planteado.

Algunas flujos asíncronos sin embargo son más complejos, como es el caso de la persistencia de estados, ya que solo deben enviarse al servidor aquellos estados que representen una interacción completa.

Al cambiar un filtro se produce un cambio en el estado de la aplicación, pero no se considera interacción completa puesto que los gráficos aun utilizan variables que en los nuevos datos pueden ser irrelevantes. Solo cuando se reciben los nuevos datos y se generan los nuevos gráficos se considera la interacción completa y se envía una petición de subida de estado.

Por otro lado, modificar un gráfico sin afectar a los filtros produce una subida de estado sin necesidad de hacer ninguna petición al servidor.

CAPÍTULO 10

Uso de ElasticSearch en HEPData Explore

ElasticSearch es el motor de búsqueda que permite que HEPData Explore acepte consultas muy flexibles. Como ya fue introducido en Transformación e indexado, el diseño del mapeo y en particular la construcción de consultas precisas es una parte compleja pero vital de la aplicación.

Este capítulo introduce cómo se utiliza internamente ElasticSearch en HEPData Explore. Muchos de los conceptos aquí explicados son genéricos, por lo que este capítulo puede servir como una introducción para el lector que tenga interés en la construcción de aplicaciones de búsqueda.

En los siguientes ejemplos se muestra el aspecto que tienen las consultas de ElasticSearch utilizadas en HEPData Explore, utilizando no los datos reales – que su tamaño los hace poco prácticos para probar la corrección del sistema, sino los datos de prueba, con un esquema simplificado.

10.1 Creación del índice

En primer lugar es necesario declarar el mapeo a utilizar para el índice. En una ejecución normal esto lo hace el programa `server-aggregator`, pero para este caso nos basta con hacer dos peticiones HTTP usando Sense o cURL. Este paso solo es necesario hacerlo una vez.

```
POST /mini-test
{
  "mappings": {
    "publication": {
      "properties": {
```

```
        "name": {"type": "string"},
        "tables": {
          "type": "nested",
          "properties": {
            "var_x": {"type": "string", "index": "not_analyzed"},
            "var_y": {"type": "string", "index": "not_analyzed"}
          }
        }
      }
    }
  }
}
```

La petición anterior declara el tipo `publication` dentro del índice `mini-test` y define su mapeo, en el que se definen los campos que aceptará y cómo serán indexados.

El índice no es necesario que haya sido creado de forma explícita con anterioridad para que la petición funcione.

En este caso el mapeo consta de dos campos de primer nivel, `name`, con el nombre de la publicación, y `tables`, con cada una de las tablas que contiene. Cada tabla contiene dos atributos de tipo cadena, `var_x` y `var_y`, los cuales no son analizados como texto natural.

Nota: El mapeo anterior es un ejemplo muy simplificado para explicar cómo se utiliza ElasticSearch en la aplicación y omite muchos detalles del mapeo real.

En particular, `var_x` y `var_y` no existen en el mapeo real, sino que son `indep_vars` y `dep_vars`, y no son de tipo cadena sino de tipo objeto, donde el nombre de la variable es solo un atributo.

Es importante destacar que la propiedad `tables` se declara con tipo `nested`. Esto hace que cada tabla de una publicación se indexe de manera independiente. De esta manera se pueden hacer consultas del tipo "obtener las publicaciones que contengan una tabla con `var_x = A` y `var_y = B`" (una misma tabla tiene `var_x = A` y `var_y = B`" mientras que de otro modo solo podrían hacerse consultas como "obtener las publicaciones que contengan tablas con `var_x = A` y tablas con `var_x = B`" (una publicación con dos tablas que cumpla solo una condición en cada tabla sería escogida como resultado).

10.2 Carga de datos

El siguiente paso es cargar las tablas en ElasticSearch. En la aplicación final, este proceso lo hace también `server-aggregator`.

```
POST /mini-test/publication/1
{
  "name": "Example publication",
  "tables": [
    {
      "var_x": "M(GLUINO)",
      "var_y": "M(NEUTRALINO)"
    },
    {
      "var_x": "M(GLUON)",
      "var_y": "M(SQUARK)"
    }
  ]
}

POST /mini-test/publication/2
{
  "name": "Other publication",
  "tables": [
    {
      "var_x": "M(GLUINO)",
      "var_y": "M(SQUARK)"
    }
  ]
}
```

Las peticiones anteriores crean un objeto de tipo `publication` dentro del índice (base de datos) `mini-test` con los identificadores `1` y `2` respectivamente. Estos identificadores pueden ser utilizados para consultar las publicaciones individualmente, actualizarlas o borrarlas.

10.3 Búsqueda simple

A continuación, utilizamos el endpoint `_search` para obtener los resultados de una búsqueda.

```
POST /mini-test/publication/_search
{
```

```
  "filter": {
    "nested": {
      "path": "tables",
      "filter": {
        "term": {
          "tables.var_x": "M(GLUON)"
        }
      }
    }
  }
}
```

Esta consulta simple devuelve las publicaciones que contengan una tabla cuya cuyo valor de `var_x` sea `M(GLUON)`.

Nota: `term` es uno de los tipos de filtro soportados por ElasticSearch. Recibe uno o más pares clave-valor donde la clave es la ruta a una propiedad del objeto a filtrar y el valor define la cadena buscada.

`term` se cumple para aquellos objetos cuyas propiedades especificadas contengan los valores de la búsqueda. En el caso de los campos no analizados (aquellos en los que se ha utilizado `"index": "not_analyzed"`), esto requiere que el valor sea exactamente igual.

En campos analizados (con texto en lenguaje natural), ElasticSearch no indexa la cadena original sino cada una de las palabras del texto por separado, después de hacerles un procesamiento de homogeneización (eliminación de stop words, sustitución por lexema, sustitución de sinónimos si se especificara). En ese caso la cadena especificada en la búsqueda también recibe el mismo tratamiento.

ElasticSearch tiene otros tipos de filtro para toda clase de propósitos. Por ejemplo, el filtro `range` permite filtrar valores numéricos inferiores y/o superiores a unos dados y el filtro `regexp` permite hacer búsquedas con expresiones regulares, entre muchos otros.

