Reingeniería de “Presentes”: Una Base de Datos sobre el Accionar del Terrorismo de Estado

Carlos E. Barcia

Director: Dr. Franco M. Luque

Marzo 2015

Resumen

"Presentes" es un software utilizado por el Archivo Provincial de la Memoria para la investigación del accionar del terrorismo de estado en la provincia de Córdoba. Este sistema informático contiene una base de datos que condensa y organiza información recopilada a lo largo de años de constante investigación.

El sistema no fue desarrollado mediante un proceso de producción de software adecuado y necesita un conjunto importante de modificaciones y extensiones. En consecuencia, en este trabajo proponemos realizar una reingeniería de "Presentes", teniendo en cuenta requerimientos relevados y planteados por el Archivo de la Memoria.

Como parte del proceso de reingeniería, realizamos una ingeniería inversa de la base de datos existente, para esclarecer su funcionamiento actual, analizar sus falencias y proponer soluciones para ellas. Además, realizamos una migración de tecnología de base de datos a un motor más adecuado a los requisitos del sistema. Finalmente, diseñamos e implementamos una nueva base de datos utilizando conceptos de los modelos de bases de datos objeto-relacional y documental. En el proceso, mejoramos la calidad de los datos, eliminando inconsistencias y redundancias para adaptarlos a los cambios en la estructura y a las nuevas restricciones introducidas en cada etapa de la reingeniería.

Clasificación:

- E.2 [Data Storage Representations]: Object representation;
- H.2.1 [Database Management]: Logical Design – Data models;
- H.2.3 [Database Management]: Languages – Query languages;
- H.2.4 [Database Management]: Systems – Relational databases;
- D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering;

Palabras clave: Sistema Informático Presentes, Ringeniería de Software, Ingeniería Inversa, Bases de Datos, SQL, Modelo de Dato Relacional, Modelo de Dato Objeto-Relacional, Modelo de Dato Documental, Mapeo Objeto Relacional, PostgreSQL, MySQL.

Agradecimientos

A mis amigos que siempre me dieron su apoyo.

A Cecilia Díaz por impulsarme a terminar la carrera.

A Marcelo Yornet y Martin Onetti por la gran cooperación para realizar este trabajo.

A mi director de tesis Franco Luque que hizo posible este logro gracias a su paciencia, esfuerzo y dedicación.

A mi padre Alfredo y a mis hermanos María y Gaspar por creer en mi.

Y en especial, a mi madre Mirta por su constante ayuda y apoyo a lo largo de este trabajo y de toda mi carrera.

Índice general

Capítulo 1

Introducción

1.1. Archivo Provincial de la Memoria

El Archivo Provincial de La Memoria creado a partir de la Ley N° 9286 titulada "LEY DE LA MEMORIA", promulgada por la Legislatura de la Provincia de Córdoba en marzo de 2006 en adhesión al Decreto N° 125920 del Poder Ejecutivo Nacional. La ley establece el marco para el desarrollo de sus funciones, entre las que destacamos la responsabilidad de recopilar y organizar documentación relacionada con violaciones a los derechos humanos y el accionar del terrorismo de estado en la Provincia de Córdoba en la última dictadura militar (1976-1983). Para esto fija como objetivo entre otros:

> *"Desarrollar los métodos adecuados, incluida la duplicación y digitalización de los archivos y* ***la creación de una base de datos*** *para analizar, clasificar y ordenar informaciones, testimonios y documentos, de manera que puedan ser consultados por los titulares de un interés legítimo, dentro del Estado y la sociedad civil, en un todo conforme a la Constitución, los instrumentos internacionales de derechos humanos y las leyes y reglamentos en vigencia."*

La sede del Archivo Provincial de la Memoria (APM) se encuentra en Pje. Sta. Catalina 66, lugar donde funcionaba el ex Departamento de Inteligencia de la Policía de la Provincia de Córdoba conocido como "D2", uno de los principales Centros Clandestinos de Detención. Actualmente en esa dependencia opera el Área de Informática y Digitalización del APM donde utilizan el Sistema Presentes. Allí se concretaron las reuniones concernientes al presente trabajo con el responsable y desarrollador del sistema Marcelo Yornet.

1.2. Historia del Sistema Informático Presentes

Presentes consta de una base de datos que almacena información sobre personas, documentos, causas judiciales y lugares vinculados a la última dictadura militar en la provincia de Córdoba. Se comenzó a desarrollar en el año 2001 por Marcelo Yornet y la asociación H.I.J.O.S. Regional Córdoba, en 2011 fue donado al APM.

Al momento de comenzar este trabajo la base de datos de Presentes que utiliza el gestor de base de datos MySQL, consta de 102 tablas que están relacionadas entre si mediante el uso de 24 claves foráneas. La parte del sistema encargada de interactuar con la base de datos y la interfaz gráfica de usuario está desarrollada en Visual Basic. Por lo que se pudo apreciar[1] sus componentes no están correctamente modularizados. A lo largo del trabajo se usa el término "Interfaz" para referirse a esta parte del sistema.

1.3. Este Trabajo

La labor de campo realizado en este trabajo especial consta de tres etapas: ingeniería inversa de la base de datos, migración del gestor de base de datos y por último su completa reingeniería.

Como comienzo en el Capítulo 2 se introducen conceptos teóricos sobre bases de datos, que son usados a lo largo del proyecto.

La primera etapa es detallada en el Capítulo 3 y consiste en comprender el funcionamiento de la base de datos y el porqué de las decisiones de diseño que llevaron a su implementación. Aquí se dilucidan la mayoría de los problemas que presenta, algunos son solucionados en esta etapa y otros que requieren mayores cambios son pospuestos para las siguientes etapas.

La segunda etapa, Capítulo 4, comprende la migración de gestor de base de datos a uno que se adecua más a las necesidades que requieren nuevas funcionalidades a incorporar al sistema. Algunas son agregadas en esta etapa y el resto en la última. También se solucionan problemas detectados en la primer etapa, a la vez que aparecen nuevos.

En la tercera y última etapa, Capítulos 5 y 6, se solucionan problemas más generales para lo cual se diseña y desarrolla una nueva base de datos. Esta se puede decir que es una combinación de distintos modelos de datos y técnicas para almacenar y relacionar datos entre sí. Los registros de la base de datos original son normalizados y adaptados para ser migrados a la nueva base de datos. Aquí se encontraron varios registros que ocasionaron dificultades, en su mayoría por una incorrecta estructuración de los datos o la errónea utilización de su estructura.

Si bien estas etapas fueron desarrolladas en el orden que se listan, en algunas ocasiones se superponen. El objetivo de comprender el funcionamiento de la base de datos (ingeniería inversa) se extiende en las etapas posteriores, dado que en ellas se hace un acercamiento más detallado de la misma. Esto ayuda a dilucidar características pasadas por alto en la primer etapa. Además, las modificaciones de la base de datos son realizadas en menor medida en etapas previas a la reingeniería completa.

En el Capítulo 7 se manifiestan las conclusiones del trabajo realizado y los posibles trabajos futuros.

[1] No se tuvo acceso al código de la Interfaz.

Capítulo 2

Marco Teórico

En el presente capítulo se introducen conceptos utilizados a lo largo del trabajo. Principalmente se exponen distintos modelos de datos que son herramientas conceptuales utilizadas para describir datos, las relaciones entre ellos, su semántica, operaciones y restricciones de integridad [SKSP02]. Los modelos que abordaremos son el relacional, objeto relacional y semiestructurado.

También se presenta el lenguaje de consultas a base de datos Structured Query Language (SQL), el formato de intercambio JavaScript Object Notation (JSON) que utilizamos para almacenar información semiestructurada y el método de mapeo objeto relacional, que almacena información persistente de objetos en una base de datos relacional.

2.1. Modelo de Datos Relacional

El modelo de base de datos relacional fue propuesto por Edgar Frank Codd en 1970 [Cod70]. Este modelo utiliza tablas para almacenar datos y relaciones entre estos. En dichas tablas cada fila es un registro y cada columna un atributo.

Los valores de los atributos deben pertenecer a un conjunto de valores previamente definidos, o sea que cada atributo está asociado a un dominio. Cada registro (visto como una tupla) es un elemento del producto cartesiano de dichos dominios. Luego, la tabla puede ser vista como un subconjunto de tuplas (registros) del producto cartesiano de los dominios de los atributos (Fig. 2.1). Esto resulta ser la definición de relación[1] en la teoría de conjuntos. El modelo esta basado en esta rama de la matemática, sumada a la lógica de predicados.

A cada columna se le asigna un nombre. De esta forma una tabla queda definida por los nombres de sus columnas, los dominios de estas y las tuplas que almacena.

[1] El modelo relacional recibe su nombre por estar basado en el concepto matemático de relación y no por las posibles relaciones entre tablas.

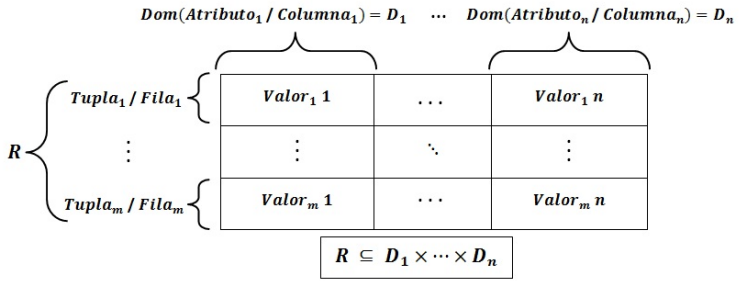

Figura 2.1: Representación del concepto de tabla en el modelo relacional.

En [Cod72] se presentan dos tipos de notaciones para la consulta de datos en el modelo relacional: el álgebra relacional y el cálculo relacional. Ambos tienen el mismo poder expresivo.

2.1.1. Álgebra relacional

El álgebra relacional es una notación algebraica en la cual las consultas se expresan aplicando a las relaciones operadores especializados, basados en la teoría de conjuntos. Dichas operaciones son aplicadas a las relaciones para generar una nueva relación que contenga los datos a obtener [Gro99].

A continuación listamos algunos de estos operadores:

SELECT: extrae tuplas a partir de una relación que satisfaga la restricción dada.

PROJECT: extrae atributos específicos de una relación.

PRODUCT: construye el producto cartesiano entre dos relaciones.

UNION: supone la unión de dos relaciones.

INTERSECT: construye la intersección de dos relaciones.

DIFFERENCE: es el conjunto que diferencia dos relaciones.

JOIN: conecta dos relaciones por sus atributos comunes.

2.1.2. Cálculo Relacional

El cálculo relacional es un lenguaje declarativo de consultas, basado en el cálculo de predicados de primer orden. Las consultas se realizan formulando restricciones lógicas mediante predicados, los cuales deben ser satisfechos por las tuplas que se desean obtener como resultado.

Se puede dividir al cálculo relacional en dos clases, según el tipo de variable que utiliza:

Cálculo relacional de tuplas: utiliza como tipo de variables a tuplas pertenecientes a una relación.

Cálculo relacional de dominio: utiliza como tipo de variables a los dominios asociados a los atributos de las relaciones.

El cálculo y álgebra relacional son equivalentes, dado que toda expresión del cálculo relacional puede ser expresado mediante álgebra relacional y viceversa.

2.1.3. El Lenguaje de consulta SQL

El lenguaje estándar (ANSI/ISO) de consulta Structured Query Language (SQL) [EMK+04] es un lenguaje declarativo para sistemas de base de datos relacional. Fue desarrollado basándose en el cálculo relacional de E. F. Codd, expuesto en la sección anterior. Además del cálculo de tuplas, que permite recuperar datos almacenados, el lenguaje posibilita manipular y almacenar datos. Incluye operaciones aritméticas, comparaciones, asignaciones, restricciones, etc.

A continuación se describen los principales comandos SQL utilizados para la creación y manipulación de tablas y registros [PS+05] [SKSP02].

2.1.3.1. `CREATE`

El comando `CREATE` es utilizado para la creación de bases de datos, tablas, vistas y otros tipos de objetos como restricciones, procedimientos almacenados, etc.

El ejemplo a continuación muestra la utilización del comando para la creación de la tabla `tabla_1` con las columnas `col_1, col_2, ..., col_n`, cuyos posibles valores deben pertenecer a los dominios definidos por los tipos de datos `tipo_dato_1, tipo_dato_2, ... , tipo_dato_n`, respectivamente.

```
CREATE TABLE tabla_1
(
  col_1 tipo_dato_1,
  col_2 tipo_dato_2,
  ...,
  col_n tipo_dato_n,
  PRIMARY KEY (col_v)
);
```

El comando `PRIMARY KEY (col_v)` es una restricción denominada Clave Primaria impuesta sobre la columna `col_v` con $v \in \{1, 2, \ldots, n\}$, que limita sus valores para que cumpla con la función de identificar inequívocamente los registros dentro de la tabla. Esta columna debe contener valores únicos y no puede contener `NULL`, el cual es el denominado valor nulo que sirve para indicar que a un atributo no se le asignó un valor. Una tabla puede tener una sola clave primaria, la cual puede estar compuesta de una o varias columnas.

2.1.3.2. `ALTER`

La sentencia `ALTER` modifica una tabla u objeto ya existente en la base de datos. En este ejemplo se agrega la restricción Clave Foránea a la columna `col_1` del ejemplo anterior.

```
ALTER TABLE tabla_1
  ADD FOREIGN KEY (col_1)
  REFERENCES tabla_2(col_tabla_2);
```

La Clave Foránea es usada para establecer una relación entre las tablas `tabla_1` y `tabla_2` . Las columnas `col_1` y `col_tabla_2` deben tener el mismo dominio y los valores de `col_tabla_2` deben identificar univocamente los registros de `tabla_2`. El valor en la columna `col_1` en cada uno de los registros de `tabla_1` debe ser `NULL` o igual al valor de `col_tabla_2` en algún registro de `tabla_2`.

Esta restricción se puede aplicar sobre una o más columnas de una tabla para que identifique inequívocamente los registros de otra tabla. La tabla que contiene la restricción es comúnmente llamada hijo y a la que hace referencia es denominada tabla padre.

2.1.3.3. `DROP`

Este comando es utilizado para eliminar una tabla u objeto ya existente en la base de datos, como se exhibe continuación.

```
DROP TABLE tabla_1;
```

2.1.3.4. `INSERT`

El comando `INSERT` es utilizado para almacenar registros en una tabla. En el siguiente ejemplo se emplea la sentencia para almacenar en la tabla `tabla_1` la tupla (`dato_1, dato_2, ..., dato_n`).

```
INSERT INTO tabla_1(
       col_1, col_2, ..., col_n)
  VALUES (dato_1, dato_2, ..., dato_n);
```

2.1.3.5. `SELECT`

`SELECT` es utilizado para recuperar columnas de registros de una o más tablas de la base de datos. Para explicar su funcionamiento se presenta el siguiente ejemplo.

```
SELECT col_1, col_tabla_2
  FROM tabla_1, tabla_2
  WHERE p;
```

`SELECT`: Especifica el nombre de las columnas que se desean recuperar de las tablas determinadas por `FROM`. En este caso las columnas `col_1` y `col_tabla_2` de `tabla_1` y `tabla_2`, respectivamente. Es equivalente a la operación **PROJECT** del álgebra relacional.

`FROM`: Establece las tablas de origen (`tabla_1, tabla_2`) de los registros. Es semejante al **PRODUCT** del álgebra relacional.

Aquí también se puede utilizar la operación `JOIN` en vez *", "*, la cual es equivalente al operador **JOIN** del álgebra relacional.

`WHERE`: En esta cláusula se establece el predicado `p` que debe cumplir los registros para ser seleccionados. Este predicado se describe mediante el uso de operadores lógicos y aritméticos proporcionados por el lenguaje. La cláusula se corresponde al predicado **SELECT** del álgebra relacional.

