Cómo crear un sistema gestor de biblioteca usando tecnologías públicas de Google*

La Biblioteca del EPBCN

Josep Maria Blasco

Espacio Psicoanalítico de Barcelona
Balmes, 32, 2º 1ª – 08007 Barcelona
josep.maria.blasco@epbcn.com
+34 93 454 89 78

21 de octubre de 2014

1 Introducción

¿Por qué algunos buscadores funcionan tan bien y otros tan mal? Google, por ejemplo, funciona estupendamente; en cambio, en la mayoría de las bibliotecas de las universidades españolas, encontrar determinados libros puede convertirse rápidamente en un suplicio.[1] Lo mismo sucede si uno intenta encontrar un libro en determinados grandes almacenes o en ciertas librerias importantes, por poner sólo algunos otros ejemplos.

Quizás hacer un buen buscador es algo muy *difícil*, podría pensarse. Entonces, siguiendo este argumento, los únicos que lo sabrían hacer serían los de Google, que tienen en nómina a los mejores informáticos del planeta (o

*URL de este documento: `https://www.epbcn.com/pdf/josep-maria-blasco/2014-09-20-La-biblioteca-del-EPBCN.pdf`.

[1] Nos referimos, por supuesto, a *los programas web*, no al acto de la retirada física de un volumen, en el que suele contarse con la asistencia de personal especializado.

los de Microsoft, que les siguen de cerca, con su buscador Bing, etcétera).

O a lo mejor no es tan difícil, al menos conceptualmente, pero es *caro*: quizá implementar un buen buscador requiere de una inversión que no todos pueden permitirse. Las universidades, es sabido, siempre andan escasas de dinero y muchas librerías están a punto de quebrar.

Otra posibilidad es que no sea ni difícil ni caro, sino que dependa de que estemos usando *herramientas modernas*: por ejemplo, Google, que parece que internamente usa, entre otros, el lenguaje de programación Java, disfrutaría de una serie de aplicaciones y posibilidades que otra empresa, quizá ligada a sistemas más antiguos (por ejemplo, aplicaciones en lenguaje PL/I, o *mainframes*[2]), sólo podría usar con más dificultad.

O quizá, por último, para implementar un buen buscador haya que contar con *personal super-especializado*: quizá el problema de la búsqueda involucre muchos conocimientos de diversa índole y sea difícil encontrar personas que los reúnan todos.

De hecho, existen una serie de sistemas, tanto gratuitos como de pago, de código abierto o cerrado, que implementan de variados modos soluciones al problema de la búsqueda.[3] Supondremos que, por cualquier razón (dificultad técnica de integración, póliticas sobre el código abierto, etcétera) no resulta aceptable integrar un programa externo y nos vemos obligados a desarrollar una aplicación nosotros mismos. ¿Será muy dificil hacerlo, como sugieren los argumentos anteriores?

En este artículo los desmontamos todos. Usamos una aplicación web, que el autor ha tenido que desarrollar, como ejemplo de que es posible escribir en poco tiempo un buscador de alta calidad contando con muy pocos recursos, tanto humanos como económicos, en un lenguaje de programación poco menos que obsoleto, sin ser especialista en nada más que en programación,[4] y sin

[2]Los grandes ordenadores, que a veces ocupan una planta entera, usados, entre otros, por la banca.

[3]Como Apache Lucene, `http://lucene.apache.org/core/`, de la Fundación Apache, uno de los más completos y conocidos. Se puede encontrar una lista de software en `http://en.wikipedia.org/wiki/Full_text_search`.

[4]Adelantando algunos términos que se introducirán después, no soy especialista en Unicode, ni en algoritmos de relevancia, ni en el problema general, muy complejo, de la búsqueda. Y eso es precisamente lo interesante: un buscador que funcione *lo puede escribir cualquiera*. ¿Por qué es tan raro encontrarlos, entonces?

tener que integrar un producto externo. Casi todas las técnicas que pueden parecer «especializadas» y se han utilizado en la confección del programa se pueden encontrar libremente en la web, en la mayoría de los casos muy bien explicadas y en fuentes muy variadas y de muy diversos niveles. Lo que resta es una colección de ideas bastante intuitivas y el hilo que lo cose y junta todo.

La aplicación web en cuestión es la que gestiona la Biblioteca del Espacio Psicoanalítico de Barcelona,[5] pero los mecanismos, conocimientos y soluciones presentados pueden generalizarse sin dificultad alguna a otros tipos de aplicación y a otros contextos. Con la intención de hacer fácil la lectura del texto a un público no especializado, nos hemos esforzado en presentar los temas del modo que nos ha parecido más simple; no hay que «saber informática», sólo tener curiosidad y, en algunos casos, un manejo «medio», a nivel de usuario, de un navegador y de las aplicaciones gratuitas más habituales. Por otra parte, no hemos querido renunciar a presentar también los detalles más técnicos y las descripciones más rigurosas o avanzadas, que estarán destinadas a un público con formación especializada y, por tanto, forzosamente más restringido; éstas están siempre circunscritas a las notas al pie, las notas en línea (que se presentan en una tipografía más pequeña) y los apéndices finales.

> Esto, por ejemplo, es una nota en línea. Es un artificio escritural que se usa poco en este tipo de textos y que, sin embargo, es muy útil. Pone aparte lo que podría intimidar o rompe la línea argumentativa; señala los puntos que se tratan con más detalle o profundidad, e invita a considerarlos candidatos a una lectura posterior, de ánimo más profundo o quizá más técnico; finalmente, permite incluir grandes cantidades de texto, que en una nota al pie afearían su composición, dándole un aspecto pesado que podría llevar al desánimo («¡uf!, ¡si hay más notas que texto!»).

Nos centraremos a partir de aquí en nuestra Biblioteca. Esto nos alejará durante un rato del problema de la búsqueda, pero volveremos a encontrarnos con él en seguida.

[5] Cuya historia se describe con cierto detalle en el apéndice A en la página 27. Se encontrará también un breve listado describiendo sus especificaciones técnicas en el apéndice B en la página 29.

2 Descripción del problema

Gestionar una biblioteca de un cierto tamaño es mucho más complicado de lo que puede parecer de entrada: a la complicación que supone gestionar las fichas bibliográficas[6] se une el problema logístico del almacenamiento, recuperación y préstamo de los libros. Tradicionalmente se utilizaban, para el almacenamiento de datos bibliográficos, las *fichas catalográficas*, generalmente de cartulina, que disponían de un espacio limitado, lo que generaba la costumbre de abreviar y limitar los datos recogidos. Para el almacenamiento se utilizaban métodos de codificación más o menos ingeniosos y ahora completamente obsoletos. La aparición de la informática posibilitó, al menos en principio, la creación de sistemas de información bibliográficos mucho más ambiciosos, pero las inercias institucionales y de poder y el alto coste material y humano de migración han hecho que, en la práctica, muchos sistemas de los que están actualmente en uso no superen y en algunos casos sean inferiores a las antiguas fichas catalográficas.