`nested` también es un tipo de filtro, que devuelve aquellos objetos raíz cuyos objetos hijo anidados (en este caso tablas de una publicación) cumplen a su vez otro filtro. El filtro anidado puede hacer referencia a propiedades tanto del objeto hijo como de los objetos padre.

La respuesta de ElasticSearch tiene este aspecto:

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "mini-test",
        "_type": "publication",
        "_id": "1",
        "_score": 1,
        "_source": {
          "name": "Example publication",
          "tables": [
            {
              "var_x": "M(GLUINO)",
              "var_y": "M(NEUTRALINO)"
            },
            {
              "var_x": "M(GLUON)",
              "var_y": "M(SQUARK)"
            }
          ]
        }
      }
    ]
  }
}
```


Nótese que a pesar de que usamos `nested` para filtrar por las propiedades de las tablas individuales, ElasticSearch siempre devuelve el objeto del tipo especificado en la URL. ElasticSearch ha encontrado una tabla que cumple las condiciones especificadas (la segunda) dentro de "Example publication" y, puesto que la búsqueda es sobre el tipo `publication`, devuelve la publicación entera con *todas* sus tablas.

10.4 Obtener las tablas

Para nuestra aplicación nos interesan solo aquellas tablas que cumplan los criterios establecidos. El resultado devuelto por ElasticSearch en la consulta anterior nos obliga a hacer el filtrado dos veces: una vez en el lado servidor para encontrar las publicaciones (hecho por ElasticSearch) y otra en el lado del cliente para filtrar las tablas que no cumplen los filtros.

Inicialmente así fue como se solucionó el problema en la aplicación. Puesto que ya existía una jerarquía de clases para todos los filtros capaz de emitir consultas en el lenguaje de ElasticSearch, no fue muy difícil añadir un método de filtrado para ejecutar la mayoría de esos mismos filtros en el lado cliente.

La excepción fue un filtro con soporte de expresiones regulares. Si bien enviar una expresión regular en el lenguaje soportado por ElasticSearch es trivial, escribir en el lado cliente un código capaz de validar si una cadena de texto cumple una expresión regular con ese mismo lenguaje es una tarea muy ardua, pues requiere un conocimiento absoluto del lenguaje específico de las expresiones regulares usadas por ElasticSearch para poder mapearlas al lenguaje específico de expresiones regulares del cliente, en este caso JavaScript.

Nota: Aunque tanto ElasticSearch como JavaScript, al igual que muchos otros lenguajes, soportan expresiones regulares, existen pequeñas diferencias entre ellos que hacen que una expresión regular válida en un lenguaje no sea válida o produzca resultados diferentes en otro.

Por este motivo, y puesto que no hay una variante estándar, se puede considerar que cada implementación tiene su propio lenguaje de expresiones regulares.

En este sitio web se puede encontrar una compilación de las diferencias de sintaxis entre distintos lenguajes de expresiones regulares: http://www.regular-expressions.info/refbasic.html

---

Afortunadamente, ElasticSearch tiene una funcionalidad para registrar qué objetos anidados (con tipo `nested`) cumplieron un filtro y devolverlos en una lista separada, junto a su objeto padre. Esta funcionalidad se utiliza añadiendo una clave `inner_hits` al filtro `nested`. Como valor, `inner_hits` recibe un objeto con opciones, que en el caso más básico, está vacío.

Así, la consulta del ejemplo anterior quedaría así después de añadirle `inner_hits`:

```
POST /mini-test/publication/_search
{
```

```
  "filter": {
    "nested": {
      "path": "tables",
      "filter": {
        "term": {
          "tables.var_x": "M(GLUON)"
        }
      },
      "inner_hits": {}
    }
  }
}
```

El resultado es el siguiente:

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "mini-test",
        "_type": "publication",
        "_id": "1",
        "_score": 1,
        "_source": {
          "name": "Example publication",
          "tables": [
            {
              "var_x": "M(GLUINO)",
              "var_y": "M(NEUTRALINO)"
            },
            {
              "var_x": "M(GLUON)",
              "var_y": "M(SQUARK)"
            }
          ]
        },
```

```
        "inner_hits": {
          "tables": {
            "hits": {
              "total": 1,
              "max_score": 1,
              "hits": [
                {
                  "_index": "mini-test",
                  "_type": "publication",
                  "_id": "1",
                  "_nested": {
                    "field": "tables",
                    "offset": 1
                  },
                  "_score": 1,
                  "_source": {
                    "var_x": "M(GLUON)",
                    "var_y": "M(SQUARK)"
                  }
                }
              ]
            }
          }
        }
      }
    ]
  }
}
```


Nótese que no se han eliminado tablas de los resultados. Nuevamente, la consulta sigue preguntando por publicaciones y en consecuencia recibe publicaciones, con todas sus tablas. Sin embargo, junto a la propiedad `_source` (aquella en la que ElasticSearch coloca el objeto original encontrado), aparece una nueva, `inner_hits`, que contiene una lista de las tablas que realmente pasaron los filtros.

En esta nueva lista se puede apreciar como efectivamente solo hay una tabla, aquella que tiene `M(GLUON)` en `var_x`. La aplicación cliente puede examinar este array para recoger las tablas concordantes.

10.5 Eliminar la redundancia

`inner_hits` soluciona el problema de encontrar los objetos hijos al tratar con filtros anidados. Sin embargo, lo hace a un precio: las respuesta es más larga puesto que contiene en el `_source` tanto las tablas concordantes como las no concordantes y después repite ya solo las tablas concordantes en `inner_hits`.

Para nuestra aplicación no necesitamos las tablas no concordantes. Sin embargo, sí necesitamos la información general de la publicación (en el ejemplo de este capítulo, se trata solo del campo `name`).

Dado que una publicación puede tener un gran número de tablas – algunas llegan a una veintena, las cuales a su vez pueden tener una gran cantidad de filas y normalmente solo unas pocas tablas cumplen los filtros, estamos utilizando la mayor parte del espacio de la respuesta (y del ancho de banda) enviando información que es descartada inmediatamente una vez recibida.