La sentencia `FROM` es obligatoria, mientras que `WHERE` es optativa y en caso de su ausencia se toma por defecto $p \equiv true$.

El comando devuelve una nueva tabla con las columnas establecidas por `SELECT` y los registros recuperados de la o las tablas especificadas en `FROM` que cumplen con el predicado `p`.

2.1.3.6. `UPDATE`

Modifica un registro almacenado en la base de datos. En el ejemplo se cambia el valor de la columna `tabla_1` a `valor_1` en todos los registros de la tabla `tabla_1` que cumplen con el predicado `p`.

```
UPDATE tabla_1
  SET col_1 = valor_1
  WHERE p;
```

2.1.3.7. `DELETE`

Borra un registro almacenado en la base de datos. En el ejemplo se borran las filas de la `tabla_1` que cumplen con la condición `p`.

```
DELETE FROM tabla_1
  WHERE p;
```

2.2. Modelo de Datos Objeto Relacional

El modelo de datos objeto relacional [EN07] es una extensión del modelo de datos relacional al cual se añaden características de la programación orientada a objetos tales como herencia y polimorfismo. Además se suma la posibilidad de definir tipos de datos complejos. Sin embargo, los datos almacenados y sus relaciones siguen teniendo el mismo significado que en el modelo relacional.

Estas características fueron introducidas en el estándar sql-99 [EM99] compatible con SQL-92, lo que simplifica el traspaso de un modelo a otro en una base de datos ya existente.

2.2.1. Implementación en PostgreSQL del Modelo de Datos Objeto Relacional

Existen diversas formas de implementar las características que fueron introducidas en este modelo, pero nos centraremos en la propuesta por el sistema de gestión de base de datos objeto relacional PosgreSQL utilizado en la reingeniería de la base de datos que se realizo como parte de este trabajo.

A continuación se describe brevemente la implementación propuesta por PostgreSQL de estos conceptos:

Clases y objetos: Los objetos son representados por las tuplas que se almacenan en las tablas, estas a su vez actúan como clases. Los atributos de cada objeto quedan entonces representados por las columnas de las tablas (clases).

Herencia: Se permite la herencia entre tablas (clases) mediante la sentencia `inherits`, esto causa que la tabla hijo contenga además de sus propias columnas las columnas de la tabla padre, estableciendo así la relación conceptual "es un".

Si una consulta es aceptada por una tabla padre también lo es para todas sus tablas hijos, a esto se lo denomina consultas polimórficas en analogía con las funciones polimórficas de la programación orientada a objetos.

Tipos de datos complejos: En este modelo se pueden definirse tipos de datos compuestos y funciones que traten con estos para poder usarlos en operaciones, asignaciones, etc. En el capítulo 5 se muestran distintas opciones nativas con que cuenta PostgreSQL para la creación de tipos de datos que se utilizaron en este proyecto.

2.3. Mapeo Objeto Relacional

Otra forma de utilizar conceptos de la programación orientada a objetos es, en vez de que las tablas sean vistas como objetos hacer que estas sirvan para almacenar datos de objetos existentes fuera de la base de datos. Como ya se explicó anteriormente, los sistemas de manejo de base de datos relacionales almacenan datos en colecciones de tablas. Por otro lado, el dominio de los lenguajes orientados a objeto puede ser representado como grafos de objetos interconectados. Cuando se utilizan bases de datos relacionales para almacenar datos persistentes de los objetos surgen diversos problemas conocidos con el término de *Impedance mismatch* [Fin01], que están vinculados a las diferencias entre estos dos modelos.

Una solución a estos problemas es la técnica denominada **Mapeo Objeto Relacional** (ORM[2]). En [BK04] se define al ORM como la persistencia automatizada y transparente de los objetos de una aplicación en una base de datos relacional, utilizando metadatos que describen el mapeo entre los objetos y la base de datos. El ORM funciona de forma reversible, transformando datos de una representación a otra.

2.3.1. Metadatos de Mapeo Objeto Relacional

Las herramientas de ORM utilizan metadatos para especificar como está realizado el mapeo entre los datos persistentes de las clases y las tablas que los almacenan, propiedades y columnas, asociaciones y claves foráneas, tipos de datos del lenguaje de programación y tipos de datos de SQL. Esta información es llamada **Metadatos de Mapeo Objeto Relacional** [BK04] y define la transformación entre las diferentes sistemas de tipos de datos y la representación de las relaciones.

2.3.2. *Shadow Information*

Los objetos para su mantenimiento utilizan información que no pertenece al dominio de sus datos. Como por ejemplo las claves primarias (particularmente cuando son claves subrogadas), marcas de control de concurrencia, etc. Este

[2]ORM son las siglas de *"Object Relational Mapping"*

tipo de información es conocida como *Shadow information* [Amb02], muchas veces es necesario almacenarla para que algunas propiedades de los objetos sean persistentes.

2.3.3. Mapeo de herencia de clases a modelo relacional

A continuación describiremos tres métodos para mapear herencia de clases en una base de datos relacional expuestos por Scott Wambbler en [Amb02]. A modo de ejemplo se utilizan las clases ilustradas en el diagrama de clases de la Figura 2.2 donde se puede observar que la clase `persona` es heredada por las clases `victima` y `represor`.

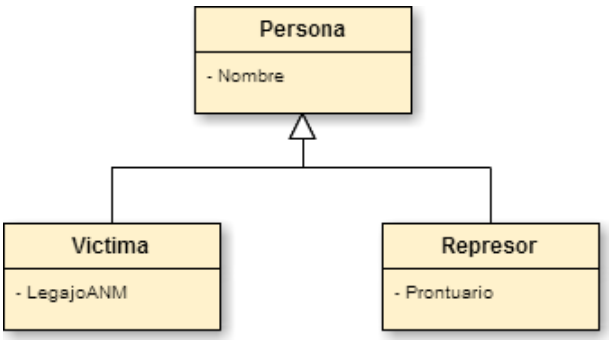

Figura 2.2: Arbol jerarquico que representa la herencia de clases.

- **Mapear todas las clases a una tabla**

  En este caso todas las clases del árbol jerárquico son mapeados a una única tabla (Fig. 2.3). Se utiliza el campo `tipo_persona` para distinguir a que clase pertenece y las columnas de los atributos que no corresponden con dicha clase son completadas con el valor `NULL`.

  El campo `persona_poid` [3] es un identificador de objeto persistente que es utilizo como clave primaria subrogada.

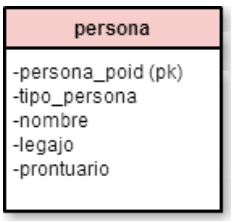

Figura 2.3: Tabla resultante de mapear todas las clases a una sola tabla.

- **Mapear cada clase concreta a una tabla**

  En este método cada clase concreta es mapeada a una tabla que contiene los atributos de la clase y los heredados de la clase padre. Además, cada tabla cuenta con su propia clave primaria `poid`.

[3] El sufijo poid viene de las siglas del termino en ingles *Persistent Object IDentifier*

En la figura 2.4 se puede observar como solo se implementan las tablas de las clases concretas `victima` y `represor`, pero no de la clase abstracta `persona`.

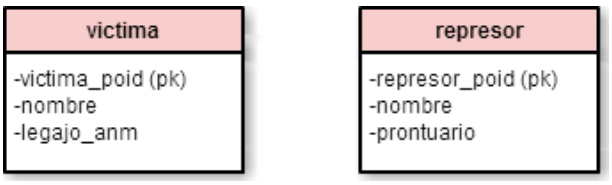

Figura 2.4: Tabla resultante de mapear cada clase concreta a una tabla.

- **Mapear cada clase a una tabla**

  Este método implementa una tabla por cada clase y cada tabla contiene una columna por atributo específico de la clase. De esta forma, como se puede observar en la figura 2.5, los atributos de la clase `victima` quedan divididos en: la tabla `victima` el atributo específico `legajo_anm` y en la tabla `persona` el atributo heredado `nombre`.

  La clave primaria `persona_poid` de la tabla `persona` es utilizada, mediante el uso de claves foráneas, como clave primaria de las tablas `victima` y `represor` para mantener la relación de estas con la tabla `persona`.

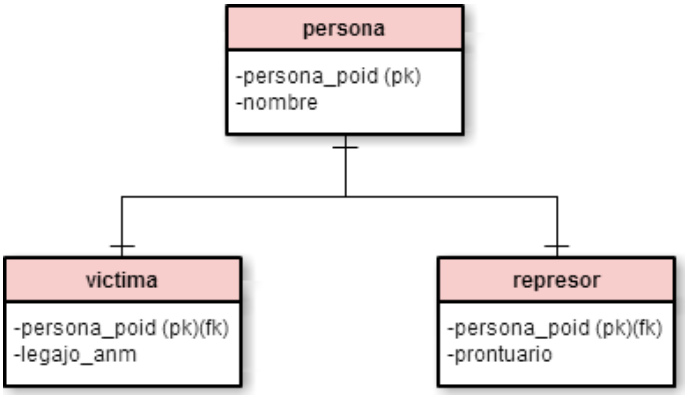

Figura 2.5: Tabla resultante de mapear cada clase a una tabla.

2.4. Base de datos documental

Las bases de datos documentales están diseñadas para almacenar como unidad básica una noción abstracta de documento. Los documentos son agrupados en colecciones, basándose en un modelo de datos semiestructurado, que se caracteriza por permitir que datos de un mismo tipo puedan tener diferentes atributos y sean autodescriptivos [SKSP02] [MSS01].

A diferencia de las bases de datos relacionales, que almacenan los datos en filas repartidas en distintas tablas, las documentales los almacenan en una sola unidad, el documento. Esto mejora el rendimiento, ya que no es necesario realizar

operaciones complejas para recuperar datos que están relacionados en distintas tablas. Además, al no imponer un esquema fijo para su almacenamiento, se obtiene ventajas en escalabilidad.

Las bases de datos documentales generalmente utilizan como formato para almacenar y retornar información, la notación de lenguajes de marcado XML (*eXtensible Markup Language*) o el formato de intercambio JSON (*JavaScript Object Notation*). Este ultimo es detallado a continuación dado que es el utilizado para almacenar distintos tipos de documentos en la implementación del nuevo diseño de Presentes propuesto en el Capitulo 5.

2.4.1. JSON

JSON es un formato de texto para facilitar el intercambio de datos entre distintos lenguajes de programación. Forma parte de la definición del estándar [jso13] en el cual esta basado Javascript. Este formato provee una notación para expresar colecciones de pares nombre-valor y listas de valores de un modo anidado. Los posibles valores de JSON pueden ser:

Par nombre-valor
: En los valores de tipo nombre-valor se divide el "nombre" (que debe ser una cadena de caracteres) del "valor" mediante el símbolo ":", cero o más de estos pares deben ser encerrados entre llaves y separados uno del otro mediante una coma de la siguiente forma:

```
{"nombre_1": valor_1,
 ...,
 "nombre_n": valor_n
}
```

Lista de valores
: Una lista de valores son cero o más valores[4], separados por coma y encerrados entre corchetes como se muestra a continuación:

```
[valor_1, ...,  valor_n]
```

Número
: Los valores de tipo número son representados en base 10, sin cero a la izquierda, pueden ser negativos y tener una parte fraccionaria mediante el uso de un punto. También se puede utilizar la notación científica mediante los símbolos "e" o "E" y + o −.

Cadena de caracteres
: Las valores de cadena de caracteres son secuencias de caracteres[5] encerradas entre comillas dobles.

Otros valores
: Ademas de los listados anteriormente, JSON permite los valores **true**, **false** y **null**.

[4] Se considera el orden de los valores.
[5] Deben cumplir con el estándar Unicode [Con11]

Siguiendo la convención propuesta por este formato se pueden representar distintas estructuras de datos de una forma sencilla para la interacción entre distintos lenguajes de programación. Estas características de simpleza y flexibilidad hace de este un formato apropiado para almacenar documentos.

2.4.2. Implementación del Modelo de Datos Semiestructurado en PostgreSQL

PostgreSQL cuenta de forma nativa con cuatro tipos de datos que permiten almacenar información semiestructurada. Además posee una gran variedad de funciones y operadores específicos para las consultas y manejo de cada uno. A continuación se los describe brevemente[Gro14]:

`hstore`: un valor de este tipo es una lista de pares llave-valor. Pueden ser anidados y tanto la llave como el valor son de tipo texto.

`xml`: es utilizado para almacenar documentos en formato XML. Antes de ser almacenado un valor de este tipo, el gestor comprueba que esté correctamente constituido según el estándar XML.

`json`: este tipo de datos almacena valores JSON y al igual que `xml` se comprueba que esté formado según el estándar.

`jsonb`: a diferencia de `json` que almacena los valores de texto tal cual como son introducidos, `jsonb` los descompone en un formato binario para acelerar las consultas. Además no conserva los valores nombre-valor si tienen el nombre repetido, en estos casos mantiene sólo el último valor.

Capítulo 3

Ingeniería Inversa de Presentes y Análisis de Problemas

En este capítulo se aborda la etapa inicial del trabajo, que abarca la comprensión y análisis de la base de datos.

Como primera medida se alcanzó un diseño conceptual de esta, que permitió comprender su funcionamiento y la información que requiere almacenar. Con el diseño sumado a una instancia concreta de la base de datos, se analizaron sus problemas y/o debilidades a la vez que surgieron indicios para mejorar y extender su funcionalidad.

3.1. Estado Inicial de la Base de Datos.

En líneas generales el objetivo de Presentes es coleccionar y relacionar información de damnificados y represores de la última dictadura militar en la Provincia de Córdoba, Argentina. Para ello se utilizan diversos tipos de documentos e información de casos judiciales que los atañen.

La información es almacenada de forma estructurada en una base de datos que está implementada en el gestor de base de datos relacional MySQL.

La base de datos no fue desarrollada a partir de un diseño conceptual específico y bien definido. Más bien, fue realizada y extendida a partir de diversos requerimientos que surgieron a través del tiempo. Esta metodología no fue respaldada con documentación que explicara claramente la estructura de la información almacenada, su propósito, ni funcionalidad.

Como única documentación y punto de partida se dispone del manual de usuario del sistema y el código fuente para la creación de las tablas de la base de datos, por lo que para comenzar a comprender el diseño subyacente fue necesario realizar la ingeniería inversa de la misma.

3.1.1. Ingeniería Inversa

Como primer paso se utilizó la herramienta de diseño, desarrollo y administración de bases de datos MySQL Workbench[1], la cual cuenta con un proceso automático de ingeniería inversa que genera una representación gráfica a partir del código fuente de una base de datos implementada en MySQL. Este esquema muestra las tablas con sus atributos, tipos de datos utilizados, claves primarias y foráneas. En la Figura 3.1 se puede observar el diagrama que da como resultado aplicar este proceso sobre la base de datos de Presentes.

La base cuenta con 102 tablas y 24 relaciones implementadas mediante el uso de claves foráneas. Para comprender mejor este diagrama se organizaron las tablas en siete grupos conceptuales descritos a continuación:

Víctimas: Información vinculada a los damnificados tales como sus datos personales, datos sobre su secuestro, organizaciones en las que militó, etc.

Represores: Información asociada a represores. Aquí se agrupan sus datos personales, fuerza en la que actuaron, rango que poseían, etc.

Relación Victimas-Represores: Tablas que relacionan víctimas con represores. También se compila información de centros clandestinos de detención, escenarios de muchos de los sucesos en los que se vinculan.

Códigos: En este grupo se reunieron tablas que son utilizadas para almacenar los posibles valores de atributos de otras tablas. Por ejemplo: la tabla `tipo_docs_cod` almacena los posibles tipos de documentos como DNI, LC, etc. que debería almacenar el atributo `tipo_documento` de la tabla `af_casos`.

Causas: Datos asociados a causas judiciales tales como: tipo de causa, actos procesales, actores implicados, juzgados en los que se llevan a cabo, etc.