2.1 ¿Qué información deberia recoger una ficha catalográfica ideal?

La gestión informatizada del manejo de ejemplares físicos no presenta ningún problema: basta con que el programa informático refleje las agrupaciones físicas (e.g.: estanterías) en las que se encuentran los libros y con escribir un conjunto de operaciones primitivas útiles para la gestión de esas agrupaciones. En claro contraste con esta simplicidad, la informatización «correcta» o

[6]Las editoriales no se ponen de acuerdo en cómo presentar la información relativa a las ediciones de los libros y, muchas veces, esa informaciòn es incompleta o simplemente descuidada y chapucera, hasta el punto existe una carrera universitaria, *Biblioteconomía y documentación*, en la que se estudia en profundidad el tema.

«ideal» de los datos bibliográficos es altamente problemática.

Veamos algunos ejemplos: un libro, es muy claro, tiene un *título*, uno o más *autores*, está publicado por una *editorial* en determinada *ciudad* y *fecha* y se presenta mediante un determinado *número* de edición. La cosa se complica enseguida, especialmente con los números de edición, las reimpresiones, y las reimpresiones en otra colección bajo licencia: por ejemplo, editoriales como RBA o Círculo de Lectores tienen tendencia a realizar ediciones especiales basadas en otras ediciones de editoriales «mas serias»; ¿qué información importa, en este caso?

Habría que ver, también, quién es el *traductor*; esto es, en muchos casos, importante y, en algunos, decisivo. O el *compilador*, cuando se trata de una colección. O el *prologuista*, y así sucesivamente. O sea que, en un libro, pueden intervenir varias *personas*, en *funciones* muy diferentes.

Después está el problema de las obras multi-volumen (por ejemplo, el *Cuarteto de Alejandría* de Lawrence Durrell) y el de las *colecciones* (por ejemplo, *El libro de bolsillo* de Alianza Editorial). Hay casos en los que obras multi-volumen aparecen como diferentes números de una colección (por ejemplo, la edición en *El libro de bolsillo* de *En busca del tiempo perdido* de Marcel Proust), lo que añade complejidad al problema.

La complejidad crece aun más si queremos almacenar también información sobre los componentes de recopilaciones, como las Obras Completas de Sigmund Freud: sería muy interesante tener una ficha de, por ejemplo, *El porvenir de una ilusión*, pero eso es una *parte* de un volumen de una colección.

La información de traducción es también muy importante: no sólo quién ha sido el traductor, sino de qué versión se ha partido para realizar la traducción; un sistema informatizado ideal debería darnos esa información (especialmente cuando en la biblioteca se dispone también de la versión en lengua original).

Resumiendo: una ficha ideal debería poder recoger información sobre *El porvenir de una ilusión*, decirnos que fue escrito en alemán con el título original del *Die Zukunft einer Illusion* por Sigmund Freud y publicado la Editorial de la Internacional Psicoanalítica en Leipzig, Viena y Zurich en 1927; que apareció en las diferentes versiones de las *Obras Completas* de Freud en alemán e inglés en tales tomos y páginas; que fue traducido directamente del alemán por (el equipo de) José L. Etcheverry, ocupando las

páginas 1 a 56 del tomo XXI de las *Obras Completas* publicadas en primera edición por Amorrortu en Buenos Aires en 1979; que se dispone de una nota introductoria de James Strachey, traducida de las obras completas inglesas; que la segunda edición es de 1986 y que nosotros estamos manejando la séptima reimpresión de esa segunda edición, que es de 2001; cual es el ISBN; etc.

2.2 ¿Qué tipo de ergonomía debería tener un programa ideal?

Además de todo lo anterior, un programa ideal debería permitirnos *navegar* directamente haciendo click sobre cualquier ítem significativo. Siguiendo con nuestro último ejemplo, deberíamos poder pulsar sobre «Amorrortu» y ver una ficha de la editorial, con una lista de todo lo publicado por esa editorial; pulsar sobre «Sigmund Freud» y ver una ficha del autor y la lista de todo lo publicado por el autor; pulsar sobre «Viena» y ver todo lo publicado en Viena; etcétera.

Del mismo modo, deberíamos disponer de un *buscador* que nos permitiese encontrar *El porvenir de una ilusión* tecleando tanto `porvenir` como `Freud 1927`, tanto `Obras Completas Amorrortu XXI` como `Freud ilusión obras completas`, tanto `1458` (suponiendo que el ejemplar físico tenga el número de catálogo 1458) como `Freud est4` (suponiendo que el libro se encuentre en la estantería denominada `est4`).

3 Objetivos de esta versión y decisiones de diseño

Establecidas las prestaciones ideales de nuestro programa, que guiarán las elecciones de diseño a nivel informático, debemos también tener en cuenta las consideraciones prácticas: disponemos un único programador (quien esto escribe) y de poco tiempo para programar; revisar los datos de los libros da también muchísimo trabajo; habrá que aceptar algunas limitaciones con respecto a nuestros ideales y sacar adelante una versión *posible*, dejando el irse acercando, si es el caso, a la *ideal* para más adelante.

Para esta versión nos propondremos, entonces:

1. Recopilar la mínima información imprescindible para poder elaborar una ficha bibliográfica *digna* de cada uno de los ejemplares de la Biblioteca. Definiremos «digna» como sigue: para cada ejemplar, se almacena 1) el *título* del libro (estructurado del siguiente modo: un *pretítulo* opcional, el *título* propiamente dicho y un *posttítulo* opcional); 2) cero o más *personas* que intervienen en el libro (en esta versión, las personas siempre intervienen como *autores*: dejamos la inclusión de los traductores, compiladores, prologuistas, etc., para más adelante); 3) una *editorial*; 4) una *localidad* de edición; 5) un *año* de edición; 6) un *número* de edición; opcionalmente 7) una *colección*, en cuyo caso se podrá incluir también un identificador único en la colección (puede ser un número, pero también elementos como «Tomo II», «Livre XXI» o «AH 0001»); 8) un número de ejemplar; 9) una denominación de estantería, con su correspondiente posición ordinal.

2. Incorporar los máximos elementos de *navegabilidad* descritos en la sección anterior.

3. Desarrollar un *buscador* lo más agil, sencillo e intuitivo posible.

La recopilación de información es un problema de recursos humanos. En cuanto a la implementación de una *navegabilidad* completa, se trata de algo

trivial a nivel informático, que no precisa de una ulterior discusión ni descripción.