Una manera simple de atajar el problema sería eliminar `tables` de `_source`, de manera que solo quedara la lista dentro de `inner_hits` con las tablas coincidentes. Hacia el final del proyecto se descubrió que ElasticSearch cuenta con una funcionalidad para filtrar `_source` que permite hacer precisamente esto.

Utilizando esta funcionalidad, la consulta anterior quedaría así:

```
POST /mini-test/publication/_search
{
  "filter": {
    "nested": {
      "path": "tables",
      "filter": {
        "term": {
          "tables.var_x": "M(GLUON)"
        }
      },
      "inner_hits": {}
    }
  },
  "_source": {
    "exclude": ["tables"]
  }
}
```

La propiedad `_source` en la raíz del objeto de consulta indica que queremos modificar cómo se devuelve `_source` en los resultados: concretamente, pedimos que se excluya la propiedad `tables` del objeto publicación.

El resultado es el siguiente:


```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "mini-test",
        "_type": "publication",
        "_id": "1",
        "_score": 1,
        "_source": {
          "name": "Example publication"
        },
        "inner_hits": {
          "tables": {
            "hits": {
              "total": 1,
              "max_score": 1,
              "hits": [
                {
                  "_index": "mini-test",
                  "_type": "publication",
                  "_id": "1",
                  "_nested": {
                    "field": "tables",
                    "offset": 1
                  },
                  "_score": 1,
                  "_source": {
                    "var_x": "M(GLUON)",
                    "var_y": "M(SQUARK)"
                  }
```

```
                }
              ]
            }
          }
        }
      }
    ]
  }
}
```

Aunque en este ejemplo la respuesta sigue siendo más larga que en el primer caso, la mayoría de la sobrecarga en este caso está por la cabecera de `inner_hits`, que solo aparece una vez por publicación y sin saltos de línea y espacios es bastante pequeña. Por otro lado, podemos ver como en `_source` ya no aparece `tables`, que es el campo que más espacio consume en la aplicación real.

10.6 Filtros compuestos

La siguiente consulta hace uso de operadores lógicos para construir un filtro compuesto que selecciona aquellas publicaciones que contengan una tabla con `var_x = M(GLUINO)` y `var_y = M(SQUARK)`.

```
POST /mini-test/publication/_search

{
  "filter": {
    "nested": {
      "path": "tables",
      "filter": {
        "bool": {
          "must": [
            {
              "term": {
                "tables.var_x": "M(GLUINO)"
              }
            },
            {
              "term": {
                "tables.var_y": "M(SQUARK)"
              }
            }
```

```
          ]
        }
      }
    }
  }
}
```

En esta consulta se hace uso del filtro `bool` que permite construir expresiones booleanas a partir de filtros anidados. La propiedad `must` admite una lista de filtros. Todos ellos se deben cumplir para que el filtro `bool` marque un objeto como coincidente.

De forma similar, también existe una propiedad `should` (aquí no mostrada) que hace un `OR` lógico en vez de un `AND`. También existe `must_not` que funciona como `NOT OR` (el filtro falla si una de las condiciones se cumple).

Este es el resultado de la consulta anterior:

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "mini-test",
        "_type": "publication",
        "_id": "2",
        "_score": 1,
        "_source": {
          "name": "Other publication",
          "tables": [
            {
              "var_x": "M(GLUINO)",
              "var_y": "M(SQUARK)"
            }
          ]
        }
      }
```

```
    ]
  }
}
```

La consulta devuelve la única consulta que cumple las dos condiciones especificadas.

Puesto que `tables` es de tipo `nested`, las tablas son indexadas como documentos separados. Es por esto que no se devuelve la otra publicación ("Example publication"), que tiene una tabla con `var_x = M(GLUINO)` y otra con `var_y = M(SQUARK)`, pero que al ser tablas distintas ninguna de ellas pasa el filtro.

10.6.1 Agregaciones con ElasticSearch

El filtrado de datos es la principal funcionalidad de ElasticSearch que esta aplicación necesita, pero no la única.

Un aspecto muy importante del diseño de HEPData Explore – tanto en el prototipo anterior como en la nueva versión – son las sugerencias automáticas. Para que la aplicación sea agradable de usar, no basta con que un usuario pueda introducir manualmente, letra por letra, el nombre de una variable para filtrar por ella.

La aplicación debe ser capaz de ayudar al usuario a escribir el nombre de variable mostrando sugerencias y ofreciendo autocompletado.

Las sugerencias deben ordenarse por relevancia, y cuando el usuario aun no ha escrito ningún valor en el filtro el mejor criterio de relevancia es el número de tablas que concuerdan con el valor del filtro.

Necesitamos que el cliente sea capaz de de descargar una lista de alguna cualidad filtrable (por ejemplo, variable dependiente, variable independiente, reacción, observable...) ordenada por el número de tablas que la contienen.

Para esto no basta con hacer una consulta de filtro y contar los resultados, puesto que la respuesta de una consulta que devolviera todas las tablas sería enorme y muy lenta de procesar.

10.7 Agregaciones globales

Además de la propiedad `filter` explicada anteriormente para las búsquedas, ElasticSearch soporta agregaciones mediante la propiedad `aggs`. Algunas de las agregaciones que soporta son suma, media, máximo, mínimo... En este primer caso solo necesitamos la más básica, que viene por defecto, `doc_count`. Esta agregación devuelve el número de documentos (publicaciones y tablas) encontrados.

La siguiente consulta – simple pero poco realista, solicita un listado ordenado de la cuenta de publicaciones agrupada por nombre de publicación.

```
POST /mini-test/publication/_search
{
  "size": 0,
  "aggs": {
    "publications_per_name": {
      "terms": {
        "field": "name",
      }
    }
  }
}
```

El resultado es el siguiente:

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "publications_per_name": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
```

```
          "key": "publication",
          "doc_count": 2
        },
        {
          "key": "example",
          "doc_count": 1
        },
        {
          "key": "other",
          "doc_count": 1
        }
      ]
    }
  }
}
```

Puesto que `name` es un campo de texto natural, cada una de las palabras que conforman su valor se indexa de forma individual (es por esto que tanto el filtro como la agregación empleadas en este capítulo se llaman `terms`). En los campos que serán verdaderamente autocompletados esto no pasará porque no están analizados, sin embargo, en este apartado se trata de mostrar un caso de agregación simple para introducir la sintaxis.