Tareas: Datos que corresponden a los usuarios del sistema, como el nombre de usuario y clave de ingreso, datos personales, tareas asignadas, nivel de acceso, etc.

Fuentes: Diversos documentos legales, diarios, testimonios, etc. así como también tablas que los relacionan entre sí y con víctimas, represores, causas judiciales, etc.

Una vez obtenida la representación gráfica de la base de datos (Fig. 3.1), sumado al manual de usuario[2], datos concretos de la base y a través de diversas entrevistas realizadas al encargado y desarrollador del sistema Marcelo Yornet, se pudo comprender de una mejor forma su funcionamiento y acercarse al diseño conceptual subyacente.

En este punto se dilucidaron variados problemas de diseño e implementación que posee la base de datos. En la siguiente sección se hará una descripción y análisis de dichos problemas a la vez que se presentan soluciones y mejoras a los mismos.

[1] http://www.mysql.com/products/workbench/

[2] Al momento de realizar esta etapa el manual de usuario se encontraba en desarrollo. Este explica el uso de la interfaz de usuario.

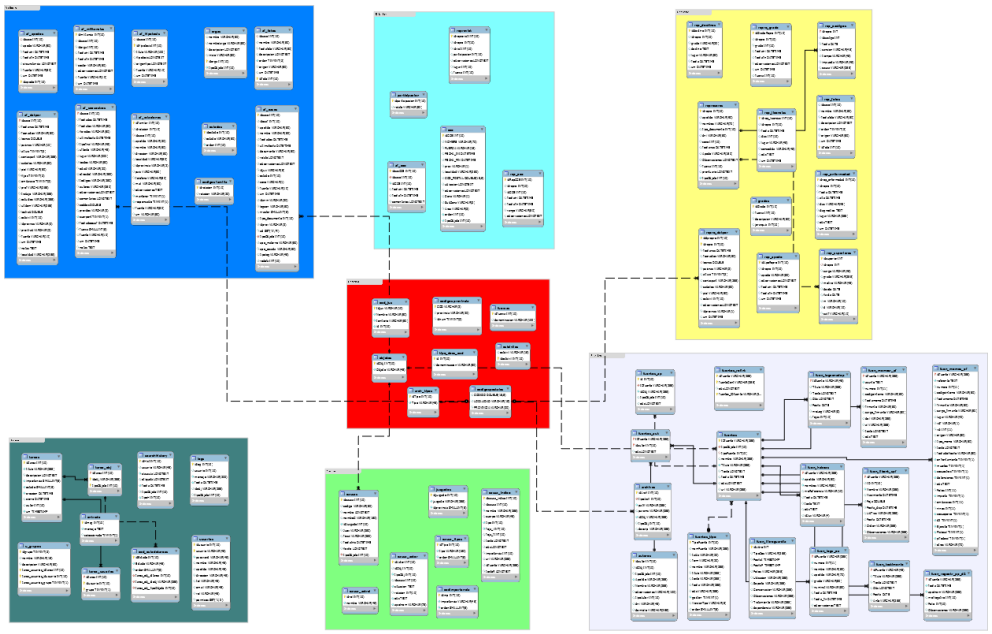

Figura 3.1: Diagrama de las tablas y restricciones de la base de datos de Presentes organizado según los grupos conceptuales: **Víctimas**, **Represores**, **Relación Victimas-Represores**, **Códigos**, **Causas**, **Tareas y Fuentes**.

3.2. Problemas y Soluciones Propuestas

Algunas de las soluciones o mejoras propuestas en esta sección, son ejecutadas mediante el uso de distintos *script* desarrollados en el lenguaje SQL. La meta es poder ejecutar estos *scripts* sobre una instancia de la base de datos, como pasos que van solucionando distintos problemas. Cada uno lleva la base de datos a un estado preparado para poder ejecutar el siguiente. De esta forma, el proceso puede ser fácilmente adaptado y reproducido en etapas bien definidas sobre distintas instancias de la base de datos.

Para agregar ciertas funcionalidades que surgen a partir del análisis desarrollado en este Capítulo es necesario realizar una migración del gestor de base de datos utilizado, esto es explicado en detalle en el Capitulo 4.

Por otro lado, la complejidad de implementar varias de las soluciones o mejoras expuestas, hace inviable su aplicación sobre esta versión de la base de datos lo que lleva a la necesidad de un rediseño completo de esta que se expone en el Capítulo 5.

3.2.1. Tablas y Vistas Obsoletas

Como comienzo del relevamiento, se hizo un recorrido general por toda la base de datos. En el cual, se localizaron veintisiete tablas y doce vistas que eran obsoletas, ya sea porque se habían dejado de utilizar o sirvieron para hacer

pruebas y nunca fueron eliminadas.

Como consecuencia de este paso se creó un *script* que borra estas tablas y vistas, el cual es ejecutado antes del paso de ingeniería inversa descrito en la primera sección de este capítulo. Por esta razón estas tablas no están ilustradas en la Figura 3.1.

3.2.2. Falta de Relaciones Explícitas Mediante el uso de Clave Foránea

Los atributos de algunas tablas se utilizan para contener valores que identifican filas de otras tablas. Por ejemplo: en la Figura 3.1 se puede observar la tabla `af_caso` que almacena información de las distintas víctimas. Cada fila representa una víctima y utiliza de identificador al campo `idcaso`. En diversas tablas como `af_fotos` o `af_militancias` se encuentra un atributo también llamado `idcaso` el cual sirve para denotar que cada registro de dichas tablas esta relacionado con registros de la tabla `af_caso`.

Como se explicó en la Sección 2.1.3, este tipo de relación se puede establecer explícitamente mediante el uso de clave foránea. Esto evita que haya información inconsistente debido a registros huérfanos ya que, volviendo al ejemplo, no tendría sentido un registro en la tabla `af_militancias` si no se sabe a qué persona corresponde. Además, cuando se agrega una clave foránea se puede establecer lo que se debe hacer con los registros en caso de que el atributo al que se hace referencia sea borrado o actualizado. Las opciones son: borrar el registro o cambiar a `NULL` el valor del atributo al que se le agregó la clave foránea para que el registro se siga manteniendo. Esto facilita el manejo de la base, ya que el borrado de un registro en una tabla padre se propaga en las tablas desde las cuales se hace referencia.

Para poder implementar estas claves foráneas fue necesario alcanzar la integridad referencial. Para ello se tuvo que hacer un análisis y consecuente modificación de sus datos y estructura, ya que había una gran cantidad de casos de registros huérfanos y columnas que utilizaban distintos tipos de datos que las columnas referenciadas.

Como producto de esta etapa se desarrolló un *script* SQL que modifica datos y tipos de datos de columnas para conseguir la integridad referencial. Luego, implementa las claves foráneas de las relaciones implícitas que fueron detectadas. En total se agregaron 67 nuevas claves foráneas, llevando a un total de 91 las utilizadas en la base de datos.

En la figura 3.2 se puede observar el resultado de realizar el proceso automático de ingeniería inversa sobre la versión de la base de datos a la cual se implementaron estas claves foráneas.

3.2.3. Relaciones Implícitas Entre Múltiples Tablas

En este punto se pudo apreciar que el diseño conceptual de la base de datos se desarrolla en torno a siete entidades conceptuales principales, declaradas explícitamente en los registros de la tabla `objetos` (Tabla 3.1) que cuenta con un campo numérico que ejerce de identificador y otro de texto para almacenar el nombre que describe a cada una.

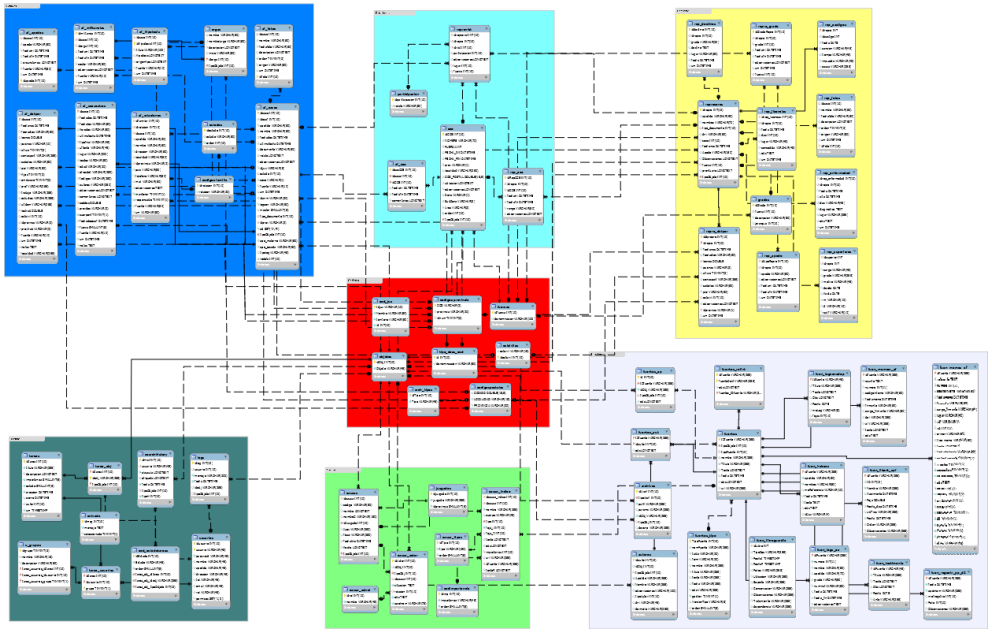

Figura 3.2: Diagrama Entidad Relación de la base de datos PRESENTES donde se muestran todas las tablas y relaciones entre ellas implementadas mediante el uso de claves foráneas.

idObj	Objeto
1	Victimas
2	Represores
3	Causas
4	Fuentes
5	Organizaciones
6	CCD
31	Acto Procesal

Cuadro 3.1: Registros de la tabla `objetos` de la base de datos de Presentes.

Cada una de estas entidades conceptuales esta representada por una tabla en la base de datos según indica la siguiente lista:

- Victimas → `af_casos`
- Represores → `represores`
- Causas → `causas`
- Fuentes → `fuentes`
- Organizaciones → `orgas`
- CCD[3] → `cce`

[3]CDD son las iniciales de Centro Clandestino de Detención

- Acto Procesal → `causa_indice`

Conforme se puede observar en la Figura 3.2 como resultado de la etapa anterior, estas entidades quedan relacionadas entre sí por medio de claves foráneas a tablas que almacenan información sobre dichas relaciones. Como es el caso de `af_cce` que vincula víctimas con centros clandestinos de detención. También se puede contemplar la relación entre actos procesales y causas, implementada directamente mediante el uso del atributo `idcausa` en la tabla `causa_indice`.

Sin embargo existen relaciones que no fueron posibles de implementar mediante el uso de claves foráneas. Una de estas se encuentra en la tabla `ap_fuentes` (Fig. 3.3) que relaciona Fuentes con cualquiera de las siete entidades conceptuales antes mencionadas. Para ello, la tabla cuenta con el atributo `tipoObjeto`[4] que indica la entidad según la clasificación dada por la tabla `objetos` con que está relacionado cada registro y el atributo `idObj`[5] que contiene el identificador del registro en la tabla que representa a esta entidad. De esta forma un registro en la tabla `ap_fuentes` que contenga el valor `3` en el atributo `tipoObjeto` y `20` en `idObj` indica que ese registro está vinculado al registro en la tabla `causas` con `idcausa` igual a `20`.

Los identificadores de las tablas a las que hace referencia el campo `idObj` de la tabla `ap_fuentes` son independientes, por lo que pueden contener valores que se repiten. Consecuentemente resulta complejo implementar una relación explícita entre este campo y las distintas tablas a las que hace referencia.

Otro de los inconvenientes que acarrea esta relación es que los identificadores de las tablas que representan a las entidades son de tipo numérico, salvo fuentes que utiliza una cadena de caracteres. Esto obliga a que los identificadores numéricos sean almacenados en `idObj` como cadena de caracteres.

Como solución a este problema se aplicó en el nuevo diseño expuesto en la Sección 5.1 la técnica de Mapeo Objeto Relacional descrita en la Sección 2.3.

3.2.4. Uso de Tablas Específicas para Cada Tipo de Documento

La entidad "Fuente" representa a las distintas fuentes documentales concernientes a la dictadura militar en Córdoba que almacena el sistema, como por ejemplo: documentos legales, diarios, etc. Está entidad está representada por la tabla `fuentes` (Fig. 3.3) que además de almacenar información como su título y fecha, incluye el campo `texto` donde se guarda la concatenación de las distintas partes que componen al documento.

A su vez, estos mismos documentos están almacenados en distintas tablas auxiliares que representan a cada tipo de documento en particular. Estas tablas, se ocupan de estructurar la información empleando atributos específicos dependiendo del tipo de documento que guarda. Cada registro representa un documento y está relacionado mediante clave foránea a su versión en la tabla `fuentes`.

[4] Notar que `ap_fuentes.tipoObjeto` contiene los valores de `objeto.idObj`.

[5] Notar que por más que el nombre de los atributos `ap_fuentes.idObj` y `objeto.idObj` sean iguales representan distinta información.

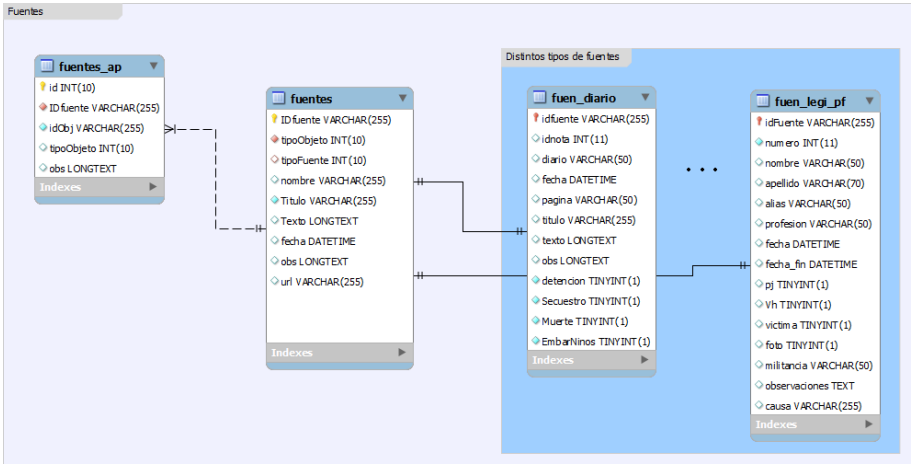

Figura 3.3: Extracto de tablas relacionadas con Fuentes de la Figura 3.2.

Cuando se desea almacenar un documento, es necesario hacerlo en la tabla específica del documento y recopilar esos mismos datos en la tabla `fuentes`. Además, si se desea comenzar a almacenar un nuevo tipo de documento, es necesario implementar una nueva tabla con los distintos campos específicos que lo componen y referenciarla a la tabla `fuentes`.

En un primer momento, para esta parte del sistema se pensó en utilizar la base de datos documental MongoDB, pero dada la complejidad que requiere su implementación y adaptación, y para evitar que la base de datos está dividida en dos gestores de base de datos distintos, se decidió emplear en el nuevo diseño (Sec. 5.3) el tipo de dato `jsonb` nativo de PostgreSQL. Como se explicó en la Sección 2.4, este tipo de dato permite definir campos arbitrarios, lo cual posibilita que en un solo atributo se pueda almacenar un valor que abarque los campos específicos de cualquier documento.