Sin embargo, hacer un buen *buscador* es mucho más complicado de lo que parece, hasta el punto de que un producto tan extendido, maduro y complejo como Wordpress[7] no puede: 1) buscar por relevancia; 2) buscar en los comentarios; 3) restringir las búsquedas por categoría; 4) resaltar las palabras buscadas en los resultados de la búsqueda; 5) manejar con propiedad los caracteres acentuados. Dedicaremos la práctica totalidad de lo que sigue a describir cómo hemos implementado nuestro buscador, que no padece de ninguna de estas limitaciones.

4 ¿Cómo buscar cómodamente?

Google marcó hace mucho tiempo la tendencia: un buscador *no tiene que notarse*, tiene que ser una interfaz virtualmente *transparente*. ¿Cuándo notamos la interfaz? Cuando 1) es demasiado pesada gráficamente; 2) cuando requiere memorizar trucos específicos para determinados casos de búsqueda; y, especialmente 3) cuando *no funciona bien*.

4.1 Diacríticos

Y que no funcione bien suele ser frecuente: aún restringiéndonos a las diversas variaciones del alfabeto romano, cada vez es más frecuente disponer

[7]Nos referimos a la versión .org y la información es de 2013. La versión 4.0 de Wordpress soluciona los problemas con los caracteres acentuados (y de este modo garantiza que un blog exportado y reimportado pierda todos los enlaces permanentes que contuviesen caracteres que la versión anterior no sabía manejar).

de libros cuyos títulos, autores, etcétera contienen signos diacríticos «inusuales». Por ejemplo, podemos tener un libro cuyo autor sea Jan Łukasiewicz. ¿Cómo introducimos la letra «Ł», si no está en el teclado? Si usamos Windows, podemos abrir el Charmap y buscarlo, o usar algún otro truco, pero eso es pesado y poco ágil. Todavía peor, puede ser que nos haya mandado la información sobre el libro, por ejemplo por correo electrónico, alguien que no sepa cómo escribir la «Ł» y la haya substituído por una simple «L». El buscador tiene que ser capaz de encontrar libros de (o sobre) Łukasiewicz aunque escribamos `Lukasiewicz`, textos sobre el Barça aunque escribamos `Barca`, ocurrencias de jamón aunque escribamos `jamon`, etcétera. Y, lo que es más importante, *tiene que saber hacer esto para cualquier combinación de diacríticos que sea posible esperar encontrar*, no sólo las que conocemos sino otras que quizá no conozcamos, como la «Ď» o la «Ý» checas, o la «ğ» o la «ş» turcas.

4.2 Mayúsculas, minúsculas y puntuación

Tampoco es cómodo que el buscador sea muy quisquilloso con la distinción entre mayúsculas y minúsculas; en la inmensa mayoría de los casos, obtendremos un mejor resultado si se nos muestra todo lo que coincida con lo que hemos escrito, excepto el hecho de que esté o no en mayúsculas. Es decir, debemos poder encontrar «Barcelona» escribiendo, por ejemplo, `barcelona`; es mucho más cómodo. Además, en algunos casos puede ser que no recordemos bien qué letras son o no mayúsculas, por ejemplo en «John DeSantis», que es imposible en castellano y nos parece poco natural.

Algo parecido ocurre con los signos de puntuación: muchas veces se busca usando operaciones de cortar y pegar, de modo que la puntuación no es lo que estamos buscando y nos conviene eliminarla antes de efectuar la búsqueda.

4.3 Te encontraré estés donde estés

A veces la información que se busca es confusa: «un libro de Freud..., o sobre Freud, quizá..., recuerdo que era de Alianza y estaba *aquí*». Si «aquí» es la estantería «E33», deberíamos poder buscar ese libro escribiendo `freud alianza e33` o, dicho en términos más generales, deberíamos ser capaces de buscar un libro por cualquier criterio (título, autor, ciudad, etc) sin tener que explicitar cuál es el criterio (lo que es farragoso, confuso y atenta contra el principio de la invisibilidad de la interfaz). Más adelante[8] nos encontraremos con un muy agradable efecto lateral de encontrar una solución a este problema.

4.4 Voy a tener suerte

Si nuestros términos de búsqueda son demasiado generales, un buscador estándar no nos resultará muy útil. Por ejemplo, si buscamos `Freud` y nuestro buscador ordena los resultados alfabéticamente, obtendremos un listado de obras de Sigmund y Anna Freud, colecciones de Obras completas de esos autores, personas apellidadas Freud..., que quizá no nos servirá de gran cosa. Sería deseable que si ponemos `Freud` obtengamos, como primer resultado de la búsqueda, la ficha del autor Sigmund Freud; que si escribimos `Alianza`, encontremos al principio la ficha de Alianza Editorial; que si tecleamos `London`, obtengamos la ficha de la ciudad de Londres, etcétera. Es decir, sería muy deseable que nuestro buscador funcionase como Google, que ordena sus resultados, entre otras cosas, por *relevancia*.

Vamos a ver ahora cómo podemos aproximarnos a esta búsqueda «cómoda», «no intrusiva», «transparente», que «funciona bien».

[8] En la sección *Ejemplos: los beneficios secundarios de nuestro modo de indexar* en la página 23.

5 Buscar por cualquier criterio indexándolo todo

Para empezar abordaremos el problema de la interfaz, en el sentido de dar buenas respuestas a búsquedas difusas. Queremos encontrar las cosas sin que importe si nos estamos refiriendo al título, el pretítulo, a la editorial, a la colección, etcétera. ¿Cómo podemos conseguir esto? Para decirlo en una sola frase, que inmediatamente desplegaremos: *indexando, para cada elemento, toda la información relacionada.*

5.1 Elementos y relaciones

Para explicar qué queremos decir con *indexarlo todo* vamos a tener que entrar en la estructura interna de la Biblioteca. Toda la información que almacenamos está en una *base de datos*;[9] para los no iniciados, algo parecido a un par de hojas de cálculo muy grandes y de acceso rapidísimo llamadas *tablas*. La información que contiene la Biblioteca es, en última instancia, información sobre los *elementos* que componen la Biblioteca, es decir, información sobre los libros, ejemplares, colecciones, autores, ciudades, etcétera y las *relaciones* entre esos elementos, que estructuran esa información. Nuestra implementación[10] se compone pues de únicamente dos tablas:

- La tabla de los *elementos*. Cada elemento se compone a su vez de una serie de atributos (denominados técnicamente *columnas*), entre otros y para lo que nos interesa ahora: un *identificador único* interno (el equivalente al número de fila en una hoja de cálculo), que es un entero

[9] En nuestro caso, estamos usando la última versión de SQLite; consúltese el apéndice B en la página 29.