Otro detalle importante es la propiedad `size`, que limita el número de resultados (publicaciones) que se devuelven como resultado de la búsqueda. Como no necesitamos devolver el contenido de ninguna de las publicaciones, lo establecemos a cero. Si no lo hiciéramos, adquiriría el valor predeterminado (10) y las diez publicaciones que ElasticSearch considerara más relevantes aparecerían dentro del array `hits`.

La parte interesante de la respuesta está en el objeto `agregations`. `aggregations` contiene un objeto anidado para cada agregación solicitada – en este caso solo `publications_per_name`, y dentro, el array `buckets` contiene cada grupo encontrado.

La propiedad `doc_count` indica cuántos documentos (en este caso publicaciones) entran en cada grupo. En este caso, dos publicaciones contienen la palabra *publication*, una contiene la palabra *example* y otra contiene la palabra *other*.

10.8 Agregaciones anidadas

Sin embargo, para nuestro caso de uso necesitamos agregaciones de tablas, no de publicaciones. Al igual que con los filtros, podemos utilizar la agregación `nested` para bajar un nivel en

la jerarquía y trabajar al nivel de objetos hijo.

La siguiente consulta muestra un ejemplo realista en el que se pide una lista de todas las variables dependientes registradas en el sistema, ordenadas por número de tablas.

```
POST /mini-test/publication/_search
{
  "size": 0,
  "aggs": {
    "agg_tables": {
      "nested": {
        "path": "tables"
      },
      "aggs": {
        "per_var_y": {
          "terms": {
            "field": "tables.var_y",
            "size": 10000
          }
        }
      }
    }
  }
}
```

El resultado se muestra a continuación. Es muy similar al anterior, con la diferencia de que los grupos (`buckets`) se encuentran dentro de la agregación anidada.

```
{
  "took": 3,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 2,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "agg_tables": {
      "doc_count": 3,
      "per_var_y": {
```

```
        "doc_count_error_upper_bound": 0,
        "sum_other_doc_count": 0,
        "buckets": [
          {
            "key": "M(SQUARK)",
            "doc_count": 2
          },
          {
            "key": "M(NEUTRALINO)",
            "doc_count": 1
          }
        ]
      }
    }
  }
}
```

10.9 Agregaciones filtradas

Para crear una interfaz más usable es importante que las sugerencias tengan en cuenta el contexto.

Por ejemplo, si dentro de un filtro compuesto de tipo `AND` el usuario ha especificado como primera condición que la variable independiente sea `M(NEUTRALINO)` y después procede a crear un filtro adicional de variable dependiente, la aplicación sería mucho más útil si sugiriera solo aquellas variables que están en tablas donde se cumple el otro filtro.

Para lograr esto, necesitamos utilizar filtros con las agregaciones. ElasticSearch implementa esto a dos niveles.

En primer lugar, el objeto raíz de la consulta acepta una propiedad `query` que permite aplicar un filtro antes de realizar el cómputo de la agregación. De esta manera, la siguiente consulta aplica la agregación solo a *Example publication*, puesto que es la única que cumple el filtro establecido (contener una tabla con `var_x = M(GLUON)`):

Nota: Las propiedades `query` y `filter` del objeto consulta de ElasticSearch son muy similares: ambas aceptan un objeto filtro que aplican para seleccionar un subconjunto de los documentos disponibles. Sin embargo, entre ellas hay pequeñas diferencias.

Figura 10.1: Sugerencias contextuales en HEPData Explore. Las variables dependientes mostradas son aquellas que están presentes en las tablas con datos sobre el neutralino.

La principal diferencia es que `query` ejecuta algoritmos para puntuar los resultados de la búsqueda y `filter` no, por lo que el segundo es más eficiente si no necesitamos resultados ordenados (por ejemplo, si no trabajamos con texto en lenguaje natural).

Otra diferencia más sutil es que `filter` no afecta al contexto de agregación, por lo que si lo usáramos en lugar de `query` obtendríamos siempre los mismos resultados incluso aunque ninguna publicación pasara el filtro. Esta es la razón por la que aquí usamos `query` en lugar de `filter`.

```
POST /mini-test/publication/_search
{
  "size": 0,
  "query": {
    "nested": {
      "path": "tables",
      "query": {
        "term": {
          "tables.var_x": "M(GLUON)"
        }
      }
    }
  },
```

```
  "aggs": {
    "agg_tables": {
      "nested": {
        "path": "tables"
      },
      "aggs": {
        "per_var_y": {
          "terms": {
            "field": "tables.var_y",
            "size": 10000
          }
        }
      }
    }
  }
}
```

El resultado es el siguiente:

```
{
  "took": 2,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "agg_tables": {
      "doc_count": 2,
      "per_var_y": {
        "doc_count_error_upper_bound": 0,
        "sum_other_doc_count": 0,
        "buckets": [
          {
            "key": "M(NEUTRALINO)",
            "doc_count": 1
          },
          {
            "key": "M(SQUARK)",
```

```
            "doc_count": 1
          }
        ]
      }
    }
  }
}
```

En `hits.total` podemos ver que el filtro efectivamente se ha ejecutado y ha devuelto en total un documento (publicación) cuyo contenido ha sido omitido de la respuesta debido a que establecimos `size` a cero.

En `aggregations` se puede ver los valores de `var_y` en la única publicación que ha pasado el filtro.

10.10 Agrupaciones con filtrado anidado

Nuestra intención inicial era hacer un ranking de tablas que cumplieran unos criterios, no de publicaciones. Sin embargo, una vez ejecutado el filtro de la raíz de la consulta se ha perdido la información de qué tablas cumplían el filtro anidado, puesto que ésta devuelve publicaciones (aunque haga uso de `nested`).