3.2.5. Tablas Utilizadas para Restringir Valores de Atributos

Las tablas del grupo "Codigos" (Fig. 3.2) son utilizadas para restringir posibles valores de atributos en otras tablas. Por ejemplo, los registros de la tabla `estciviles` (Caudro 3.2) son utilizados por las tablas `af_datper` y `repre_datper` las cuales almacenan datos personales de Víctimas y Represores respectivamente. Ambas tienen un atributo llamado `estcivil` el cual almacena un valor numérico que hace referencia al identificador `estadocivil.idestcivil`. De esta forma, se restringen los posibles valores de los atributos `af_datper.estcivil` y `repre_datper.estcivil` a los almacenados en `estadocivil.estcivil`.

idestcivil	estcivil
1	Casado/a
2	Soltero/a
3	En pareja
4	Viudo/a
5	Separado/a

Cuadro 3.2: Registros de la tabla `estciviles`.

En algunos casos, como se puede ver en la Tabla 3.3 `codimportancia`, se cuentan con una columna adicional que almacena valores que cumplen la función de asignar un orden a los registros de la tabla.

idimp	importancia	orden
1	Sin importancia	50
2	Poca Importancia	40
3	Normal	30
4	Importante	20
5	Muy Importante	10

Cuadro 3.3: Registros de la tabla `codimportancia`.

Este método de restringir y ordenar los posibles valores de algunas columnas, implica (como se puede observar en la Figura 3.2) la utilización de una gran cantidad de tablas y claves foráneas, que se traduce en una base de datos más compleja. Además, no agrega información ni funcionalidad relevante para el propósito del sistema.

En lugar del uso de este tipo de tablas, se optó por la utilización de "Dominios Definidos por el Usuario" para reemplazar aquellas que simplemente listan posibles valores y "Tipos de Datos Enumerados" para las que además tienen definido un orden. En total son eliminadas catorce tablas y veintisiete claves foráneas utilizadas por este sistema, este cambio es desarrollado en la Sección 5.2 donde se explica en detalle estos conceptos.

3.2.6. Dominio de los Atributos

En la Sección 3.2.2 antes de poder agregar relaciones explícitas entre diversas tablas, fue necesario cambiar el tipo de datos de varias columnas para que fueran del mismo tipo que el de las columnas a las que se determinó referenciar mediante el uso de claves foráneas. También vimos en la Sección 3.2.4, que en la tabla `fuentes_ap` se utiliza el campo `idObj` de tipo texto para almacenar valores numéricos convertidos a texto. Estos y otros casos descubiertos en etapas posteriores, presentan una incorrecta utilización o elección de los dominios de los atributos.

Una errónea asignación de dominios a las columnas no solo genera problemas como los mencionados sino que implica una débil estructuración de la información que hace a la base de datos (como se muestra en el Capítulo 6) propensa a permitir almacenar datos inconsistentes, información redundante, etc.

La mayoría de estos casos son solucionados en esta etapa simplemente modificando el tipo de dato del atributo. El resto, como es el caso de `fuentes_ap.idObj` y

otras tablas que utilizaban de la misma forma un atributo `idObj` son resueltos en las siguientes etapas.

3.2.7. Tablas, Atributos e Información Redundante

Para almacenar los datos personales de las Víctimas se utilizan las tablas `af_datper`, `af_apodos`, `af_fotos` y `af_casos`. Análogamente para los Represores se usan `repre_datper`, `rep_apodo`, `rep_fotos` y `represores`. Ambas entidades almacenan los mismos datos personales (salvo unos pocos atributos), estructurada de la misma forma. También la base de datos cuenta con otras tablas que almacenan datos de personas relacionados con causas judiciales (Tabla `causa_actor`), autores de los documentos (Tabla `autores`) y individuos relacionados con víctimas (Tabla `af_relaciones`). Una persona puede cumplir con más de uno de estos roles por lo que sus datos pueden verse repetidos en estas tablas.

En la Sección 5.4 se presenta una mejora a este diseño, en la cual todos los datos de las personas están almacenadas en una única tabla. Esto disminuye la cantidad de tablas y claves foráneas utilizadas, simplificando el diseño y permitiendo relacionar a los distintas personas representadas en la base, de una forma más clara y sencilla.

Por otra parte como se mostró anteriormente en la Sección 3.2.4 existe una redundancia de información de Fuentes, dado que los mismos documentos están almacenados tanto en la tabla `fuentes` como en las tablas específicas de cada tipo de documento.

3.2.8. Múltiples Criterios para la Asignación de Nombres de Tablas y Columnas

En el desarrollo del sistema no se siguió una convención de nombres bien definido para tablas y columnas. Ej: el nombre de la tabla `cod_juz` es la concatenación de dos palabras utilizando un guión bajo, distinto del de la tabla `codigo familia`, el cual utiliza un espacio; o `codigospostales`, que simplemente las une. También se puede observar que los nombres de las columnas están escritos algunos completamente en mayúsculas, otros en minúsculas o la primera letra de cada palabra concatenada en mayúscula o minúscula.

Algunos nombres están registrados en plural y otros en singular. Además, se puede observar distintas abreviaciones para un mismo nombre. Ej: `repre_datper` y `rep_fotos` para referirse a tablas vinculadas con Represores.

También se puede apreciar que se usan columnas con un mismo nombre que almacenan distinto tipo de información o con distinto nombre y que guardan el mismo tipo de información.

La falta de un estilo para la elección de nombres, dificulta la comprensión del código y su mantenimiento. Por este motivo se propuso en el rediseño de la base de datos (Sec. 5.5) una mejora especificando reglas para dar un estilo de programación y convención de nombres bien definidos.

3.3. Diagrama Conceptual de la Base de Datos de Presentes

Como resultado de esta etapa se llegó a comprender la mayor parte del funcionamiento de la base de datos y su estructura, lo suficiente para poder seguir con las demás etapas en las cuales de a poco se va vislumbrando con más detalle.

La Figura 3.4 muestra un esquema reducido del diseño de la base de datos. Ahí se puede observar claramente las relaciones y tablas de los grupos conceptuales a excepción de los Códigos, dado que este no hace un aporte sustancial al diseño conceptual y cuenta con varias tablas y un gran porcentaje de las claves foráneas que utiliza la base de datos.

Si bien de esta forma se ve un diseño más claro existe una gran cantidad de relaciones entre distintas tablas que, dado la manera en que se implementaron, no se ven reflejadas en este diseño. Más bien pertenecen a un diseño conceptual subyacente que no se desarrolló correctamente como es el caso de las relaciones entre múltiples tablas (Sec. 3.2.3).

Los problemas expuestos en la sección anterior sumados a la gran cantidad de tablas y relaciones entre los datos hacen de esta, una base de datos más compleja de lo que debería ser. Por lo que el objetivo de los siguientes avances están orientados a ordenar de una forma más estructurada la información, disminuir la cantidad de tablas utilizadas y mejorar la calidad de los datos, todo esto a medida que se agregan características que amplían su funcionalidad.

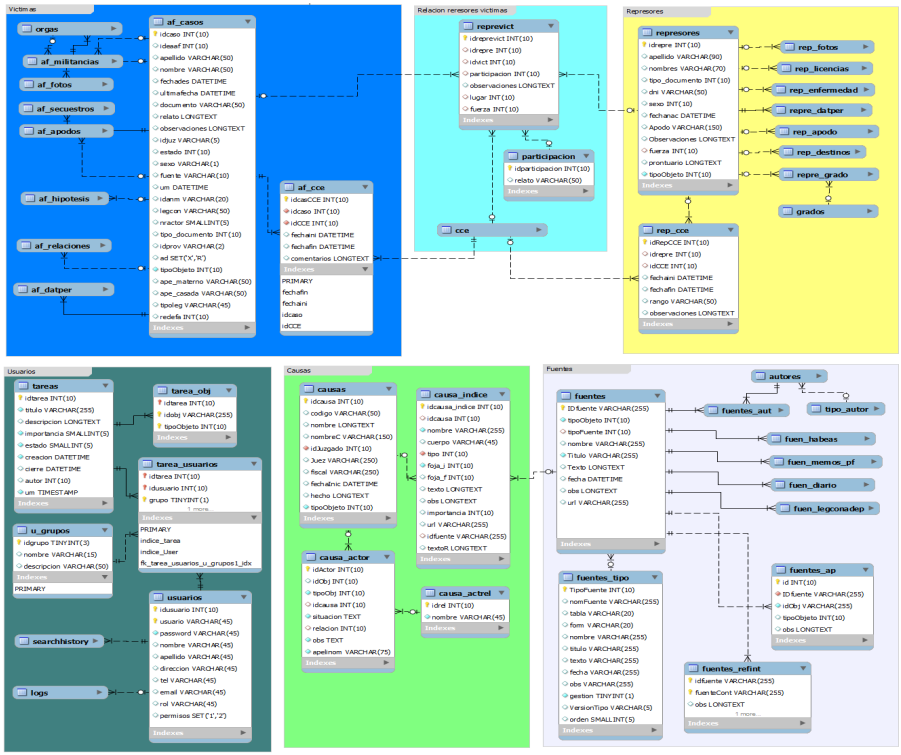

Figura 3.4: Diagrama de la base de datos de PRESENTES donde solo se muestran los grupos, tablas y relaciones más relevantes.

Capítulo 4

Migración del Gestor de Base de Datos

Una vez hecho el análisis y superados los inconvenientes iniciales, el siguiente paso fue migrar Presentes del gestor de base de datos relacional MySQL al gestor de base de datos objeto relacional PostgreSQL.

En este capítulo se describe el motivo de esta migración, el proceso para llevarla a cabo y por último, mejoras que se agregan usando características que ofrece el nuevo gestor utilizado.

4.1. Razones para Migrar de Gestor de Base de Datos

Para almacenar, manejar y recuperar datos de las tablas, MySQL permite seleccionar entre distintos motores de almacenamiento. Estos mecanismos presentan diferentes características en cuanto a su funcionalidad y rendimiento. El gestor permite definir que motor es empleado para gestionar cada tabla, de esta forma podemos utilizar la que mejor se adecue a los requerimientos particulares de las mismas.

Los principales motores de almacenamiento de MySQL son: MyISAM y InnoDB [Ora14]. Estos cuentan con distintas características, la que más nos concierne es que InnoDB permite la utilización de claves foráneas y MyISAM no. A su vez, a diferencia de InnoDB, MyISAM permite utilizar Full Text Search que es una herramienta muy útil para realizar búsquedas en textos de gran tamaño como es el caso de los documentos que reúne Presentes. Esta es la principal razón que motivó la decisión de migrar de gestor, ya que si se desea utilizar FTS en MyISAM se debe renunciar al uso de claves foráneas en las tablas que lo apliquen.

A diferencia de MySQL, PostgreSQL no utiliza este sistema de motores de almacenamiento, por lo que las características y funcionalidades son las mismas para todas las tablas. Además, la implementación de búsqueda de texto completo provista de forma nativa por PostgreSQL tiene mejores prestaciones y permite una gran variedad de ajustes para adaptar la herramienta a las particularidades de las búsquedas requeridas.

Si bien esta fue la principal razón para plantear una migración de gestor de base de datos, también se analizaron otros aspectos pensando en futuras mejoras y que entorno se adapta mejor a los requerimientos actuales del sistema. En general se puede apreciar que MySQL es más simple de utilizar, ya que no se rige de forma tan estricta a SQL como PostgreSQL. Este a su vez cuenta con un importante desarrollo en materia de geolocalización, mediante el módulo PostGIS. También permite de forma nativa la utilización de XML y JSON, como tipo de datos de los atributos para almacenar información semiestructurada.

Estas y otras ventajas no mencionadas son razones suficientes para determinar que debe llevarse a cabo la migración del gestor de la base de datos usado por Presentes.

4.2. Metodología utilizada

El proceso de migración se diseñó siguiendo el objetivo de poder aplicar cambios paso a paso, a partir de distintos *script* ejecutados sobre la instancia de la base de datos que da como resultado el aplicar lo desarrollado en la primer etapa. De esta forma, en la instancia de la base que utiliza el APM se puede primero aplicar las soluciones de la etapa anterior y luego migrar a PostgreSQL, empleando en un orden establecido estos *scripts*.

Para cumplir con una clave foránea, el valor que es ingresado en la columna que tiene esta restricción, primero tiene que estar en algún registro en el atributo al que se hace referencia en la tabla padre. Esto implica que los datos deben ser ingresados en una secuencia determinada. Para obviar este trabajo y a la vez cumplir con el objetivo propuesto en el comienzo de esta sección, se dividió el proceso de migración en los siguientes tres pasos:

Paso 1: Crear una versión en PostgreSQL de las tablas utilizadas por Presentes en MySQL. Esta versión, no incluye índices, restricciones como claves primarias o foráneas, etc. solo se crean las tablas y sus columnas.

Paso 2: Extraer todos registros de las tablas de Presentes almacenadas en el gestor MySQL e introducirlos en la versión de PostgreSQL que da como resultado el paso 1.

Paso 3: Agregar a la versión en PostgreSQL las claves primarias y foráneas, índices, etc. que utiliza la base de datos.

Para el primer y tercer paso se utilizó pgAdmin III[1] que es una herramienta de código abierto para la administración y desarrollo de bases de datos PostgreSQL. Este software tiene una herramienta que proporciona una forma sencilla de crear elementos de la base de datos, mediante formularios que generan el código SQL para crear dichos elementos. Luego el código es ejecutado en la base de datos. Con este mecanismo se creó una versión de Presentes en PostgreSQL con las mismas tablas, columnas[2], claves foráneas e índices que la versión en MySQL pero sin datos.

[1] http://www.pgadmin.org/

[2] Si bien los nombres de las columnas son los mismos, los dominios que se les asignan son distintos ya que los tipos de datos que presentan los gestores difieren en su implementación y sintaxis.

Lo siguiente fue hacer un volcado de esta versión a un archivo que contiene las sentencias SQL, para crear la base completa. Luego fue dividido en dos, uno que crea sólo las tablas con sus columnas y otro que contiene el resto (claves foráneas y primarias, índices, etc.). Estos *scripts* son los utilizados para realizar los pasos uno y tres, respectivamente.

Para efectuar el paso dos se hace un volcado en un archivo de sólo los datos de la versión de Presentes en MySQL. Posteriormente sobre ese archivo se ejecuta `mysql2pgsql`[3], un *script* desarrollado con el lenguaje Perl, que transforma la sintaxis SQL utilizada por MySQL a una aceptada por PostgreSQL. De esta forma se obtiene un nuevo archivo SQL que, con unos pocos cambios, es capaz de insertar los datos en la versión de Presentes en PostgreSQL.

4.3. Problemas que se Presentaron en la Migración de Gestor de Base de Datos

Realizar esta migración fue un trabajo más complicado de lo que aparentaba en un primer momento. La gran cantidad de tablas, relaciones entre ellas, datos de gran tamaño y las diferencias entre los gestores generaron muchas complicaciones. A esto se suma la necesidad de poder realizar este proceso en otras instancias de la base de datos.

En la siguiente sección se hace un recorrido por los inconvenientes más importantes que presenta el desarrollo del proceso de migración.

4.3.1. Constantes Modificaciones en Presente

Mientras realizamos el trabajo la base de datos utilizada en el APM era constantemente modificada, ya sea con columnas agregadas a tablas ya existentes o con el desarrollo de nuevas tablas (principalmente para almacenar nuevos tipos de documentos).

Por este motivo fue necesario ir actualizando los *scripts* de los pasos uno y tres a medida que ocurrían estas modificaciones y así poder realizar la migración desde la instancia de la base utilizada en ese momento por el APM.

4.3.2. Asignación Automática de Valores para Claves Primarias

Los campos de enteros que son utilizados como clave primaria subrogada en Presentes emplean la cláusula `AUTO_INCREMENT`, que se declara junto con el tipo de dato de la columna. Lo que hace que cada vez que se ingresa un nuevo registro a la tabla no sea necesario especificar el valor de esta columna, a la cual automáticamente se le asigna el siguiente del mayor de los valores que tiene la columna en ese momento. De esta forma el gestor de la base es el encargado de administrar los valores de algunas claves primarias.