[10] Hacemos referencia aquí a ella para ilustrar claramente qué tipo de índice estamos proponiendo. Es claro que no es la única posible, y no se nos ocurre plantearla como modelo. Sólo nos interesa en cuanto nos permite articular lo que queremos transmitir.

positivo, la llamada *clave primaria* de la tabla; un indicador del *tipo* de elemento (esto es, si se trata de un autor, un libro, una ciudad, etcétera); varios atributos que describen de forma legible el elemento (para las personas, nombre, dos apellidos y, opcionalmente, un pseudónimo; para los libros, el titulo, opcionalmente el pretítulo, el posttítulo y quizá un número de colección: etcétera). Podemos figurarnos la tabla así:

...	...	...	...
101	Autor	Junichirō	Tanizaki
102	Autor	Irwin David	Yalom
...	...	...	...
242	Libro	El día que Nietszche lloró	
243	Libro	El elogio de la sombra	
...	...	...	...
355	Editorial	Siruela	
...	...	...	...
369	Editorial	Destino	
...	...	...	...

- La tabla de las *relaciones*. Cada fila es una relación que está formada, fundamentalmente, por un par ordenado de números (n_1, n_2), que tienen que ser claves de la tabla de elementos, y un indicador de tipo de relación R. Lo que se expresa es que el elemento con clave n_1 está en relación R con el elemento con clave n_2 o, dicho de manera quizás más clara, pero también más apretada, que *el* R *de* n_1 *es* n_2.

  Veamos algunos ejemplos (añadiremos también un identificador de fila para podernos referir a ellas):

...	...	...	...
18	Autor	242	102
19	Autor	243	101
...	...	...	...
54	Editorial	243	355
56	Editorial	242	369
...	...	...	...

  La fila 18 *expresa* que el autor de 242 (es decir, mirando la tabla anterior: del libro *El día que Nietszche lloró*) es 102 (es decir, Irwin D.

Yalom); la fila 19, que el autor de *El elogio de la sombra* es Junichirō Tanizaki; la 54, que la editorial del último libro es Siruela; y la 55, que la del primero es Destino.

De este modo podemos manejar información complejisima con sólo dos tablas muy sencillas (las tablas reales son un poco más complejas, pero no mucho; describiremos algunos de los atributos o columnas que faltan a medida que lo vayamos necesitando).

5.2 Construyendo nuestro índice

Ahora ya estamos en condiciones de explicar qué queremos decir con *indexarlo todo*:

- En una aproximación *clásica*, permitiríamos que el buscador encuentre las cosas buscando únicamente en los atributos legibles. Así, escribiendo `Nietszche` encontraríamos la fila 242 de la tabla de elementos, tecleando `Siruela` encontraríamos la 355, etcétera. Claramente, esto no es lo que queremos.
- *Nuestra aproximación* consiste en añadir a cada fila de la tabla de elementos un atributo que contenga *toda la información relevante relacionada con ese elemento*.

Por ejemplo, ya sabemos (gracias a la tabla de relaciones) cuáles son el autor y la editorial de *El elogio de la sombra*; puesto que disponemos de esa información, nada nos impide añadir a la tabla de elementos una columna nueva, que llamaremos *columna del índice*, dedicada íntegramente a la búsqueda, con el siguiente contenido (por razones de espacio, mostramos sólo la clave y la nueva columna):

...	...
243	El elogio de la sombra Junichirō Tanizaki Siruela
...	...

Es decir, almacenamos *todo* (todo lo relevante): no sólo el nombre del libro, sino también su autor, la editorial, etcétera (estos ejemplos son, claro está, simplificaciones: en realidad, almacenamos mucha información más,

pero esto no importa ahora). De este modo, usando nuestra nueva columna para las búsquedas, podemos encontrar el libro buscando `sombra`, pero también `Tanizaki` o `Siruela`, que era lo que queríamos conseguir. Hemos resuelto así el tercero de nuestros problemas; nos quedan tres más.

> Para ser precisos, lo que almacenamos, más allá de las modificaciones que introduciremos en la próxima sección, es el «todo» al que nos referimos, agregándole antes y después un espacio en blanco. Esto nos permite buscar en SQL con una cláusula LIKE del siguiente modo: si el usuario ha buscado las palabras P_1, P_2, ..., P_n, pedimos que el campo de indexación, pongamos *IX*, sea `LIKE ' Pi '` para cada i, es decir,
>
> ```
> IX LIKE ' P1 ' AND IX LIKE ' P2 ' AND ... IX LIKE ' Pn '.
> ```
>
> Si la base de datos crece mucho, o por cualquier otra razón estas búsquedas se hacen ineficientes, puede recurrirse a sistemas de búsqueda *full text*, como los FTS3 o FTS4 de SQLite o las columnas *full text* de MySQL, sin que esto altere los argumentos que presentamos.
>
> Un sistema de indexación así tiene muchas ventajas, pero también algunos inconvenientes: algunos cambios sencillos en un solo elemento pueden implicar cambios en el índice para muchos elementos. Por ejemplo, si cambiásemos la descripción «Alianza» por «Alianza Editorial», tendríamos que reindexar todos los libros publicados por esa editorial. Esto puede hacerse usando un proceso en modo *batch* para no entorpecer el tiempo de respuesta de la interfaz ni del servidor.

6 Indexar en mínúsculas, sin puntuación ni acentos

Para abordar el problema de los diacríticos y a la vez el de las distinciones entre mayúsculas y minúsculas y el de la puntuación, vamos a hacer lo siguiente: modificaremos la columna de índice descrita en el apartado anterior, de modo que lo que se almacene esté en minúsculas y carezca de puntuación y acentos. Es decir, siguiendo el último ejemplo de la sección anterior, en vez de almacenar

```
El elogio de la sombra Junichirō Tanizaki Siruela
```

lo que guardaremos será

```
el elogio de la sombra junichiro tanizaki siruela,
```

y en vez de

```
«¡Papá!», dijo ella Autor Inventado DeLaMurga
Ediciones
```

almacenaríamos

```
papa dijo ella autor inventado delamurga ediciones.
```

Para poder hacer esto precisaremos de:

- una función que quite los diacríticos de los caracteres, es decir, transforme «á» en «a», «ō» en «o», «ç» en «c», «ñ» en «n», etcétera;
- una función que determine qué son signos de puntuación y qué no lo son, para poder eliminarlos; y
- una función que transforme mayúsculas en minúsculas.

Ahora bien; cuando nos referimos, por ejemplo, a «quitar los diacríticos de los caracteres», ¿qué queremos decir, exactamente, con «los caracteres»? ¿A qué conjunto de caracteres nos estamos refiriendo? Si queremos hacer las cosas lo mejor posible, tendríamos que estar refiriéndonos a un conjunto máximamente amplio, especialmente si tenemos en cuenta que cada vez es más frecuente disponer de libros de autores extranjeros, cuyos nombres incluyen caracteres como «ø» (Søren Kierkegaard), «ō» (Junichirō Tanizaki) o «Ł» (Jan Łukasiewicz).