Para obtener resultados agregados sobre sólo aquellas tablas que cumplan unos determinados criterios está a nuestra disposición la agregación `filter`. Repitiendo dentro de ella el filtro obtendremos una agregación con los datos anidados filtrados. A continuación se muestra un ejemplo de cómo se usa:

```
POST /mini-test/publication/_search
{
  "size": 0,
  "query": {
    "nested": {
      "path": "tables",
      "query": {
        "term": {
          "tables.var_x": "M(GLUON)"
        }
      }
    }
  },
  "aggs": {
```

```
    "agg_tables": {
      "nested": {
        "path": "tables"
      },
      "aggs": {
        "filtered_tables": {
          "filter": {
            "term": {
              "tables.var_x": "M(GLUON)"
            }
          },
          "aggs": {
            "per_var_y": {
              "terms": {
                "field": "tables.var_y",
                "size": 10000
              }
            }
          }
        }
      }
    }
  }
}
```

La consulta consta de tres agregaciones anidadas en total.

La agregación nueva es `filter`, que recibe como parámetro un filtro que ejecuta sobre los datos que recibe – en este caso las tablas (`agg_tables`, de tipo `nested`) de la lista filtrada de publicaciones (resultado de `query` en la raíz del objeto de consulta).

`filter` recibe a su vez una o más agregaciones – en este caso `per_var_y`, a las que suministra los resultados del filtro – en este caso las tabla con `var_x = M(GLUON)`, la misma condición que se utilizó en el filtro `nested` de `query`.

Este es el resultado:

```
{
  "took": 4,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
```

```
  },
  "hits": {
    "total": 1,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "agg_tables": {
      "doc_count": 2,
      "filtered_tables": {
        "doc_count": 1,
        "per_var_y": {
          "doc_count_error_upper_bound": 0,
          "sum_other_doc_count": 0,
          "buckets": [
            {
              "key": "M(SQUARK)",
              "doc_count": 1
            }
          ]
        }
      }
    }
  }
}
```


La respuesta indica que el único valor de `var_y` que tienen las tablas cuyo `var_x = M(GLUON)` es `M(SQUARK)`, el cual se puede encontrar en una sola tabla.

CAPÍTULO 11

Conclusiones y líneas de trabajo futuras

Las principales conclusiones y aportaciones de este trabajo son las siguientes:

- Se ha diseñado, implementado y desplegado un sistema de visualización de datos de publicaciones científicas sobre física de partículas.

  Este sistema se alimenta de los datos de un repositorio público – HEPData.net y permite a los físicos localizar con eficiencia trabajos publicados a partir del conocimiento de características concretas de sus datos.

  El sistema también permite obtener datos agregados de publicaciones de manera que es más fácil examinar el estado del arte en un determinado área.

  Por otro lado, el sistema abre la puerta a mayores esfuerzos de homogeneización dentro de HEPData: Curadores y revisores pueden fácilmente encontrar publicaciones similares a una recién subida para evitar asignar nombres distintos a variables iguales. De la misma forma, los curadores pueden buscar patrones diferentes para mismos datos, identificar las publicaciones en las que se usan y normalizarlas.

- Se ha construido un índice de datos de publicaciones de física de partículas basado en ElasticSearch capaz de resolver consultas complejas, así como una herramienta capaz de poblarlo, tanto en lote como de forma incremental.

- Se ha construido una interfaz de filtros recursiva que permite a un usuario componer consultas complejas de forma rápida, sin necesidad de código.

- Se ha diseñado un sistema de persistencia de estados que permite a los usuarios compartir las vistas de su aplicación con otras personas, enlazarlas en marcadores, así

como deshacer y rehacer interacciones.

La parte servidor de este sistema se ha diseñado de forma que pueda escalar a las necesidades de alta disponibilidad, rendimiento y seguridad que pueda tener en el futuro.

El desarrollo de este trabajo también ha supuesto el aprendizaje de nuevas técnicas y herramientas por parte del estudiante:

- Se han puesto en práctica los conocimientos adquiridos en el Máster en Ingeniería Informática mediante la realización de un proyecto que incluye visualización de datos, recuperación de información y arquitecturas orientadas a servicios.
- Se han puesto en práctica de forma satisfactoria un proyecto altamente multiparadigma, que mezcla técnicas de programación imperativa, funcional, reactiva, orientada a objetos, reflexiva y dirigida por flujos.
- Se han adquirido conocimientos de un dominio ajeno.
- Se ha aprendido a desarrollar aplicaciones gráficas de cierto tamaño utilizando TypeScript y Knockout.js.
- Se ha aprendido a indexar datos en el lado del servidor con ElasticSearch y en el lado del cliente con crossfilter.
- Se ha aprendido a implementar flujos asíncronos complejos de forma manejable con RxJS.

Se han identificado varias áreas en las que este trabajo de fin de máster podría ser continuado:

- La aplicación actualmente está desplegada en una infraestructura propia del autor, con fines de desarrollo y prueba. Las prestaciones técnicas de los servidores en los que actualmente se aloja, por motivos económicos, no son las óptimas.

  El paso inmediato para este proyecto es integrar HEPData Explore dentro de la infraestructura del CERN. De esta manera, la aplicación dispondría de muchos mejores recursos, de manera que sería notablemente más rápida.

- En línea con el punto anterior, otro paso sería integrar HEPData Explore dentro del sitio https://hepdata.net/.

  La interfaz de la aplicación ha sido diseñada desde el principio para que esta integración

sea posible. Concretamente, el aspecto visual es coherente con el sitio principal, la estructura visual es la misma y toda la información de estilo está definida como una extensión de la hoja de estilos del sitio padre de manera que la probabilidad de conflictos entre ambas es baja.

El indexador también ha sido construido de manera que es fácil añadir un *hook* (manejador de evento) desde otro sistema, como es la plataforma de HEPData.net, basada en Invenio[36], para que cada vez que se añada o modifique una publicación en el sitio principal, automáticamente quede indexada también en HEPData Explore.

- Actualmente en la vista de publicaciones es difícil distinguir qué puntos pertenecen a qué tablas cuando hay muchas de ellas, ya que esta asociación se hace basada en el color.

  Para facilitar esta exploración con gráficos con un número de tablas elevado, y también para usuarios con percepción del color reducida, sería de gran ayuda implementar un foco contextual, de manera que al pasar el cursor por encima de una tabla o publicación se ocultaran (o pasaran a un segundo plano) los datos del resto de tablas.