PostgreSQL no cuenta con ese mecanismo, pero se pueden utilizar las secuencias. Estas son tablas con un campo numérico que almacena un valor y cada vez que se hace una consulta a esta tabla el valor es incrementado para la próxima consulta. De esta forma se puede declarar que la columna que cumple con la función de clave primaria se le asigna automáticamente el valor que devuelve la consulta a la secuencia.

[3] https://github.com/ahammond/mysql2pgsql/blob/master/mysql2pgsql.pl

4.3.3. Caracteres no Reconocidos y Textos de Gran Tamaño:

Varios de los textos almacenados en las tablas `fuentes` fueron generados a partir de imágenes de documentos escaneados, utilizando una herramienta conocida como Optical Character Recognition (OCR). Algunos de los caracteres de estos documentos digitalizados no fueron bien reconocidos por esta herramienta, lo que causó que se almacenarán caracteres con un mal formato en su codificación. Al utilizar `mysql2pgsql` para realizar el paso dos, aparecían errores ocasionados por las sentencias que insertan los registros que tienen estos caracteres.

En otros casos `mysql2pgsql` fallaba al ejecutarse debido a al gran tamaño de algunos registros que llegaban a superar los 4MB. Dado que eran pocas las filas que causaban estos problemas, se decidió que lo conveniente era escribir las consultas SQL que insertan estos registros en la versión de Presentes en PostgreSQL.

4.3.4. Cambios en la Base de Datos en MySQL Antes de Migrar Los Datos:

Como ya se comento anteriormente, PostgreSQL interpreta de una forma más estricta el estándar SQL. Por lo que para poder implementar algunas restricciones de claves primarias y foráneas que tiene Presentes, fue necesario realizar algunos cambios.

Por ejemplo en PostgreSQL para declarar una clave foránea, ambas columnas deben exactamente el mismo dominio y la columna referenciada debe tener una restricción que haga que sus valores sean únicos para cada registro de la tabla. Por ello, en algunas columnas se tuvo que agregar la restricción de unicidad y cambiar el tipo de dato. Estas modificaciones se hicieron en la versión de Presentes en MySQL, antes de realizar la migración de datos, para así detectar registros que no cumplían con estos requisitos.

4.4. Problemas que se Presentaron para Adaptar la Interfaz a la Base de Datos Implementada en PostgreSQL

Si bien no nos encargamos de adaptar la interfaz de Presentes a la base de datos implementada en PostgreSQL, se prestó ayuda para ajustar la sintaxis SQL de las consultas más complejas. Además se adecuaron funciones y características específicas del gestor MySQL que no pertenecen al estándar SQL, utilizando funciones equivalentes provistas por PostgreSQL.

Cuando se intento realizar esta adaptación se presentaron una serie de dificultades, a continuación se describen las más importantes.

4.4.1. Búsquedas Insensibles a Mayúsculas y Acentos

Por defecto, las búsquedas que se pueden realizar en MySQL no son sensibles a los acentos ni a las mayúsculas. Diferente es el caso de PostgreSQL, que es sensible tanto a los acentos y como a las mayúsculas[4].

[4]Nota: PostgreSQL provee el operador `ILIKE` que es insensible a mayúsculas pero fue un requerimiento del APM que no se cambiaran los operadores SQL.

Si bien en nuestro idioma un acento puede cambiar la semántica de una palabra, en este caso es deseable que una consulta devuelve todas las formas de la palabra buscada ya que puede estar mal escrita y no tener su correspondiente acento o simplemente ser una variación gramatical de esta.

Para que las búsquedas sean insensibles a los acentos se usó la función provista por PostgreSQL `unaccent()`[5] que remueve los signos diacríticos[6] de cada palabra.

También es necesario que las consultas no sean sensibles a mayúsculas. Para cumplir con este objetivo se utilizó el tipo de datos `citext`[7] que es similar al tipo de datos nativo de PostgreSQL `text` con la diferencia que es insensible a mayúsculas.

Las consultas deben ser a la vez insensibles tanto para los acentos como para las mayúsculas. Para lograr esto se desarrollaron funciones que pueden tomar palabras de los términos de búsqueda o columnas completas donde se desea realizar la consulta. Estas funciones hacen un cambio de tipo de datos de estas palabras a `citext` y se lo pasa como parámetro a la función `unaccent()`, de esta forma las funciones devuelve un texto sin acentos de tipo `citext` listo para ser utilizado en cualquier consulta.

Este método no presenta un buen rendimiento en cuanto a la velocidad para procesar consultas, ya que a cada palabra se le aplica el proceso mencionado, solo se utiliza en campos de textos pequeños por lo que no presenta una diferencia de tiempo perceptible. Para las fuentes que contienen los textos de mayor tamaño no es necesario utilizar estas funciones, ya que las búsquedas se realizan utilizando Full Text Search que provee búsquedas insensible a mayúsculas y acentos mediante los diccionarios que utiliza.

4.4.2. Diferencias de Sintaxis SQL entre MySQL y PostgreSQL

La sintaxis que utiliza MySQL para interpretar sentencias SQL tiene varias diferencias con respectos PostgreSQL. Principalmente debido a las distintas formas de utilizar comillas para escribir valores de diferentes tipos de datos o nombres de tablas, columnas, etc.

En PostgreSQL cuando se utilizan tablas y columnas que contienen mayúsculas en sus nombres, deben ir encerrados entre comillas dobles. Esto acarreaba un gran trabajo para adaptar la interfaz a la base de datos migrada, ya que se debía buscar en el código fuente los lugares donde se los utilizó y agregarle comillas. Por este motivo el responsable del sistema requirió que se pasaran los nombres de todas las tablas y atributos a minúscula.

Solucionar este problema fue sencillo ya que ambos gestores son insensibles a mayúsculas en la interpretación de las sentencias SQL. Aprovechando este hecho, se transformó a minúscula el contenido completo de los *scripts* de los pasos uno y tres del proceso de migración y en el *script* del paso dos se cambió a minúscula solo los nombres de las tablas que contenían alguna letra mayúscula.

[5] La función `unaccent()` es parte del diccionario `unaccent`. El concepto de diccionarios en PostgreSQL es explicado más adelante en la Sección 4.5.1.

[6] Un signo diacrítico es un signo gráfico que confiere a los signos escritos (no necesariamente letras) un valor especial. Fuente: http://es.wikipedia.org/wiki/Signo_diacr%C3%ADtico

[7] Si bien `citext` no pertenece al núcleo de PostgreSQL esta incluido como módulo en el paquete `contrib` que es parte de la distribución del gestor.

4.4.3. Funciones Específicas del Gestor MySQL

En las consultas que genera la interfaz, constantemente se encuentran funciones propias de MySQL que no pertenecen al lenguaje de consulta SQL, por ejemplo: concatenar cadenas de caracteres, cambiar palabras a mayúscula o minúscula, escribir el resultado de la consulta con un formato específico, etc. Cada una de estas funciones es reemplazada por una equivalente provista por PostgreSQL.

4.5. Mejoras Aplicadas a la Base de Datos en PostreSQL

Luego de concretar la migración se introdujeron dos mejoras a la base de datos. La primera consiste en implementar Full Text Search sobre campos de las tablas de Fuentes y Causas. La siguiente agrega un sistema que lleva un registro meticuloso de los datos que son insertados, borrados o modificados en la base de datos.

4.5.1. Full Text Search en PostgreSQL

Las tablas que almacenan los distintos tipos de documentos contienen registros con textos de gran tamaño. Una manera de mejorar las búsquedas en estos documentos es emplear la herramienta Full Text Search (FTS) que identifica de mejor forma (a otras herramientas del gestor) las palabras y sus derivaciones en las consultas. También agrega otras características como una mayor rapidez o la posibilidad de ordenar los resultados según su relevancia.

Para explicar la utilización de FTS se definen a continuación algunos términos y conceptos [Gro14]:

Documento En este contexto un documento es la unidad de búsqueda del sistema FTS. Es normalmente el valor de un campo de texto de una tabla o la concatenación de valores obtenidos de varios campos de una o más tablas.

Token Las palabras de los documentos son clasificadas según su tipo en distintas clases de *tokens*, por ejemplo se suelen utilizar como clase de *token*: números, palabras, direcciones de mail, url, etc. PostgreSQL provee un conjunto predefinido de estas clases, pero si se requiere para una aplicación específica se pueden definir nuevas.

Lexema: A diferencia de su definición gramatical, un lexema en PostgreSQL no necesariamente debe tener un significado léxico. La idea es que diferentes formas de una palabra están representadas sólo por una, el lexema. Para ello se les aplica un proceso de normalización que por ejemplo puede incluir el cambio de mayúsculas por minúsculas, remover sufijos como “s” o “es”, etc.

Diccionario: Los diccionarios son programas utilizados para transformar palabras en lexemas y eliminar aquellas que no deben ser consideradas en las búsquedas[8].

[8] Las palabras que no se utilizan en las búsquedas son denominas *stop words*.

PostgreSQL provee diccionarios predefinidos y plantillas que pueden ser utilizadas para crear nuevos diccionarios con parámetros personalizados. A continuación se describen los principales tipos de diccionarios utilizados:

***Simple*:** Cambia las letras de todas las palabras a minúscula.

***Synonym*:** Reemplaza cada palabra con un sus sinónimos. Opcionalmente se puede dejar la palabra original.

***Thesaurus*:** Es una extensión del diccionario de sinónimos que agrega la posibilidad de reemplazar palabras por frases.

***Ispell*:** Utiliza diccionarios morfológicos, que pueden normalizar diferentes formas lingüísticas de una palabra a un lexema. Por ejemplo, se puede coincidir las conjugaciones del término de búsqueda "secuestro" como: secuestrar, secuestrando, secuestrado, etc. La distribución estándar de PostgreSQL no incluye un archivo de diccionario morfológico pero se pueden utilizar los desarrollados para OpenOffice[9].

***Snowball*:** Este diccionario utiliza el algoritmo *Stemming* inventado por Martin F. Porter [Por80] que elimina las formas plurales, tiempos verbales, etc. Por ejemplo las palabras secuestro, secuestrar, secuestrando y secuestrado son normalizadas a "secuestr". Aunque de esta forma se pierde el sentido gramatical de las palabras, se conserva la relación de similitud entre ellas.

tsvector: Un valor `tsvector` es una lista almacenada de distintos lexemas. Opcionalmente se puede agregar un número a cada lexema que indica la ubicación de las palabras de origen en el documento.

También se puede agregar a los lexemas una marca de peso[10] que es utilizada para reflejar la estructura del documento. Por ejemplo: marcando palabras que están en el título de diferente forma a las que pertenecen al cuerpo del documento. Se permite además definir distintas prioridades a las marcas de peso para que las consultas den mayor o menor relevancia dependiendo a qué marca de peso pertenece el lexema.

Para normalizar un documento se puede utilizar la función `to_tsvector()` que devuelve un valor de tipo `tsvector` con los palabras normalizadas a lexemas. Por ejemplo: si utilizamos la función sobre el siguiente documento "Detuvieron a un dirigente gremial y a un profesional" agregando la marca de peso A, devuelve el siguiente valor de tipo tsvector:

```
'detuv':1A 'dirigent':4A 'gremial':5A 'profesional':9A
```

tsquery: Es un tipo de dato cuyos valores son utilizados para realizar consultas. Estos contienen lexemas que pueden ser combinados utilizando los operadores booleanos & (conjunción), | (disyunción) y ! (negación).

La función `to_tsquery` es utilizada para convertir texto a un valor `tsquery`. Por ejemplo: si deseamos crear un término de búsqueda que encuentre documentos que están relacionados con dirigentes o profesionales se puede utilizar esta función sobre el texto "dirigentes | profesionales", lo cual devuelve el siguiente valor `tsquery`:

```
'dirigent' | 'profesional'
```

[9] http://extensions.openoffice.org/en/project/diccionario-espanol-argentina-sinonimos-y-separacion-silabica
[10] Las marcas de peso pueden ser A, B, C, o D

4.5.1.1. Proceso de Normalización de Documentos

Como primer paso para el proceso de normalización de un documento se ejecuta sobre este un *parser* (provisto por PostgreSQL), que identifican los límites de las posibles palabras y las divide en los distintos tipos de *token* según su clase[11]. Es posible que se produzca una superposición de *tokens* de la misma parte de texto, esto permite que las búsquedas trabajan tanto para la palabra compuesta como para sus partes.

Luego, el arreglo de *tokens* resultante es procesado utilizando una lista de diccionarios. Cada diccionario es consultado por turnos hasta que alguno lo reconoce. Si es identificado como *stop word* o si ningún diccionario lo reconoce, el *token* será desechado y no se utilizará en las búsquedas.

Sobre cada *token* un diccionario puede devolver:

- Un arreglo de lexemas, si el *token* es reconocido por el diccionario.
- Un lexema con el indicador `TSL_FILTER`, que reemplaza el *token* original con uno nuevo que es pasado a los siguientes diccionarios de la lista. [12]
- Un arreglo vacío, si el diccionario reconoce el *token* como *stop word*.
- NULL, si el diccionario no reconoce al *token* de entrada.

El resultado de este proceso es un valor de tipo `tsvector` que contiene lexemas de las palabras que componen al documento. Sobre este valor se pueden realizar consultas FTS utilizando como términos de búsqueda valores de tipo `tsquery`.

4.5.1.2. Implementación de FTS en Presentes

Hay diversas formas de implementar FTS en PostgreSQL que presentan distintas ventajas. Dado que los documentos que almacena Presentes no son modificados habitualmente, se optó por guardar los documentos normalizados en una columna. Esto tiene la ventaja de que las consultas sean muy rápidas ya que no se tiene que normalizar constantemente los documentos y permite la utilización de índices[13].

Para implementar FTS en Presentes se agregó a las tablas que almacenan documentos, una columna de tipo `tsvector` de la siguiente forma:

```
ALTER TABLE fuentes ADD COLUMN busqueda_texto tsvector;
```

Luego se utilizó la función `to_tsvector()` para rellenar esta columna con la concatenación de los `tsvector` generados a partir de cada campo de texto sobre los que se desea habilitar la búsqueda FTS como se muestra a continuación:

```
UPDATE fuentes SET busqueda_texto =
  setweight(to_tsvector('spanish', coalesce("Titulo",'')), 'A') ||
  setweight(to_tsvector('spanish', coalesce("nombre",'')), 'B') ||
```

[11] El conjunto de posibles tipos de *tokens* es definido por el *parser*.

[12] Este tipo de diccionario es denominado "Diccionario Filtro"

[13] PostgreSQL provee dos tipos de índices específicos para columnas de tipo `tsvector`: GiST y GIN.

```
setweight(to_tsvector('spanish', coalesce("Texto",'' )), 'C') ||
setweight(to_tsvector('spanish', coalesce("obs",'')), 'D');
```

La opción `'spanish'` indica que se debe utilizar el *parser* y diccionarios predefinidos por PostgreSQL para textos en español. Aquí también se agregaron marcas de peso, mediante la función `setweight` para saber en qué campo se encontró el término buscado y establece un orden de relevancia para mostrar los resultados de las consultas. La función `coalesce` es utilizada para asegurarse que un campo seguirá siendo concatenado aun cuando otro sea `NULL`.

Para mantener actualizado este campo se crea un `TRIGGER` que ejecuta una función que realizar nuevamente el UPDATE de la columna `busqueda_texto` en un registro si alguno de los atributos: `Titulo`, `nombre`, `Texto` o `obs` es modificado.

4.5.2. Historial de Cambios en Registros de Tablas en PostgreSQL

La siguiente mejora que fue introducida en esta etapa es un historial de la información de la base de datos que es agregada, modificada o borrada por los distintos usuarios del sistema. Para ello se utilizó Audit trigger 91plus[14], que es un sistema de auditoria que crea en una base de datos paralela con funciones y `TRIGGERS` que permite llevar un detallado registro de los cambios que se realizan en los datos.