6.1 Unicode

Por suerte, la nuestra es una aplicación web, y los estándares actuales para la web[11] incorporan soporte para Unicode, una especie de super-alfabeto universal que contiene o, mejor dicho, aspira a contener todos los caracteres existentes.

Más precisamente, Unicode es un estándar[12] que tiene el objetivo de incorporar los caracteres correspondientes a los sistemas de escritura de todas las lenguas conocidas por el hombre, sean estas vivas o muertas, usen sistemas alfabéticos o ideográficos, sean mayoritarias o muy minoritarias, se usen para la expresión o para el intercambio simbólico (en particular, las matemáticas). Lo hace asignando a cada carácter su correspondiente *código* numérico («code point», un número natural), único y distinto para cada carácter. Incorpora el alfabeto romano en todas sus variantes, pero también el griego, el cirílico y el hebreo; los alfabetos descendientes del sánscrito, como el devanagari, propio del hindi, idioma oficial de India, pero también el gujarati, y el malayalam, hablado en el estado indio de Kerala; los alfabetos ideográficos, como los kanjis que configuran la escritura japonesa, junto con los silabarios hiragana y katakana; los jeroglíficos egipcios; las diversas simbologías técnico-científicas; los símbolos matemáticos; y un larguísimo etcétera. Es un estándar en constante evolución y crecimiento, y en su encarnación actual, la versión 7.0.0, codifica más de 110.000 caracteres.

A	0	Đ	
65	48	272	33258

Algunos lenguajes de programación incorporan soporte para Unicode; en varios casos, ese soporte es parcial o incompleto. Otros lenguajes, como el que estamos usando para nuestra aplicación de Biblioteca, no disponen de ninguna implementación de soporte para Unicode. Tendremos que desarrollar ese soporte nosotros, lo que en última instancia está muy bien: aprenderemos más.

[11] Como HTML5.

[12] `http://www.unicode.org/standard/standard.html`.

6.2 La *Unicode Character Database*

El estándar Unicode no se limita a codificar los caracteres existentes, lo que tendría una utilidad más bien limitada, sino que incluye también una base de datos, la llamada *Unicode Character Database*[13] (a partir de aquí: UCD), que define una larga serie de *propiedades* para cada carácter. Mediante el estudio de la UCD,[14] que a su vez está descrita con todo detalle en una serie de documentos bastante farragosos que también pertenecen al estándar, es posible construirse uno mismo, como hemos hecho nosotros para nuestra aplicación, los programas necesarios para soportar Unicode. Describiremos en líneas generales lo que hemos hecho.

La UCD se implementa a partir de una serie de ficheros; hay una versión XML, pero nosotros trabajaremos con la versión de texto plano. El primer fichero que nos va a interesar se denomina `UnicodeData.txt`. Está compuesto por líneas como la siguiente (que se muestra en dos líneas por cuestiones de espacio)

```
00C0;LATIN CAPITAL LETTER A WITH GRAVE;Lu;0;L;0041
  0300;;;;N;LATIN CAPITAL LETTER A GRAVE;;;00E0;
```

que contienen información separada por el carácter «;», siempre en el mismo formato. En particular:

- La *primera* posición contiene el *código* del carácter en formato hexadecimal, en este caso `00C0`, equivalente al decimal 192.
- La *segunda* posición contiene una *descripción* legible (en inglés) del carácter (en este caso, se podría traducir por «letra mayúscula latina A con acento grave»).
- La *tercera* posición nos informa del *tipo* de carácter. En este caso, «Lu» nos indica que se trata de una letra («L» del inglés «letter») mayúscula («u» del inglés «uppercase»).

[13] Consultable en `http://www.unicode.org/ucd/`.

[14] Descrita fundamentalmente en `http://www.unicode.org/reports/tr44/`, aunque hay muchos otros informes que proporcionan información adicional.

- La *sexta* posición nos da, si existe, una *descomposición* del carácter en otro(s) carácter(es) con menos diacríticos y un diacrítico. En este caso, nos dice que la letra «À» (`00C0`) se descompone en la letra «A» (`0041`) y el diacrítico para el acento grave (`0300`).

- La *decimotercera* posición contiene, si existe, el equivalente del carácter *en mayúscula*. En este caso la información es vacía, puesto que el carácter de partida ya está en mayúscula.

- La *decimocuarta* posición contiene, si existe, el equivalente del carácter *en minúscula*. En este caso, la correspondiente minúscula de nuestro carácter «À» (`00C0`) es «à» (`00E0`).

  > Hay que tener en cuenta que las descomposiciones que proporciona la UCD no son completas, en el sentido de que el o los caracteres en que se descompone un carácter determinado pueden ser a su vez susceptibles de descomposición; por ejemplo, cuando un carácter tiene más de un diacrítico, la descomposición suele referir a otro carácter con un diacrítico menos, que a su vez puede ser descompuesto, y así sucesivamente. Esto quiere decir que si queremos llegar, por así decir, al «fondo de la cuestión», es decir, averiguar cuál es el carácter no acentuado que corresponde a nuestro carácter de partida, quizá debamos recurrir a la UCD varias veces o, si lo queremos expresar de un modo más técnico, que la descomposición es *recursiva*.

El *tipo* del carácter (en la tercera posición) nos informa también de si el carácter es de puntuación o no. Por tanto, parecería que, mediante el estudio de `UnicodeData.txt`, podemos extraer e implementar fácilmente un sistema que determine qué carácteres son puntuaciones, cuál es el correspondiente carácter en minúscula, si existe, y cuál es el correspondiente carácter sin acentos, si existe.

6.3 Filtrando los resultados

El problema es que, tomado en su totalidad, el sistema Unicode puede resultar *demasiado grande* (recordemos que estamos hablando de más de $110,000$ caracteres), lo que puede hacerlo ineficiente en términos de espacio y tiempo. Probablemente no necesitamos soporte para el japonés, ni para los

jeroglíficos egipcios, ni para el malayalam.

La UCD incluye dos ficheros, `Scripts.txt` y `ScripExtensions.txt`, que nos facilitarán trabajar sólo con la parte de Unicode que nos interesa. El primero, `Scripts.txt`, asigna a cada carácter del estándar un *script*. «Script» es un término inglés que se aplica tanto a los alfabetos (v. gr. el romano o el cirílico, pero también los alfabetos finitos matemáticos) como a los silabarios (v. gr. los hiragana y katakana japoneses) y los sistemas logográficos (v. gr. los ideogramas chinos). Asigna a cada carácter C una propiedad *Script*(C); centrándonos en los caracteres cuyo *script* es `Latin` (el alfabeto romano), `Common` o `Inherited` (volveremos en seguida sobre estos dos valores) podremos eliminar los caracteres de los alfabetos y sistemas de escritura que no nos interesan y reducir el número de caracteres que estamos considerando.