- Actualmente solo se soportan diagramas de dispersión. Si bien estos diagramas son útiles y versátiles, hay otros tipos de diagramas que también son utilizados frecuentemente por los físicos y que HEPData Explore podría soportar.

  Un ejemplo de tipo de diagrama que sería útil es el de líneas apiladas, el cual es utilizado para comparar las interacciones descritas por los modelos físicos con las mediciones hechas en los aceleradores de partículas.

  La arquitectura de gráficos basada en capas utilizada actualmente en HEPData Explore permite superponer distintos tipos de gráficos. De esta manera, se podrían crear visualizaciones como en la figura 11.1.

  Otros diagramas populares en el ámbito de la física de partículas y que podrían ser soportados en HEPData son el diagrama de contornos y el mapa de calor.

- Las búsquedas en el sistema de autocompletado utilizado actualmente se hacen con lunr. lunr funciona muy bien con datos en texto natural, como el índice de filtros. Sin embargo, muchas de las variables utilizadas en HEPData tienen un formato más parecido al código de ordenador que al texto natural, muchas llegando a incluir expresiones matemáticas codificadas en LaTeX.

  Un sistema de búsqueda específico para autocompletado de este tipo de expresiones

[36] http://invenio-software.org/

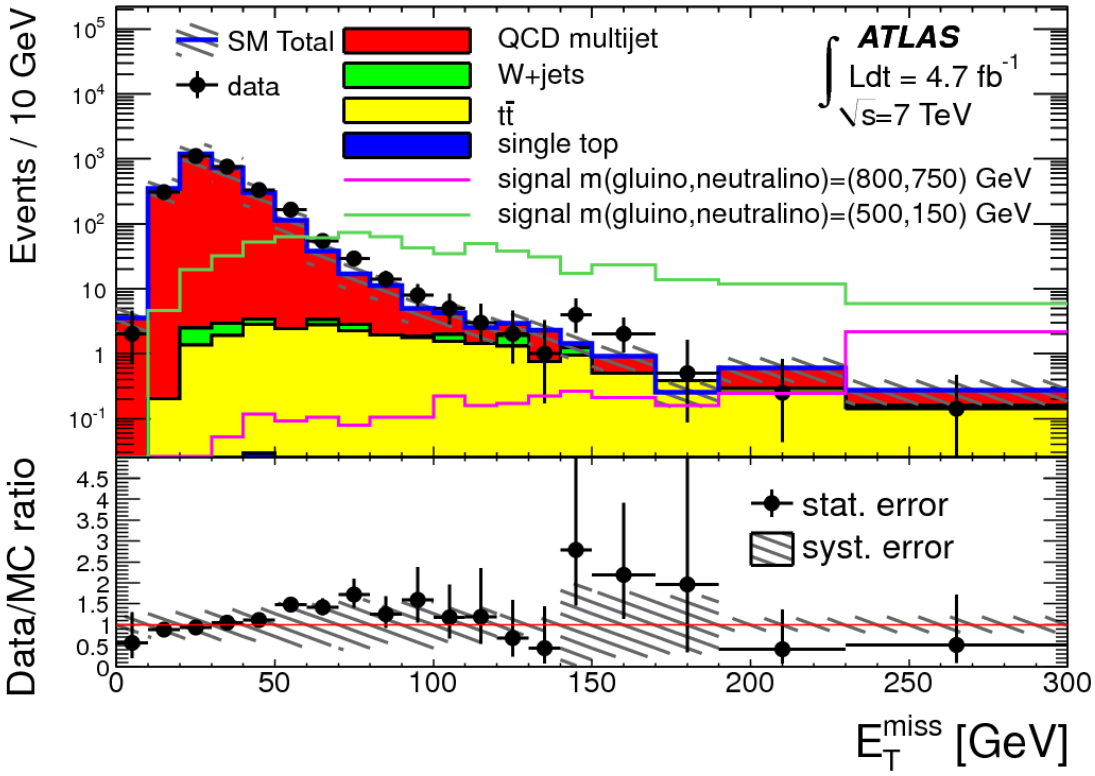

Figura 11.1: Un gráfico de líneas apiladas muestra la distribución de la energía de un tipo de interacción tal como es descrita por un modelo. Superpuesto, un diagrama de dispersión indica los valores medidos experimentalmente que, en este caso, no contradicen el modelo.

devolvería mejores resultados y podría, por ejemplo, aceptar símbolos, de manera que al buscar `M(` podrías ver todas las variables de masas.

- Sería posible añadir nuevos formatos de exportación comúnmente utilizados por los físicos, como JSON, ROOT o CSV.

- Actualmente HEPData Explore procesa más información de la que muestra. Datos como niveles de energía, reacciones, observables, etc. se almacenan en memoria y se utilizan para filtrar, pero el usuario no puede verlos sin acceder a las publicaciones originales enlazadas de HEPData.net.

  Se podrían añadir elementos de interfaz para ver información más detallada de las tablas sin salir de la aplicación.

- Un posible proyecto a futuro consistiría en incorporar un módulo de homogeneización en HEPData, de manera que el usuario pudiera marcar que dos nombres representan en realidad la misma variable, con lo podría mezclar datos de ambas en un mismo gráfico.

  Esta homogeneización podría hacerse de forma colaborativa para que el trabajo de unos usuarios ayude a otros y, en particular, pueda ser utilizado por los curadores de HEPData para homogeneizar los datos directamente en el origen.