Para que las modificaciones en los datos de una tabla sean registrados por este módulo se debe especificar mediante una función que crea un `TRIGGER` que es activado cada vez que se modifica, inserta o borra una fila en la tabla. Si el `TRIGGER` es accionado por un `INSERT`, la fila es almacenada en la base de datos paralela creada por el módulo. En cambio, si es accionado por un `UPDATE` o `DELETE` se almacena el registro existente antes de ser ejecutada la sentencia.

Estos registros son almacenados en una columna, utilizando el tipo de datos semiestructurado nativo de PostgreSQL, `hstore`. De esta forma las filas de distintas tablas con diferente estructura pueden ser guardadas en un mismo campo. El sistema además guarda para cada registro modificado información como: el nombre de usuario que lo realizo, nombre de la base de datos, nombre de la tabla, fecha, etc.

4.5.2.1. Implementación de Audit Trigger 91plus en Presentes

Para utilizar este módulo en Presentes se ejecuta un archivo SQL que crea un nuevo esquema llamado `audit`, este cuenta con una única tabla con columnas que almacenan la información de los registros modificados. Cada fila representa el cambio de un registro de alguna tabla.

Es necesario especificar las tablas que se quiere llevar registro de sus cambios mediante la función `audit_table()` de la siguiente forma:

```
SELECT audit.audit_table('afectados.af_fotos');
```

Esto crea un `TRIGGER` en la tabla `af_fotos` que en caso de que ocurra un `INSERT`, `UPDATE` o `DELETE` llama a una función del módulo que se encarga de almacenar la información de los cambios que realizan estos comandos.

[14] https://github.com/2ndQuadrant/audit-trigger

Capítulo 5

Reingeniería de Presentes: Nuevo diseño de la Base de Datos

Varias de las soluciones y mejoras que requería el sistema no pudieron ser aplicadas en fases previas ya que implican grandes cambios en la estructura de la base de datos o no se adaptan al modelo relacional utilizado en su diseño. Por lo tanto se decidió realizar una reingeniería de la base de datos de Presentes.

El rediseño que se propone está orientado a un modelo de base de datos objeto relacional. Se utilizan algunas características del modelo orientado a objetos para definir las distintas entidades conceptuales descritas en la Sección 3.1, usando el método de Mapeo Objeto Relacional. También se agregan propiedades del modelo de dato documental empleando un formato para almacenar de forma semiestructurada documentos y otro tipo de información. Todo esto manteniendo las tablas, relaciones y lenguaje de consulta del modelo relacional.

En este capítulo se explican las decisiones que determinaron el desarrollo del nuevo diseño de la base de datos, a la vez que se manifiestan diferencias con el original.

5.1. Relaciónes Múltiples entre Tablas

Para satisfacer la necesidad de relacionar las distintas entidades conceptuales[1] entre si, es necesario contar con un identificador unívoco para cada fila entre las tablas que representan a estas entidades.

Para ello se utilizó la técnica Mapeo Objeto Relacional (MOR) pensando a estas entidades como clases que heredan de la super clase denominada `Objeto`. De esta forma las tablas que representan las entidades contienen los atributos persistentes de los objetos de esas clases. En particular, el atributo que interesa para este propósito es el Identificador de Objeto Persistente (POId[2]) que permite

[1] Víctima, Represores, Centro Clandestino de Detención, Organización, Fuente, Causa y Acto Procesal

[2] POId es información de tipo *Shadow Information* ya que no pertenece al dominio de los datos de los objetos.

tener un identificador común para los registros que representan instancias de las entidades.

Al utilizar este método, cada clase es mapeada a una tabla y la herencia es representada a través de claves foráneas entre los identificadores POIds de las tablas de las subclases y la superclase.

Las tablas que representan a Víctimas y Represores comparten varios atributos entre si, ya que contienen su información personal. Esto se repite en otras tablas que representan autores de documentos, personas que están relacionadas con Víctimas, etc. Por ello es conveniente generalizar las clases de estas entidades en una sola que llamaremos `Persona` y disponer de las siguientes subclases: `Víctima` y `Represor` dado que cuentan con atributos específicos.

En la figura 5.1 se puede observar la utilización del método MOR sobre las clases propuestas en esta Sección donde se mapea cada clase a una tabla.

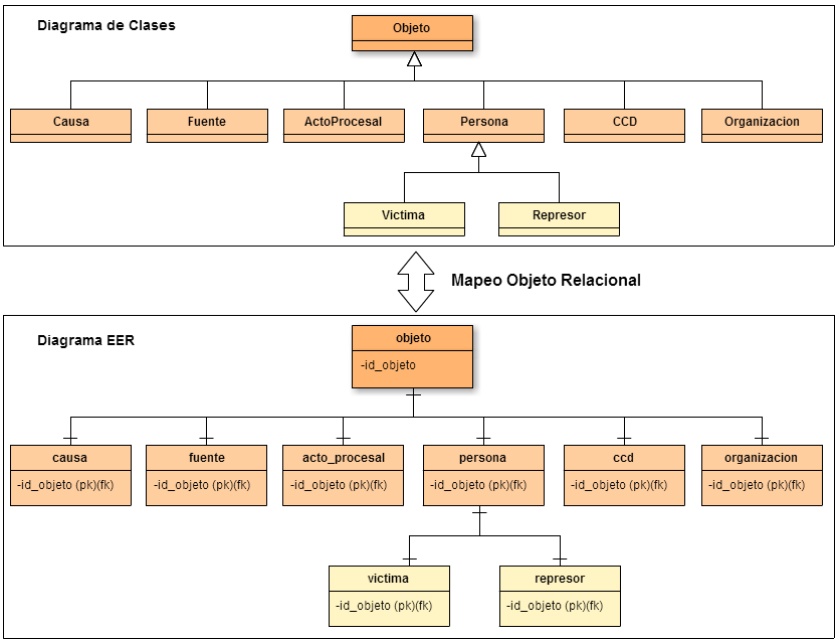

Figura 5.1: Utilización del MOR en las entidades conceptuales de Presentes. Para simplificar el diagrama solo se muestra el atributo POId.

Este diseño permite relacionar por ejemplo una Fuente con cualquier otra entidad conceptual, simplemente creando una relación entre las tablas `fuente` y `objeto`. También se puede vincular Causas con Víctimas, Represores y otras personas implicadas, estableciendo una relación entre las tablas `causa` y `persona`.

5.2. Restricción de Posibles Valores en las Columnas

En la implementación original de Presentes varias de las tablas son utilizadas para almacenar datos estáticos, que son referenciadas por otras tablas que emplean estos datos para restringir los valores de algunas de sus columnas.

En reemplazo de este sistema, en rediseño se definieron nuevos tipos de datos y dominios que permiten limitar los posibles valores de atributos sin emplear tablas adicionales. De esta manera se simplifica en gran medida la base de datos, ya que un considerable porcentaje de su estructura es debido a este tipo de tablas y la cantidad de claves foráneas que demanda su correcta utilización. Además, se facilitan las consultas dado que ya no es necesario combinar los registros de las columnas de las tablas relacionadas a este tipo de tabla para obtener los valores.

A continuación se presentan dos mecanismos provistos por PostgreSQL, que son aplicados en la implementación de este diseño para restringir los posibles valores de atributos.

5.2.1. Dominios

Cuando se definen los atributos de una tabla se especifica el conjunto de valores que cada uno puede almacenar. Esto se realiza asignando un tipo de dato a cada atributo, de esta manera se está escogiendo un dominio para estos. Por ejemplo, si a un atributo se le asigna el tipo de dato `VARCHAR(10)` sus valores deben ser cadenas de caracteres de longitud menor o igual a 10.

PostgreSQL permite definir dominios agregando restricciones a los valores de un tipo de dato. Utilizando como ejemplo los valores de la tabla 3.2 `estadocivil`, se muestra a continuación como se crea un dominio definido por el usuario:

```
CREATE DOMAIN estado_civil VARCHAR(10)
  CONSTRAINT estado_civil_valido
  CHECK (VALUE IN ('Casado/a', 'Soltero/a', 'En pareja',
                   'Viudo/a', 'Separado/a'));
```

A cualquier columna que se le asigne como dominio `estado_civil`, solo podrá contener como valores cadenas de caracteres que pertenezcan al conjunto `{'Casado\a', 'Soltero\a', 'En pareja', 'Viudo\a', 'Separado\a'}`.

5.2.2. Tipo de Datos Enumerado

Las tablas que además de almacenar valores estáticos tienen una columna que define un orden para estos valores (Ej. Tabla 3.3, `codimportancia`), son reemplazadas por Tipo de Datos Enumerado definidos por el usuario. Estos son creados en PostgreSQL mediante una sentencia de la siguiente forma:

```
CREATE TYPE importancia AS ENUM
  ('Sin importancia', 'Poca Importancia',
  'Normal', 'Importante', 'Muy Importante');
```

Como resultado se crea un nuevo tipo de dato denominado `importancia`, que es un conjunto de valores ordenados de menor a mayor según se listaron en el momento de su creación.

Para utilizar el nuevo tipo de dato se lo asigna como dominio a una columna, de igual manera que cualquier otro tipo de datos, como se muestra en el siguiente ejemplo:

```
CREATE TABLE tareas
( ...
  importancia importancia NOT NULL DEFAULT 'Normal',
  ...);
```

En el Apéndice D se listan los dominios y tipo de datos enumerado que fueron definidos en este diseño.

5.3. Almacenamiento de los Distintos Tipos de Documentos

En este diseño la información almacenada en las tablas específicas de cada tipo de Fuente es recopilada en una única columna llamada `documento` perteneciente a la tabla `fuente`. Esto se logra mediante el uso del tipo de dato `jsonb`, almacenando cada fila en un valor de tipo "nombre–valor". Se utiliza como "nombres" las denominaciónes de las columnas usada en la implementación original o similares que siga la convención de nombres adoptada en este diseño. Como "valores" se usan los valores que componen la fila.

El ejemplo a continuación muestra el valor de tipo `jsonb` que da como resultado procesar una fila de la tabla `fuen_libroguardiareg`.

```
{
  "obs": null,
  "hora": "T2x:xx:xx",
  "fecha": "1xxx-xx-xxT0x:xx:xx",
  "hojas": "1xx",
  "texto": "xxxxxx xxxx, xxxxxxxx xxxxxx",
  "entrada": "Detenidos 1xx",
  "id_libro": 2x,
  "dependencia": "Cxx",
  "id_registro": 5xxx
}
```

De esta forma los registros de las tablas que almacenan los distintos tipos de Fuentes y los de la tabla `fuentes`, de la implementación original, quedan fusionados en la tabla `fuente`. Esto evita el uso de tablas específicas para cada tipo de documento y permite almacenar nuevos, sin necesidad de crear otras tablas y relaciones.

5.4. Personas

En diversas tablas se almacena información de personas que pueden ser Víctimas, Represores, individuos que están vinculadas a Causas o a Víctimas, etc. y en algunos casos una persona puede cumplir más de un rol.

En este diseño se agrupan los datos de las personas (a excepción de los usuarios del sistema) en una única tabla llamada `persona`. Los atributos de esta tabla contienen los datos personales más comunes, tales como nombre, apellido, documento, etc. Además cuenta con una columna de tipo `jsonb` llamada `otros_datos`, en la que se almacena información particular según cada persona. Por ejemplo:

para Víctimas y Represores en este campo se guardan datos como el color de su cabello, altura, etc., mientras que para personas relacionadas a Víctimas en algunos casos se almacena su email, teléfono, etc.

En el diseño original tanto las Víctimas como los Represores pueden tener relacionados apodos y fotos[3]. En el nuevo diseño estas relaciones son unificadas y asociadas a las Personas en general.

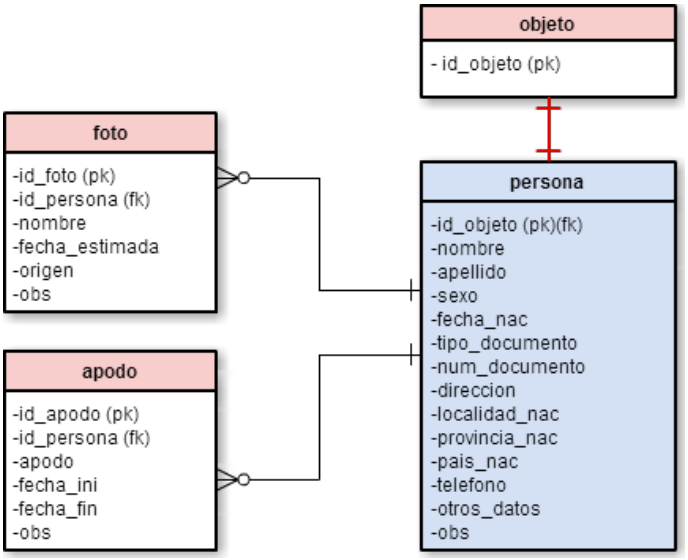

Figura 5.2: Diagrama Rediseño Personas.

Unificando las tablas que contienen información de Personas que cumplen distintos roles, se logra utilizar menos tablas y columnas que en el diseño original. Se evita repetir datos de Personas que cumplen con más de un rol y se encuentra de forma más sencilla los registros a los que están relacionados, dado que de esta manera no están repartidas en distintas tablas.

5.5. Convención de nombres

En el capítulo 2 se expusieron algunos problemas que ocasionó no seguir una convención de nombres homogénea para las tablas y columnas en el diseño original. Sumado a esto, la utilización de mayúsculas y espacios en los nombres produjo algunas dificultades a la hora de migrar de gestor de base de datos.

En el nuevo diseño, con el objetivo de evitar estos problemas, se definieron las siguientes reglas para la asignación de nombres a tablas y columnas:

- Los nombres de tablas y columnas se escriben en singular, excepto que esto implique que pierdan su semántica.
- Todas las letras de los nombres de tablas y columnas se escriben en minúscula.

[3]Las Víctimas utilizan las tablas `af_fotos` y `af_apodos` para almacenar fotos y apodos respectivamente, mientras que para los Represores se usan las tablas `rep_fotos` y `rep_apodo`

- Nombres de tablas o columnas que estén compuestas por más de una palabra se separan utilizando guión bajo "_".
- Al nombre de las columnas utilizadas como clave primaria o clave foránea se agrega el sufijo "id_".
- En las tablas se colocan primero las columnas que son claves primarias, seguidas de las claves foráneas y a continuación el resto.
- Los nombres utilizados deben ser descriptivos de su contenido.
- Los campos de distintas tablas que son utilizados para albergar la misma clase de información deben tener el mismo nombre.
- Los campos de distintas tablas que son utilizados para albergar distinta clase de información deben tener distinto nombre.

5.6. Usuarios

En la base de datos original el nombre y clave de ingreso de los usuarios al sistema son almacenados en la tabla `usuarios`. La interfaz accede a la base de datos y utiliza los datos de esa tabla para establecer los niveles de acceso asignados a cada usuario. Esta forma de administrar la clave de ingreso y los niveles de acceso es reemplazada en el nuevo diseño por el sistema de usuarios nativo de PostgreSQL, de este modo es el gestor de base de datos y no la interfaz el encargado de realizar esta tarea.

Por más que la información de los niveles de acceso y clave de ingreso no se almacenen en la base de datos, se siguen manteniendo tablas que guardan los datos personales de los usuarios, tareas asignadas y relaciones entre estas y cualquier entidad conceptual.

5.7. Diseño e Implementación de la Base de Datos Presente

En esta Sección se describe de forma detallada las tablas y relaciones que componen el rediseño de la de la base de datos Presente. Para explicar de una forma más sencilla, el diseño es dividido en las siguientes seis partes: "Fuentes", "Causas Judiciales y Actos Procesales", "Víctimas", "Represores", "Centros Clandestinos de Detención" y "Usuarios". Por último se exhibe el diseño completo resultante de unir a estas partes.