En cuanto a `ScriptExtensions.txt`, se trata de un fichero que aclara a qué sistemas de escritura corresponde un carácter cuando este tiene la propiedad *script* de `Common` o `Inherited`. En este caso, a cada carácter le corresponde una serie de sistemas de escritura; si entre estos sistemas está `Latn`, que corresponde al alfabeto romano, se trata de un carácter que nos interesa tratar; en otro caso, podemos ignorarlo.

Estos dos ficheros nos permiten, entonces, *filtrar* `UnicodeData.txt`, pero, al hacerlo, observamos que todavía tenemos que vérnoslas con una cantidad muy grande de caracteres. Aquí vendrá en nuestra ayuda otro fichero de la UCD, `Blocks.txt`, que clasifica los caracteres Unicode en *bloques*, destinados a diversos usos. Por ejemplo, los caracteres del bloque *IPA Extensions*, a pesar de contar con la propiedad de incluir el *script* romano, no los vamos a encontrar si manejamos libros, ya que son los caracteres del Alfabeto Fonético Internacional (en inglés, las iniciales son IPA), que se usan por ejemplo en los diccionarios para definir cómo deben pronunciarse las palabras, pero no aparecen nunca en los elementos de un libro. Si estudiamos `Blocks.txt`, podremos saber qué partes del alfabeto romano nos interesa tratar y cuáles no.

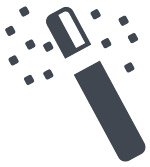
Resumiendo,

- partimos de `UnicodeData.txt`, que contiene todo lo que necesitamos, pero es demasiado grande, de modo que
- filtramos los caracteres romanos mediante `Scripts.txt` y *ScriptExtensions.txt*, y

- volvemos a filtrar eliminando los bloques (definidos en `Blocks.txt`) que no nos interesan.

De este modo solucionamos nuestros dos primeros problemas. Ya sólo nos queda uno.

> Esta incursión por el universo Unicode puede haber resultado un poco árida, pero el hecho de desarrollar el soporte nosotros mismos nos permite tener pleno control y modificar el programa a nuestro gusto, incorporando, por ejemplo, si lo juzgamos conveniente, descomposiciones nuevas, por ejemplo para letras como «Æ», que Unicode no descompone, ya que no se consideran ligaduras, pero a nosotros sí que nos interesa descomponer, o decidiendo sobre la marcha a qué bloques Unicode damos soporte, por poner sólo dos ejemplos.



7 Ordenar por relevancia

Para abordar nuestro cuarto problema, *ordenar por relevancia*, necesitamos definir algún tipo de indicador que mida la relevancia de una página. Por suerte, los algoritmos que llevaron a la fama a Google son públicos, están descritos en muchas páginas de internet y son relativamente fáciles de implementar, además de ser muy elegantes desde el punto de vista técnico y matemático.

¿En qué consiste la *relevancia* de una página? En realidad, no lo sabemos bien; entre otras cosas, porque la relevancia es un concepto *subjetivo*: una determinada página puede ser relevante *para mí* y no serlo para mi vecina o mi portera; recíprocamente, es seguro que hay páginas que le parecen relevantes, incluso admirables, a mi portera, y a mi no me interesan nada, y así sucesivamente. Parece claro, pues, que con un concepto subjetivo de relevancia no se puede hacer un programa; seguramente no es lo que empezó haciendo Google.

En efecto: lo que empezó haciendo Google fue *definir* la relevancia de un modo bastante increible: la relevancia de una página P es una cantidad constante que,

1. Por una parte, recibe de las páginas que enlazan a ella. Cada página X que enlaza a P le *envía* una parte, calculada de un modo que veremos enseguida, de su propia relevancia. La suma de las partes de relevancia obtenidas de cada una de las páginas que enlazan con P es la relevancia r de P.

2. Por otra parte, *distribuye* entre las páginas enlazadas por P, del siguiente modo: si la relevancia de P es r y P tiene n enlaces, entonces P «entrega» o «distribuye» una n-esima parte de su relevancia a cada uno de sus enlaces, es decir, distribuye su relevancia «equitativamente».

En nuestro caso, las *páginas* estarán determinadas por los *elementos* de nuestra Biblioteca (libros, autores, editoriales, ciudades, ejemplares, etcétera) y los *enlaces* por las *relaciones* entre esos elementos. Por ejemplo, si Alianza Editorial tuviese una relevancia de 10.000 y hubiese 250 libros en la Biblioteca publicados por esa editorial (y además no existiese ningún otro tipo de vínculo para las editoriales que los vínculos editorial-libro), entonces cada libro publicado en alianza recibiría una relevancia de $10.000/250 = 40$ por el hecho de estar publicado en esa editorial.

La pregunta que cabe hacerse es, ¿existe la relevancia?, es decir, ¿existe una asignación de valores para cada página que cumpla las propiedades anteriores, para todos y cada uno de los elementos de la Biblioteca? Y, si existe, ¿cómo calcularla?

> En términos matemáticos, y suponiendo para simplificar que las claves de los elementos van de 1 a n, ¿podemos encontrar un *vector de relevancia*
>
> $$R = \begin{bmatrix} r_1 \\ ... \\ r_i \\ ... \\ r_n \end{bmatrix},$$
>
> donde r_i es la relevancia correspondiente a la página (elemento) i, que permanezca invariable si la relevancia se distribuye? Observemos que la distribución de la relevancia se puede representar de un modo matricial: si cada página i tiene n_i enlaces, entonces m_{ij} será $1/n_j$ cuando i enlace con j y 0 en caso contrario:
>
> $$M = \begin{bmatrix} 0 & 1/10 & ... & 1/2 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 1/5 & 1/10 & ... & 0 & ... & 1/5 \\ ... & ... & ... & ... & ... & ... \\ 0 & 1/10 & ... & 1/2 & ... & 1/5 \end{bmatrix}.$$

En el ejemplo, la página 1 tiene 5 enlaces, uno de ellos a (digamos) k; la página 2 tiene 10 enlaces, tres de ellos a 1, k y n; etcétera. Ahora, lo que estamos buscando es que

$$MR = R,$$

o, en notación escalar, que para cada i

$$r_i = \sum_{j=1}^{n} m_{ij} r_j,$$

es decir, un vector I que sea invariante ante la multiplicación por M, lo que se denomina técnicamente un *autovector* de M.