CAPÍTULO 12

Bibliografía

12.1 Referencia técnica

- Documentación de ElasticSearch

  https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html

- Documentación de d3.js

  https://github.com/d3/d3/wiki

- Documentación de crossfilter

  https://github.com/square/crossfilter/wiki/API-Reference

- Manual de TypeScript

  https://www.typescriptlang.org/docs/handbook/basic-types.html

- Documentación de KnockoutJS

  http://knockoutjs.com/documentation/introduction.html

- Referencia de ReactiveX y RxJS

  http://reactivex.io/documentation/observable.html

- Documentación de Flask

http://flask.pocoo.org/docs/0.11/

- Documentación de SQL Alchemy

  http://docs.sqlalchemy.org/en/rel_1_0/

12.2 Libros

- RxJS - JavaScript library for functional reactive programming

  https://xgrommx.github.io/rx-book/index.html

- Reactive programming with RxJS

  https://pragprog.com/book/smreactjs/reactive-programming-with-rxjs

12.3 Otros

- Glosario de física de partículas

  http://www.interactions.org/cms/?pid=1002289

- Proceso de subida de artículos a HEPData

  https://hepdata.net/submission

CAPÍTULO 13

Glosario de términos

Para poder explicar de forma precisa este trabajo ha sido necesario utilizar abundantes términos técnicos, tanto del dominio (publicaciones de física de partículas) como conceptos específicos de la implementación informática.

Con el fin de despejar las dudas del lector sobre cualquiera de estos términos, en esta sección se presenta un glosario.

publicación Cada uno de los artículos científicos que los físicos de partículas elaboran. A menudo estos artículos tienen una naturaleza muy cuantitativa y suelen contener tablas con resultados de experimentos, o bien análisis de experimentos anteriores.

script Programa breve (a menudo, con un único archivo de código fuente) que se utiliza para una tarea muy concreta.

servidor Programa o ordenador conectado en red cuya misión es proveer funcionalidad a otros programas (clientes) que por sí solos no podrían tener. Por ejemplo, un servidor de base de datos permite a varios clientes compartir información entre ellos y almacenarla de forma segura y persistente.

alta disponibilidad Cualidad de un sistema informático de estar funcional con pocas y breves interrupciones.

Las principales dificultades a la hora de ofrecer alta disponibilidad en un sistema vienen por un lado de que las máquinas eventualmente fallan por defectos de hardware, y por otro lado de la necesidad de apagar o reiniciar los equipos para actualizaciones u otros procesos de mantenimiento.

Por estos problemas, para que un sistema pueda tener alta disponibilidad es necesario que esté replicado en varias máquinas y que exista sincronización entre ellas para reaccionar a problemas.

endpoint En una API HTTP, cada par URL y método (ej. `GET`, `POST`) que un cliente puede utilizar para realizar una acción.

objeto Entidad de programación que alberga funcionalidad y datos.

programación imperativa Estilo de programación basado en una serie de órdenes que se le dan a la máquina para que modifique su estado (memoria) hasta obtener los resultados deseados.

refactorización Modificación de la estructura del software que no afecta a su funcionalidad externa, sino que se introduce para mejorar su calidad interna o abrir camino para implementar nueva funcionalidad.

regresión Fallo de un software consecuencia de una actualización que rompe una característica que antes funcionaba.

framework Biblioteca de software que se toma como base para construir una aplicación.

Mientras que la mayoría de bibliotecas tienen una funcionalidad muy concreta que se usa en partes muy concretas de una aplicación, de un framework se espera que la aplicación lo use de forma extensiva.

Generalmente los frameworks incorporan funcionalidad habitual en paquetes reutilizables – a menudo pensados para ser extendidos por el usuario, y ofrecen soluciones a problemas que se dan a lo largo de toda una aplicación, como el acceso a base de datos, la comunicación entre módulos o la sincronización de datos.

ORM Siglas de *Object-Relational Mapper*. Se refiere a una biblioteca que ofrece una interfaz de programación basada en objetos para manipular bases de datos.

Generalmente, estas bibliotecas son capaces de abstraer las diferencias entre varios sistemas gestores de bases de datos (MySQL, PostgreSQL, Oracle...) y reducen la complejidad de la comunicación con la base de datos.

HTTP Siglas de *HyperText Transfer Protocol*. Es el protocolo utilizado por los navegadores para intercambiar código y datos. El mismo protocolo es utilizado también de forma muy habitual por otras aplicaciones para intercambio de todo tipo de datos.

DOM Siglas de *Document Object Model*. Es una serie de interfaces de programación estándares que ofrecen los navegadores web a los desarrolladores para crear aplicaciones interactivas.

Incluye funcionalidad básica como detectar un clic en un elemento, modificar el contenido de una página o llevar al usuario a una página distinta.

HTML Son las siglas de *HyperText Markup Language*. Es el lenguaje con el que se define la estructura de las páginas web. Contiene elementos esenciales como texto, imágenes, botones...

DNS Son las siglas de *Domain Name Server*. Son cada uno de los servidores que resuelven nombres como `usal.es` en direcciones IP como `82.223.76.203`. De esta manera no solo es más fácil acceder a un sitio web, sino que los administradores pueden añadir, quitar o mover sus servidores sin que los usuarios lo noten.

Zona DNS Define un conjunto de dominios con un mismo padre, administrados por una misma persona. Por ejemplo, `usal.es` es una zona DNS que contiene los subdominios `studium.usal.es` y `fciencias.usal.es` entre muchos otros.

API Son las siglas de *Interfaz de Programación de Aplicación*. Es el nombre que se le da a la parte externa de un sistema informático que le permite a otras aplicaciones informáticas comunicarse con él.

serialización Conversión de un objeto residente en la memoria de una aplicación en una representación textual o binaria del mismo susceptible de ser enviada por una red y decodificada por otro sistema informático para reconstruir el objeto inicial.

Para serializar un objeto es necesario representarlo en un lenguaje, por ejemplo, JSON. Un ejemplo de serialización es convertir una instancia de una clase `Coche` con atributos `color` y `matricula` en una cadena como `{"clase": "Coche", "color": "rojo", "matricula": 1234}`.

deserialización Proceso inverso a la deserialización: crea un objeto a partir de una representación textual o binaria del mismo.

CSV Siglas de *Comma Separated Values*. Es un formato textual de intercambio de tablas. Es soportado por muchos programas de estadística.

JSON Formato de serialización jerárquico y textual. Su simplicidad lo ha convertido en uno de los formatos más populares para intercambio de información entre aplicaciones. JSON es fácilmente legible por humanos.

Este es un ejemplo de documento JSON:

```
{
  "recid": 49090,
  "record": {
    "title": "Measurement of event shape distributions",
    "phrases": ["Exclusive", "W Dependence"]
  }
}
```

YAML Formato de serialización textual. A diferencia de JSON, YAML está diseñado para ser creado o editado por humanos.