5.7.1. Parte 1: Fuentes

Como se puede observar en la Figura 5.3, las Fuentes se relacionan con cualquier entidad conceptual y viceversa[4]. Esto se consigue mediante la tabla `fuente_objeto` que implementa una relación "muchos a muchos", entre las tablas `fuente` y `objeto` con el uso de claves foráneas a sus respectivas llaves primarias. De esta manera (a diferencia del diseño original) se logra alcanzar

[4]Notar que de esta forma se puede relacionar los distintas documentos entre sí, lo que en la base de datos original se hacía con la tabla `fuent_refint`.

la integridad referencial en la relación entre Fuentes y cualquier otra entidad conceptual.

En la Sección 5.4 se comentó que la tabla `persona` almacena información de distintas Personas, entre ellas se encuentran autores[5] de algunos de los documentos almacenados en `fuente`. Para registrar esta información se implementa la relación "muchos a muchos" entre las tablas `fuente` y `persona`, mediante la tabla `fuente_autor`. De este forma se reemplazan las tablas `autores` y `fuentes_aut` utilizadas para esto en el diseño original.

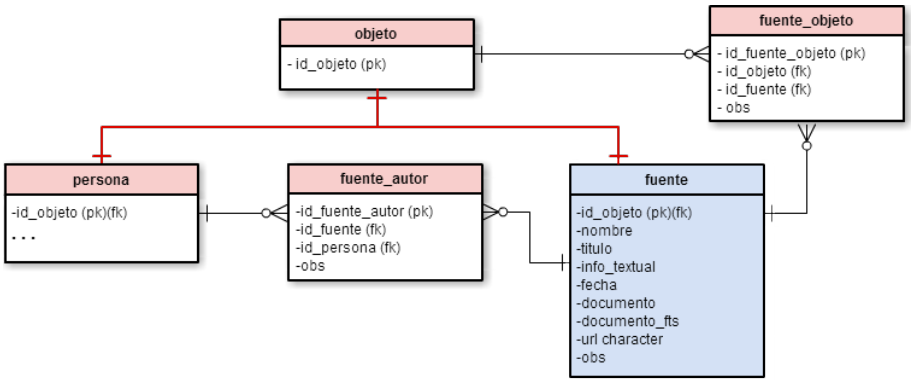

Figura 5.3: Diagrama Rediseño Fuentes.

5.7.2. Parte 2: Causas Judiciales y Actos Procesales

Los Actos Procesales forman parte de las Causas Judiciales, es decir que cada uno pertenece a una Causa Judicial determinada. Esto se ve representado en la base de datos con la relación "uno a muchos" mediante la clave foránea aplicada a la columna `id_causa` en la tabla `acto_procesal` (Figura 5.4). El valor de esa columna en los distintos registros, es el identificador de la Causa Judicial almacenada en la tabla `causa` a la que pertenece.

Las Personas pueden cumplir distintos roles en las Causas Judiciales como por ejemplo ser: 'Víctima', 'Imputado', 'Testigo', etc. Esto es declarado a través de la relación "muchos a muchos" que se implementa mediante la tabla `causa_persona`. Esta, además de relacionar las tablas `persona` y `causa`, contiene la columna `relacion_causa` que tiene asignada un dominio definido con el mismo nombre para restringir los valores que determinan los distintos roles.

[5] Algunos de estos autores pueden ser Víctimas o Represores.

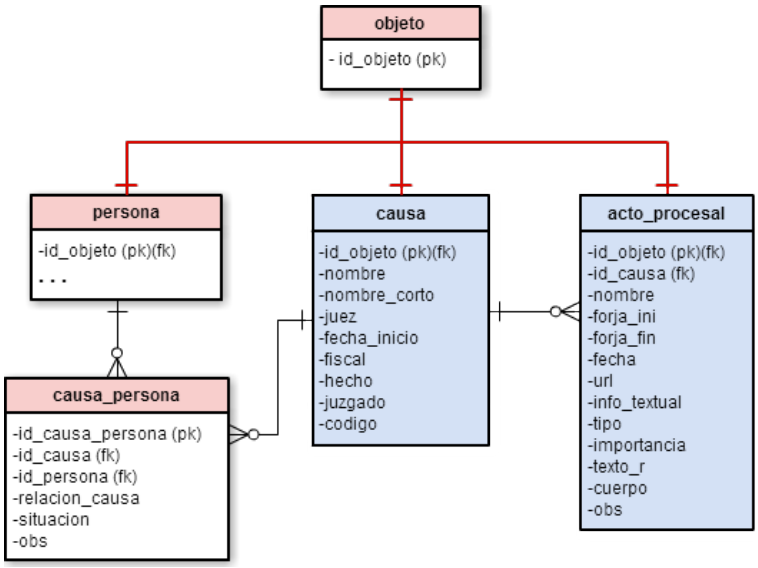

Figura 5.4: Diagrama Rediseño Causas y Actos Procesales.

5.7.3. Parte 3: Víctimas

La Figura 5.5 muestra como las Víctimas pueden tener asociada información sobre su secuestro e hipótesis de distintos hechos en que se las vincula. Para ello se utilizan las tablas `secuestro` e `hipotesis` respectivamente, ambas contienen relaciones "muchos a uno" al identificador de la tabla `victima`. Además, las Personas pueden estar ligadas a Víctimas ya sea por parentesco, amistad, etc. Como los datos de todas las Personas fueron reunidos en la tabla `persona`, se implementa la relación "muchos a muchos" entre esta y `victima` mediante la tabla `persona_victima`. También se puede vincular a las Víctimas con distintas organizaciones políticas, guerrilleras, etc. mediante la tabla `victima_organizacion` que implementa la relaciona "muchos a muchos" entre las tablas `victima` y `organizacion`.

En la tabla `victima` a las columnas `idanm`, `num_reg_fallecido` y `legajo` se les agrega la restricción `UNIQUE`, dado que estas deben contener valores únicos entre todos los registros.

Los posibles estados de las Víctimas: 'Desaparecido', 'Asesinado', etc. que antes se almacenaban en la tabla `estados` y se relacionaba mediante clave foránea, es reemplazada en este diseño por el dominio definido `estado` y asignado a la columna con el mismo nombre. Además, los datos personales de las Víctimas que en el diseño original están repartidos entre las tablas `af_casos` y `af_datper` y que no pertenecen a los atributos más usuales almacenados en la tabla `persona`, son incorporados en formato JSON en la columna `otros_datos` de esta tabla.

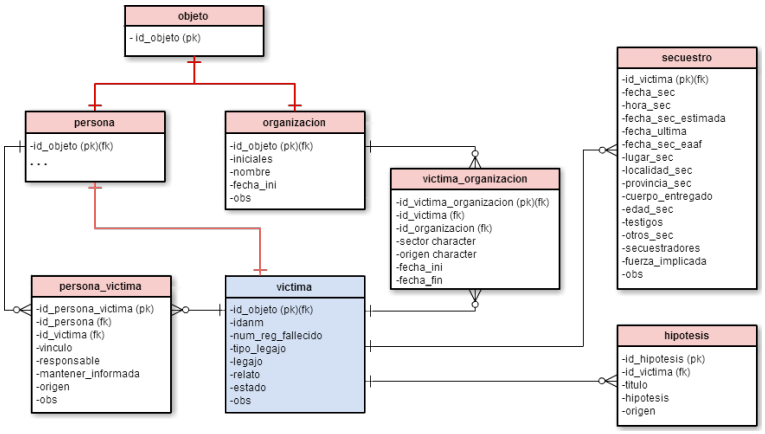

Figura 5.5: Diagrama Rediseño Víctima.

5.7.4. Parte 4: Represores

Los Represores (Figura 5.6) pueden tener asociados distintos rangos militares que obtuvieron a lo largo de su carrera en la fuerza armada de la que formaron parte. Esta información es almacenada mediante la tabla `represor_grado` que implementa una relación "muchos a muchos" entre las tablas `represor` y `grado`[6].

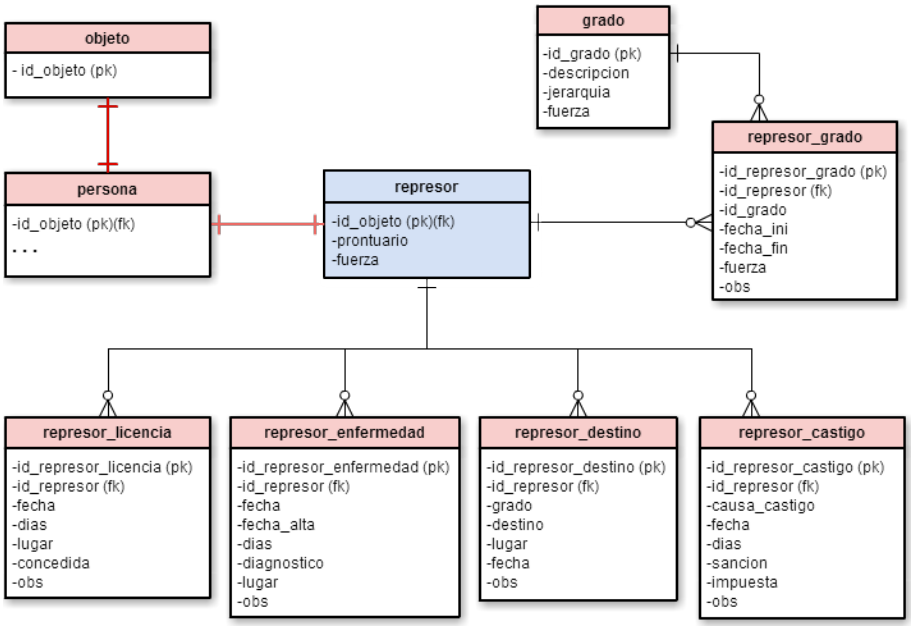

Figura 5.6: Diagrama Rediseño Represor.

De cada Represor se guarda información de los destinos a los que fue asignado, licencias concedidas, enfermedades que padeció y castigos que le fueron impuestos.

[6] La tabla `grado` almacena los distintos rangos militares de cada fuerza armada

Estos datos son almacenados en tablas separadas, cada una de ellas esta relacionada a la tabla `represor` mediante la clave foránea aplicada a `id_represor` que almacena el identificador del Represor al que está asociado. De esta manera se implementa una relación "muchos a uno" de cada una de estas tablas a `represor`.

De forma similar que se hizo para Víctimas, algunos de los datos personales de los Represores que en el diseño original están repartidos entre las tablas `repre_datper` y `represores`, son almacenados en formato JSON en la columna `otros_datos` de la tabla `persona`.

5.7.5. Parte 5: Centros Clandestinos de Detención

La Figura 5.7 ilustra la parte del diseño relacionada con los Centros Clandestinos de Detención (CCD). La tabla `ccd` que guarda información de cada CDD está vinculada tanto a Víctimas como a Represores mediante las tablas `represor_ccd` y `victima_ccd`. En esta relación se puede almacenar el rango de fechas en que tuvo lugar dicho vínculo. Estas tablas representan relaciones "muchos a muchos" de CCD con Víctimas y Represores.

Por otra parte la tabla `represor_ccd_victima` que vincula Víctimas con Represores, puede además especificar si el hecho que los asocia ocurrió en algún CCD. Esto se hace relacionando mediante una clave foránea el campo `id_ccd` con el identificador del CCD (`id_objeto`). La columna `id_ccd` puede ser `NULL` si el hecho no ocurrió en un CCD o en caso contrario, almacenar el identificador del registro en la tabla `ccd` al que está asociado.

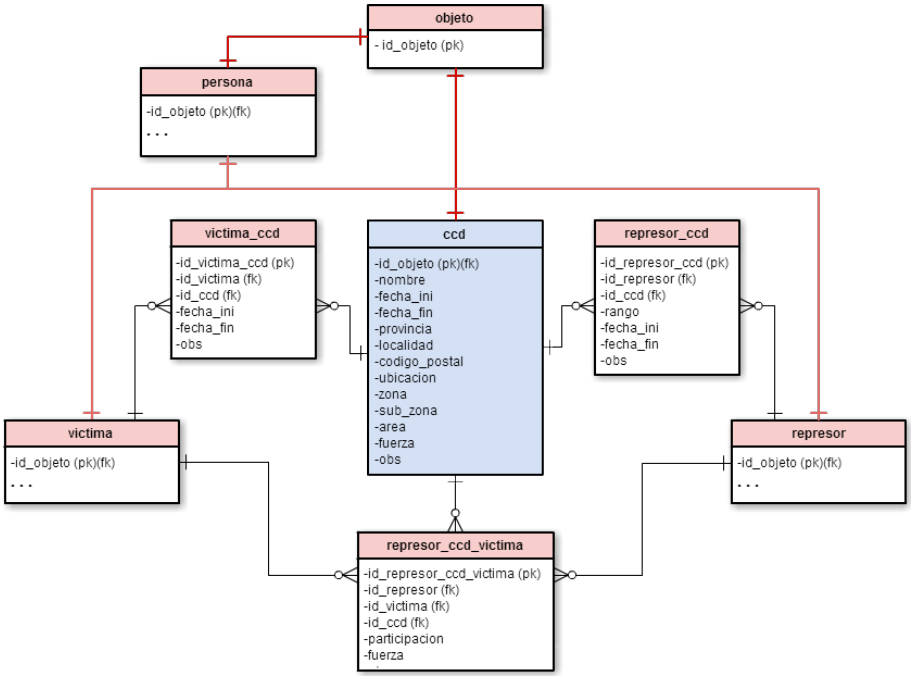

Figura 5.7: Diagrama Rediseño CCD.

5.7.6. Parte 6: Usuarios

Los datos de los Usuarios que almacena el sistema están representados en la Figura 5.8. Pueden tener asignadas tareas de diversos tipos que deben cumplir, para ello se implementa la tabla `tarea` que contiene esa información. En esta, el campo `importancia` tiene asignado como dominio un tipo enumerado definido del mismo nombre. Además, al campo `estado_tarea` se limita sus posibles valores mediante la asignación de un dominio definido de igual nombre. Los Usuarios están relacionados a las tareas que deben cumplir mediante la tabla `tarea_usuario` que describe una relación "muchos a muchos" entre las tablas `tarea` y `usuario`.

Las tareas pueden estar relacionadas con cualquier entidad conceptual, para ello se implementa una relación "muchos a muchos" mediante la tabla `tarea_objeto` que contiene los identificadores de los registros de las tablas `tarea` y `objeto` que se vinculan.

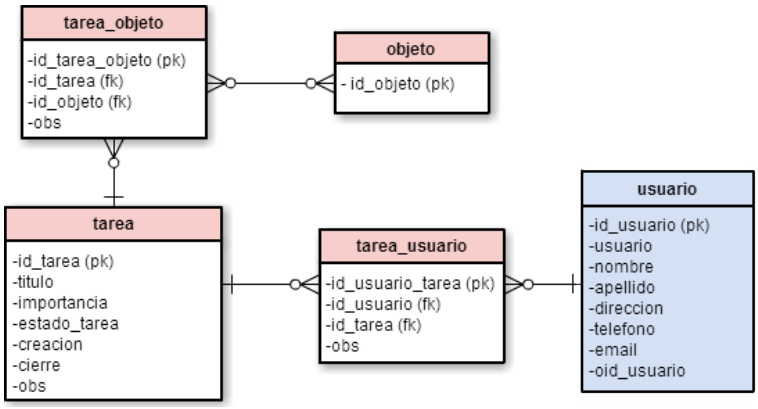

Figura 5.8: Diagrama Rediseño Usuarios.

5.7.7. Parte 7: El diseño completo

La figura 5.9 exhibe el diseño completo de la base de datos que da como resultado la unión de las distintas partes explicadas en las secciones anteriores. Para una mejor comprensión del diseño en su totalidad se eliminaron todos los atributos de las tablas salvo aquellos que son clave primaria o clave foránea.