Lo sorprendente es que este cálculo pueda realizarse y de hecho sea fácil hacerlo (estamos resolviendo simultáneamente un sistema de ecuaciones con miles de incógnitas): basta con asignar un valor arbitrario de relevancia a cada página (en general, se suele asignar 1) y aplicar la definición de relevancia (es decir, distribuir la relevancia equitativamente entre todos elementos a los que enlazo y recibir la relevancia de los elementos que enlazan a mí). Con esto se consigue, claro está, unos valores distintos de relevancia; pero nadie nos impide repetir el proceso, hasta conseguir unos nuevos valores, y repetirlo otra vez, etcétera.

A medida que realizamos estas iteraciones, los valores de relevancia se van estabilizando, hasta quedar completamente fijos, y esto es así, en conjunto, *para todos los elementos*. Es lo que en matemáticas se conoce como *convergencia*: partiendo de una aproximación inicial arbitraria e iterando el proceso se obtienen los deseados valores de relevancia.

De hecho, los algoritmos que Google usaba al principio (ahora son mucho más sofisticados, además de secretos, e incorporan muchos otros factores) incorporaban también lo se conoce como el *factor de amortiguación*. La idea es que, después de seguir una serie de enlaces, al final uno se para en una página determinada; no se siguen enlaces indefinidamente y por tanto no es correcto pensar que la relevancia fluye sin resistencia. Este factor a suele definirse arbitrariamente en un 85%; visto de otro modo, se supone que hay una probabilidad $(1-a)$ (en nuestro caso, un 15%) de dejarlo estar y empezar a nevegar de nuevo, eligiendo una página cualquiera. La fórmula para calcular la relevancia de una página se modifica del siguiente modo: si N es el total de páginas y a es el factor de amortiguación, la relevancia de una página p será $(1-a)/N$ más a multiplicado por la suma de las relevancias «enviadas» por las páginas que enlazan con p. La solución al problema planteado por esta modificación de las fórmulas puede también calcularse iterativamente.

Para cada i, tendremos

$$r_i = \frac{1-a}{N} + a \sum_{j=1}^{n} m_{ij} r_j.$$

Con estas ideas básicas, es muy sencillo implementar un programa que realice el cálculo iterativo de la relevancia y lo almacene en una columna destinada al efecto en la tabla de elementos. Una vez hecho esto, basta con ordenar los resultados de la búsqueda por relevancia para haber resuelto nuestro cuarto y último problema.

El cálculo de la relevancia es computacionalmente intensivo, por lo que deberá realizarse en un proceso *batch*.

8 Ejemplos: los beneficios secundarios de nuestro modo de indexar

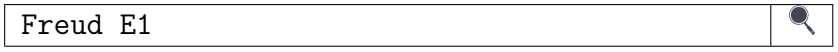
Un beneficio secundario, no específicamente buscado pero extraordinariamente útil, de nuestro modo de indexar, es que permite realizar consultas muy complejas en un lenguaje casi natural. Vamos a verlo mediante determinados ejemplos.

Supongamos, para fijar las ideas, que la Biblioteca cuenta con una serie de elementos que representan estanterías; los ejemplares de los libros, que tienen un identificador externo, se almacenan físicamente en una posición ordinal determinada de cierta estantería. Puesto que nos hemos propuesto indexarlo todo, los elementos principales, es decir, los libros, indexan los números de sus ejemplares y los nombres de sus correspondientes estanterías. Supongamos ahora que la Biblioteca se halla físicamente repartida en varias salas, denominadas `Sala1`, `Sala2`, ..., `Salan` y que en cada una de estas salas encontramos varias estanterías, que reciben un nombre compuesto del indicador de sala, la notación `An` para referirse al armario número n y la notación `Em` para referirse a la estantería m, contando de abajo arriba. Así,

por ejemplo, la estantería `Sala2 A1 E4` referiría a la estantería número 4 del primer armario de la sala 1. Ahora, la búsqueda

encuentra, si existen, los libros de Freud presentes en el armario 1 de la sala 2, además de contarlos aproximadamente.

> Decimos «aproximadamente» porque es posible que el buscador encuentre otras cosas: libros de Anna Freud, por ejemplo, o libros en los que la palabra `Freud` aparezca en el título, pero de todos modos es probable que la aproximación sea buena y, en cualquier caso, la funcionalidad es excelente. En casi todos los ejemplos que siguen se podrían hacer consideraciones similares, con las que no aburriremos al lector.

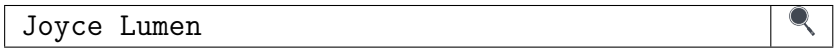


nos da los libros de Freud que están en la estantería superior de cualquier armario de cualquier sala. No es una consulta tan tonta: a veces uno busca eso, por ejemplo porque sólo recuerda ese detalle.

Nos da los libros de Freud publicados por Alianza, y naturalmente

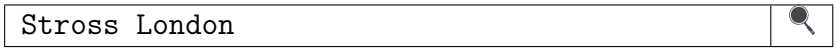
nos da los libros de Freud publicados por Alianza que se encuentran físicamente en la Sala 3. Por otra parte,

nos da los libros de Freud publicados en la colección *El libro de bolsillo* de Alianza Editorial;



los libros de Joyce publicados por Lumen;

todo lo que tenemos de Hermann Hesse, incluyendo (y en primer lugar, porque es *más relevante*) la ficha del autor, que lista exactamente todos sus libros;



nos da los libros de Charles Stross cuya ciudad de edición es Londres, y

restringe la búsqueda a los libros cuyo año de edición es 2012.

9 Conclusiones

Es una funcionalidad maravillosa: la interfaz es invisible, se busca como en Google, cosa que cualquiera sabe hacer y, además, tenemos, de premio, un sistema de búsquedas avanzadas en lenguaje casi natural sin tener que implementar farragosos sistemas avanzados de búsqueda. La impresión que uno recibe, cuando se encuentra con cosas así sin haberlas buscado, es la de que, de algún modo, se han tomado, a nivel técnico, las decisiones correctas.

En resumen: hemos presentado un sistema de búsquedas para bibliotecas fácil, barato, sencillo, que no requiere de una gran especialización y que, sin embargo, produce resultados excelentes. Las partes más *hardcore* del proyecto (el soporte para Unicode y la implementación de la relevancia) han sido desarrolladas por un solo programador trabajando a tiempo parcial en menos de dos meses y usando un lenguaje de programación muy anticuado. Aunque es claro que el producto podría mejorarse en varios aspectos (cálculo incremental de la relevancia, agrupamiento de palabras por familias, etcétera), la diferencia entre lo que presentamos y los buscadores «tontos» es abismal. Esperamos que este artículo sirva de estímulo para que los programadores se animen a dotarlos de más inteligencia; pasarán un buen rato y los usuarios se lo agradecerán.