YAML es mucho más complejo que JSON y algunas de sus funcionalidades varían de un lenguaje de programación a otro. Por este motivo la mayoría de las aplicaciones, incluido este proyecto, usan un subconjunto de YAML capaz solo de representar la misma información que JSON, de manera ambos sean intercambiables.

Este es un ejemplo de un documento YAML:

```
recid: 49090
record:
  title: Measurement of event shape distributions
  phrases: [Exclusive, W Dependence]
```

POJO Acrónimo de *Plain Old Javascript Object*, por analogía al acrónimo *Plain Old Java Object*, utilizado con fines similares en ese lenguaje.

Se consideran POJOs aquellos objetos que no contienen métodos (con la excepción de aquellos que el lenguaje proporciona de serie) y cuyas propiedades tienen valores simples: nulos, cadenas, números, arrays u otros POJOs.

El interés de los POJOs reside en que se pueden serializar y deserializar fácilmente en formato JSON o YAML de forma no ambigua y sin pérdida de información.

REST Filosofía de diseño para una API HTTP que utiliza los métodos estándar de HTTP `POST`, `GET`, `PUT` y `DELETE` para efectuar operaciones de creación, recuperación, actualización y borrado de objetos en un sistema.

En una API REST los objetos tienen URLs que los identifican y que se utilizan para operar con ellos.

Por ejemplo, una API REST podría soportar las siguientes llamadas:

- `GET /publications`. Devuelve todas las publicaciones.
- `POST /publications`. Crea una nueva publicación.
- `GET /publications/ins1234`. Obtiene una publicación.
- `PUT /publications/ins1234`. Modifica una publicación.
- `DELETE /publications/ins1234`. Elimina una publicación.

licencia Apache Licencia de software permisiva utilizada por un gran número de proyectos de software libre.

Permite la utilización, modificación y redistribución de versiones derivadas bajo otras licencias con la condición de que se reconozca la autoría del código original.

stop words Palabras pertenecientes a un idioma natural que por su frecuencia de aparición y carencia de información léxica son consideradas irrelevantes a la hora de hacer una búsqueda textual porque de otro modo desplazarían a palabras más cargadas de significado.

Ejemplos de stop words en español son "no", "si", "para", "por", "a", "en"...

Por ejemplo, si en una búsqueda no se filtraran stop words, al buscar `el clima en España`, el resultado "Análisis del **clima** de **España**" podría recibir una puntuación inferior que "**el** elefante **en el** Norte de África" a pesar de que el segundo es a todas luces irrelevante para la búsqueda.

acelerador de partículas Dispositivo que utiliza campos electromagnéticos para acelerar partículas cargadas a velocidades cercanas a la de luz en una dirección controlada.

Los aceleradores utilizados en el estudio de física de partículas son denominados también colisionadores puesto que están diseñados para provocar que varias partículas choquen. Los colisionadores cuentan con un gran número de sensores que en conjunto permiten detectar la trayectoria de las partículas subatómicas después de la colisión.

Los aceleradores de partículas varían en tamaño y características y generalmente se construyen para estudiar uno o más fenómenos muy concretos.

LHC Acrónimo de *Large Hadron Collider*, el acelerador de partículas operado por CERN en Ginebra.

colaboración Cada uno de los grupos de investigación en los que un gran número de científicos de varias universidades investigan un tema muy concreto ejecutando experimentos con aceleradores de partículas.

frases, phrases Etiquetas que describen el tema estudiado en una tabla. Fueron incorporadas a HEPData por Michael Whalley hace escasos meses. Su principal utilidad es la de identificar tablas de diferentes publicaciones que tratan un mismo tema.

Estas *frases* son una herramienta de homogeneización. El equipo de revisión de HEPData (actualmente solo Michael Whalley) las asigna a los artículos nuevos y existentes de manera que dos artículos con el mismo tema tengan una frase exactamente igual, letra a letra.

Las frases hacen mucho más fácil filtrar publicaciones de un determinado tema con esta herramienta.

GeV Unidad de energía utilizada habitualmente en física de partículas. Se utiliza indistintamente para expresar energía y masa, ya que de acuerdo a la teoría de la relatividad especial ambas son lo mismo.

error sistemático Error asumido en una experimento debido a la imprecisión del material de medida. En el contexto de física de partículas generalmente se refiere a la incertidumbre propia de los modelos físicos utilizados.

error estadístico Junto con otra variable, el *valor esperado*, representa el posible rango de valores de una variable donde se considera probable que esté el valor real de la misma. Dependiendo del número de muestras y la varianza de las mismas el error estadístico será mayor o menor.

SQRT(s), cmenergies Variable que representa el nivel de energía de una colisión, en GeV. Recibe ese nombre por la variable s de Mandelstam, cuya raíz cuadrada coincide con la suma del momento lineal de las partículas que participan en la colisión.

También recibe el nombre *energía del centro de masas*, codificado de forma abreviada como `cmenergies`.

INSPIRE Repositorio de artículos científicos sobre física de partículas. Antes de entrar en HEPData, todos los artículos entran en este otro repositorio.

DOI Siglas de *Digital Object Identifier*, un código único para cada artículo u objeto de interés científico. DOI sigue un modelo federado, donde una asociación internacional asigna prefijos a editoriales y curadores de contenido (entre ellos HEPData) y ellos añaden

sufijos únicos para cada uno de sus recursos.

Por ejemplo, HEPData.net tiene el prefijo `10.17182/`. A partir de ese prefijo HEPData.net genera un DOI como `10.17182/hepdata.71624.v1/t33` que enlaza a la tabla 33 de la versión 1 de la publicación con id 71624.

Los DOIs pueden ser codificados como una URL y resueltos a través de un servicio web, por ejemplo: http://doi.org/10.17182/hepdata.71624.v1/t33

LaTeX Lenguaje utilizado para producir documentos, muy popular en el ámbito científico. LaTeX es capaz de representar fórmulas matemáticas complejas de forma precisa y legible.

singleton En programación orientada a objetos, única instancia de una clase, o método para conseguir dicha instancia.