10 Agradecimientos

La psicoanalista Norma Cirulli me prestó su paciencia, su apoyo y su afecto alrededor de 2004, cuando desarrollé la primera versión de este programa, además de discutir conmigo el conjunto de prestaciones iniciales. Además,

ha tenido la amabilidad de realizar una revisión ortotipográfica completa del texto.

Los datos de la versión anterior de la Biblioteca no incluían información sobre colecciones, la Biblioteca sufrió un traslado físico que desorganizó las estanterías y, además, se hacía necesaria una revisión a fondo de los datos, introducidos al principio por personas muy diversas y que en algunos casos no tenían la preparacion suficiente. Los psicoanalistas Silvina Fernández, María del Mar Martín y Olga Palomino, además de nuestra secretaria, Laura Blanco, revisaron los libros uno por uno y completaron y enriquecieron las fichas, de modo que la primera revisión de los datos está terminada en el momento de escribir este informe.

Andreu Veà me persuadió de que compartiera las ideas utilizadas al crear y refinar la biblioteca, pues se trata de un problema recurrente en muchas organizaciones, y además se tomó el trabajo de sugerir cambios en la presentación y la adición de ilustraciones para que el texto respirase,

Laura Blanco, Norma Cirulli, Silvina Fernández, María del Mar Martín, Olga Palomino, David Palau y Andreu Veà leyeron pacientemente varias versiones del manuscrito y lo enriquecieron con multitud de observaciones, comentarios y sugerencias. A todas las personas citadas, mi más sincero agradecimiento.

Apéndices

A Historia de la Biblioteca del EPBCN

La Biblioteca del EPBCN se constituyó el 27 de enero de 2001. Su denominación inicial fue *Biblioteca del Seminario*, debido a su vinculación con los por entonces recientemente inaugurados seminarios *de Psicoanálisis Freudiano*[15] y *de Psicoanálisis, Grupos e Instituciones.*[16] La web del Espacio vino cinco meses más tarde, el 1 de junio del mismo año. En el momento de redactar este informe, la Biblioteca cuenta con más de 1,700 referencias bibliográficas para unos 1850 ejemplares (algunos, muy usados, están duplicados, o se dispone de ediciones distintas) distribuidos en aproximadamente 400 editoriales, 60 ciudades de edición y 1,000 autores. Los socios de la Biblioteca, alumnos, docentes y psicoanalistas del EPBCN, pagan una pequeña cuota mensual, que se destina a la compra de libros y les da derecho a pedir prestados hasta dos ejemplares a la vez por un máximo de 15 días, renovables si nadie más ha pedido el mismo libro durante ese tiempo. También se dispone de obras que no pueden ser retiradas, pero si consultadas en las mismas instalaciones del EPBCN.

Para poder gestionar el almacenamiento, la localización y el préstamo de libros se desarrolló un programa especializado, que cumplió fielmente su cometido hasta 2011. En ese momento migramos las webs del EPBCN a la nube (previamente el servidor web estaba alojado fisicamente en las instalaciones del EPBCN en el número 21 de la calle Enric Granados de Barcelona), pero no pudimos migrar el programa.

> Las presión de las necesidades institucionales no nos dejaba, en ese momento, ni un respiro para programar, de modo que tuvimos que recurrir a una pequeña chapuza: como la migración del programa era muy costosa (estábamos usando una versión muy vieja de IBM DB2/2 sobre sistema operativo OS/2 Warp Server, descatalogado desde hacía más de diez años,

[15] El 25 de noviembre de 2000, a continuación de las Jornadas Inaugurales *Psicoanálisis, grupos e instituciones*, celebradas el 28 de octubre.

[16] El 24 de noviembre

y una aplicación de uso interno de IBM llamada `EZSelect`, modificada por Francesc Rosés para interfacear con VisPro/REXX, que no resistió bien el paso a Windows), hackeamos una versión de WordPress y, jugando con las taxonomías para no perder información, volcamos la información a un blog. Esto degradó bastante la calidad del servicio, pero era un parche temporal hasta que pudiésemos rehacer el programa y nos permitió seguir trabajando sin perder los datos.

En los últimos meses hemos podido desarrollar el nuevo programa que describimos aquí. Aunque retomamos los mejores conceptos del anterior, nos esmeramos en reescribirlo desde cero, incorporando ideas nuevas (las novedades de esta versión son el soporte para Unicode y el cálculo de relevancia) y rehaciendo con más perfección las antiguas. Aunque ninguna de las técnicas utilizadas es especialmente novedosa, su combinación quizá sí lo sea, especialmente para un programa desarrollado *in-house* y destinado a gestionar una biblioteca relativamente pequeña. Lo presentamos por varios motivos: para dejar constancia histórica de su desarrollo, por un criterio de transparencia institucional y, por último, para compartir las ideas que se implementan con quien pueda encontrarlas de utilidad.

B Especificaciones técnicas de la aplicación

La Biblioteca del EPBCN está implementada en ooRexx,[17] la versión de código abierto con extensiones orientadas a objeto del ya clásico Rexx[18] creado a finales de la década de 1970 por Mike Cowlishaw de IBM e incorporado en 1983 al sistema para *mainframes* IBM VM/CMS.

El servidor Web es Apache 2.2; para esta aplicación, no se necesita ningún módulo no estándar.

El programa principal reside en un fichero de 100 KB y 3.100 líneas denominado `bib` (sin extensión) que reside en la raíz del servidor, de modo que el acceso a la Biblioteca se consigue usando la muy sencilla URL `http://www.epbcn.com/bib/`. El fichero está definido en `httpd.conf` de modo que sea gestionado por `REXXHTTP`,[19] un sistema de código abierto que desarrollé en 2006 y que en este caso funciona como un encapsulador de la complejidad del protocolo CGI (nos referimos a la Common Gateway Interface[20] para aplicaciones web). La aplicación analiza el `PATH_INFO` y proporciona una navegación «natural», de modo que el hecho de que la implementación se realice con un solo CGI es completamente transparente.

Como Rexx no incorpora soporte para Unicode, lo hemos tenido que escribir nosotros mismos, como se explica más arriba. En particular, se ha escrito una pequeña biblioteca para la gestión de tiras en formato UTF-8.

El sistema gestor de base de datos es SQLite;[21] el acceso a la base de datos se realiza mediante ooSQLite,[22] una extensión de ooRexx destinada a tal efecto.

[17] `http://www.oorexx.org/`.
[18] `http://en.wikipedia.org/wiki/Rexx`, ver también `http://www.rexxla.org/`.
[19] `http://www.epbcn.com/personas/jmblasco/publicaciones/rexxhttp.pdf`.
[20] `http://en.wikipedia.org/wiki/Common_Gateway_Interface`.
[21] `http://www.sqlite.org/`.
[22] `http://sourceforge.net/projects/oorexx/files/ooSQLite/`.