La tabla `objeto` y las tablas que "heredan"[7] de esta, están coloreadas de un amarillo que se vuelve más claro a medida que se desciende en el árbol jerárquico de "clases".

[7] Hay que recordar que no es herencia de Programación Orientada a Objetos sino el resultado de aplicar el método Mapeo Objeto Relacional sobre herencia de clases.

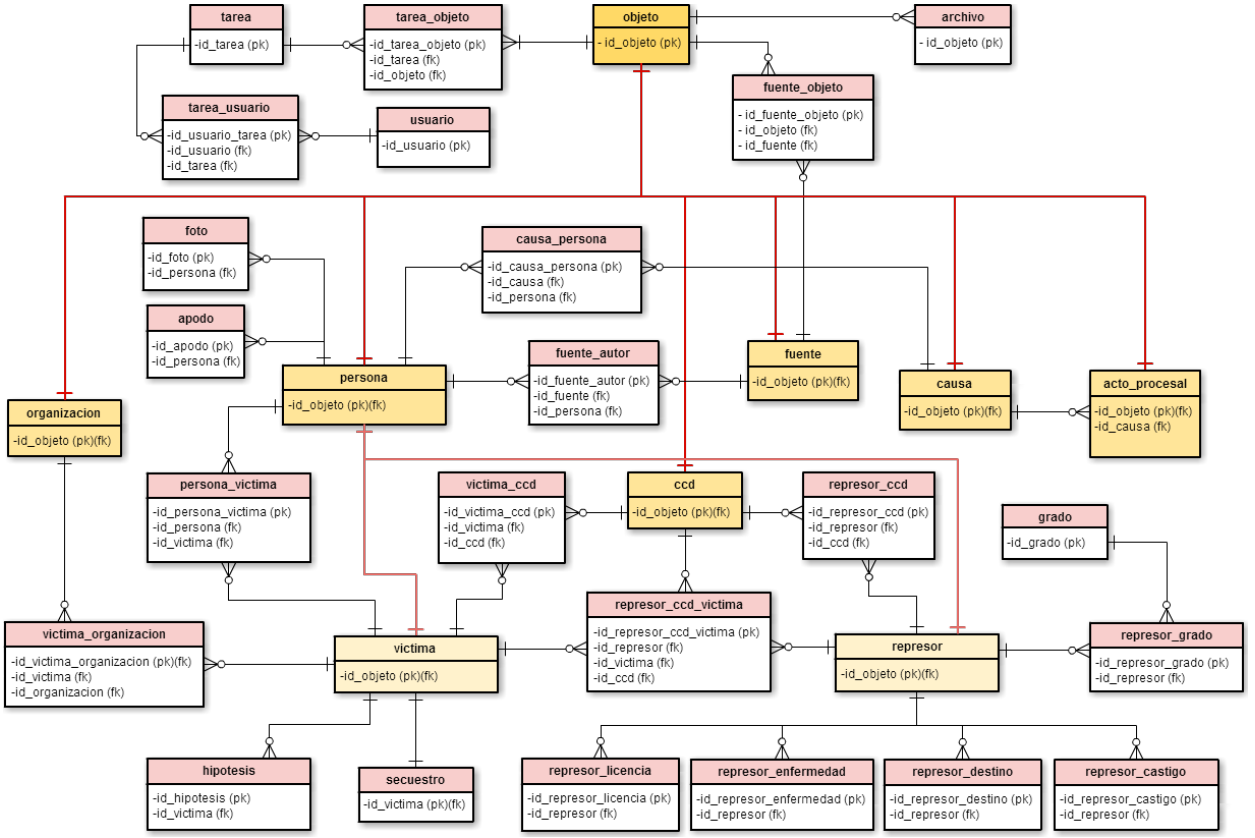

Figura 5.9: Diagrama Rediseño todas las tablas y solo las columnas que tienen clave primaria o clave foránea.

Capítulo 6

Proceso de Implementación del Nuevo Diseño de la Base de Datos en el Sistema Presentes

La implementación del nuevo diseño de la base de datos de Presentes fue planificada para ser adaptada al resto del sistema en cuatro etapas. De esta forma se divide el trabajo a la vez que permite ir utilizando las mejoras que agrega cada etapa del proceso en el que está dividido.

Las dos primeras son ejecutadas sobre la base de datos original. La etapa inicial consta de unificar las tablas que almacenan los distintos tipos de documentos y la segunda de asignar a determinadas columnas nuevos dominios y tipos de datos enumerados. La tercera es la más amplia y abarca la mayor parte del trabajo de implementación, se reemplaza la base de datos original por la propuesta en el nuevo diseño e implica la migración de datos entre las dos. La última etapa consiste en: implementar FTS en documentos y causas judiciales, configuran el sistema de usuarios de PostgreSQL y generar el sistema de historial de cambios en los registros de la nueva base datos.

Para poder implementar las modificaciones realizadas en la estructura de la base de datos original y por la migración de datos al nuevo diseño, es necesario modificar y borrar una gran cantidad de valores en registros de diversas tablas. Esto es debido a por ejemplo: no cumplir con el tipo de dato asignado al dominio de una columna, tener registros con valores repetidos en campos con restricción de unicidad, etc. La alteración de los datos y su migración es descrito en la última parte de este capítulo.

6.1. Etapa 1: Documentos

Como primer etapa se implementa sobre la base de datos original la parte del rediseño explicada en la Sección 5.3, donde se unifican las tablas que almacenan los distintos tipos de documentos. El primer paso para lograr esto es modificar la tabla `fuentes` (Figura 3.4) agregando una columna que será llamada `documento`

a la cual se le asigna como dominio el tipo de dato nativo de PostgreSQL `jsonb`.

Se seleccionan los campos de las tablas específicas de cada tipo de Fuente que se desea almacenar en el "documento"[1]. En general son todos a excepción del campo que contiene el identificador[2] y columnas que tienen el valor `NULL` en todas las filas. En esta selección varios de los campos son renombrados para seguir la convención de nombres adoptada en el nuevo diseño. La tabla que da como resultado es pasada como parámetro a la función de PostgreSQL `to_json()`. Esto devuelve un documento en formato `json` que luego es transformado al tipo de dato `jsonb` mediante el operador "::". El último paso consiste en almacenar los documentos en la columna `documento` de la tabla `fuentes` en los registros correspondientes.

El código a continuación muestra el proceso para generar los documentos a partir de los datos almacenados en la tabla `fuen_habeas` e insertarlos en la tabla `fuentes`.

```
UPDATE afectados.fuentes AS a SET documento =
  (SELECT  to_json(doc) FROM
    (SELECT b.apellido, b.nombres AS nombre,
            b."nroReferencia" AS numero_referencia,
            b.fecha , b."idJuz" AS id_juzgado
      FROM afectados.fuen_habeas AS b
      WHERE a."IDfuente" = b.idfuente
     ) doc
  )::jsonb
  FROM afectados.fuen_habeas
  WHERE a."IDfuente" = afectados.fuen_habeas.idfuente
```

Con esta etapa implementada se puede adaptar la interfaz de Presentes para usar los documentos almacenados en la tabla `fuentes` y dejar de utilizar las tablas específicas de los distintos tipos de documentos. La versión de esta tabla en el nuevo diseño se diferencia solo por algunos nombres, pero la columna `docuemento` no se modifica, por lo que la adaptación puede seguir siendo aprovechada a medida que se avanza en las siguientes etapas.

6.2. Etapa 2: Dominios y Tipos de Datos enumerados

En la segunda etapa se cambia el sistema de tablas utilizado para restringir posibles valores de atributos en los registros[3] según lo planteado en la Sección 5.2. Este sistema usa tablas auxiliares que contienen un listado de valores junto con un identificador al que se hace referencia desde las tablas que lo utilizan y en algunos casos además contienen un campo numérico adicional que le da un orden a los valores.

[1] En este contexto los documentos son los registros de la tablas específicas de las distintas Fuentes transformados a formato JSON

[2] Recordamos que el identificador de las tablas específicas de cada tipo de Fuente hace referencia a través de una clave foránea al registro de la tabla `fuentes`, que contiene campos comunes en todos los tipos.

[3] La tabla `objetos` también es parte de este sistema, pero no es reemplazada por un dominio o tipo de dato enumerado. En cambio, es sustituida en el nuevo diseño por la tabla `objeto` que almacena identificadores en vez de valores que determinan a qué entidad conceptual corresponde.

Este sistema es sustituido en la base original creando dominios y tipos de datos enumerados. Son reemplazados los campos que almacenan los identificadores a estas tablas por otros con el mismo nombre que contienen el valor correspondiente del registro al apuntaba. Para esto las columnas tienen que ser borradas y vueltas a crear, dado que el gestor restringe las posibilidades de cambio del tipo de dato asignado como dominio.

Algunas de las tablas utilizadas por este sistema que cuentan con una columna que le otorga un orden a sus valores, no son implementadas como un tipos de datos enumerados sino como un dominio. Esto responde al propósito de independizar la base de datos del resto del sistema ya que en estos casos, el ordenamiento cumple con la función de organizar el modo en que son presentados los valores en la interfaz. Por este motivo solo se implementan tipos de datos enumerados en aquellos casos en que el orden es utilizado para comparar entre sí valores almacenados en la base de datos.

En la siguiente Figura se muestra la tabla resultante de aplicar el cambio propuesto en esta etapa sobre el campo `estado`[4] de la tabla `af_casos`:

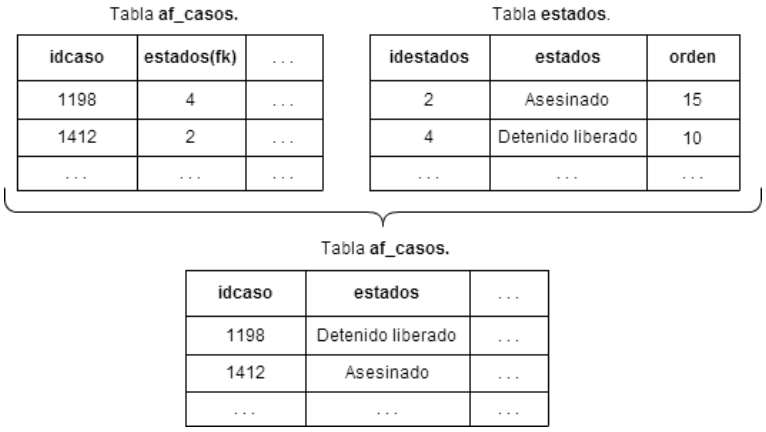

Figura 6.1: Estado referenciado desde en la tabla `af_casos` a la tabla `estados` reemplazado por el dominio `estado`.

De este modo se simplifican las consultas dado que ya no es necesario buscar en otras tablas los valores de los atributos. Además, las tablas utilizadas por la base de datos disminuyeron de 75 a 61 y las claves foráneas necesarias para relacionar los registros se redujeron de 91 a 64.

Aquí también los cambios que se hacen en la interfaz para adaptar estas modificaciones pueden seguir siendo usados en las siguientes etapas.

6.3. Etapa 3: Reemplazo Completo de la Base de Datos

El primer paso del proceso para reemplazar la base de datos original por la desarrollada en función del nuevo diseño, es crear un nuevo esquema en la base

[4]Notar que no se reemplaza por un tipo de dato enumerado ya que el orden que propone es utilizado sólo por la interfaz.

de datos que es nombrado `afectados_new`. Este es un contenedor que utiliza el gestor para generar un espacio de nombres de los objeto de la base de datos tales como: tablas, tipos de datos, funciones y operadores. De esta forma puede tener en distintos esquemas, tablas y otros objetos con el mismo nombre. Lo que permite tener funcionando en la misma base de datos la versión original y la nueva, facilitando así la migración de datos entre estas y la adaptación al sistema.

El siguiente paso es realizar la migración de datos, pero antes es necesario alterar muchos de los registros que contienen las tablas de la base de datos original (esto es explicado en detalle en la Sección 6.5) para que cumplan con las características y restricciones de la nueva versión. Luego se ejecuta un *script* que recoge los datos de las tablas del esquema `afectados` y los inserta en las correspondientes tablas de `afectados_new`. La inserción de estos datos es realizada en una secuencia específica para cumplir con las restricciones de claves foráneas en las distintas tablas.

Una vez realizado este proceso, la base de datos del nuevo diseño ya se encuentra en condiciones de ser adaptada al sistema. Las consultas deben ser modificadas para usar la nueva estructura. Aparte de cambiar los nombres de tablas y columnas, es necesario en la mayoría de los casos modificar el origen de los datos.

En la versión resultante hay columnas que serán borradas en un futuro, ya que con el tiempo dejarán de ser necesarias. Ejemplo de estas son las columnas nombradas `um` que se encuentran en varias tablas y que guardan el momento de la última modificación que se realizó en cada fila. También como se explica en la Sección 6.5.2 hay columnas en la tabla `objeto` que almacenan los identificadores de las entidades conceptuales de la base de datos original para migrar relaciones entre registros.

6.4. Etapa 4: FTS, Historial y Configuración de Usuarios

En esta última etapa y de forma similar a la aplicada sobre el diseño original en la Sección 4.5, se implementan sobre la base de datos `afectados_new` el sistema de historial y FTS. Esto se realiza como paso final dado que es necesario que las nuevas tablas con los datos migrados desde la base de datos original se encuentren en el esquema definitivo.

Los usuarios y sus niveles de acceso son configurados en el sistema de PostgreSQL con los datos de nombre de usuario y clave que se encuentran en la tabla `usuarios`. Luego las columnas `usuario` y `password` de esta tabla deben ser borradas.

Es requerimiento del APM que el sistema de niveles de acceso a la base de datos permita imponer restricciones a nivel de registros pero el sistema de usuarios de PostgreSQL solo aplica restricciones a nivel de tablas. Una forma de lograr restringir el acceso a determinados registros es creando vistas que los contengan, se les asignan permisos a estas vistas y se deniega el acceso a las tabla de origen.

6.5. Migración de Datos y Relaciones

Antes de migrar los datos de la base de datos original a la nueva, es necesario modificar la mayoría de los registros por diversos motivos. Algunos de los cambios se realizan sin la necesidad de un análisis detallado de los datos pero en otros casos se debe buscar y examinar los valores que deben ser modificados.

Por otro lado para poder conservar las relaciones entre los datos de las distintas tablas, es necesario almacenan de forma paralela los identificadores originales de las entidades conceptuales y los asignados de forma automática por la nueva base de datos.

6.5.1. Normalización de Datos

En general la mayoría de los registros contienen en alguno de sus atributos el número "0" o la cadena de caracteres de longitud cero para indicar que no tiene asignado un valor. Esto además de presentar un uso incorrecto de la estructura de la base de datos, genera problemas para realizar la migración de datos. Como solución se ejecuta una función que recorre todos los registros de la base de datos original, reemplazando las cadenas de caracteres de longitud cero por `NULL`. En el caso de los valores iguales a "0" se usa otra función que sólo se ejecuta en determinadas columnas, ya que en algunos casos es el valor correcto que debe almacenar. Antes de ejecutar estas funciones, se eliminan restricciones innecesarias sobre algunas columnas que no permiten que puedan contener el valor `NULL`.

Algunas de las columnas de la base de datos original tienen asignado un tipo de datos incorrecto para la información que debe almacenar. Por ejemplo, la columna `horades` de la tabla `af_secuestro` debe guardar la hora en que ocurrió un secuestro pero posee como dominio el tipo de dato `VARCHAR(50)`. Lo que permitió que en este campo se guarde además de la hora, texto introducido por los usuarios o prefijos generados por la interfaz, como `1899-01-01`, `1900-01-01` etc. La versión de esta columna en el nuevo diseño utiliza como dominio el tipo de datos `time` que solo permite almacenar horas, minutos y segundos. Antes de migrar los datos se realiza una búsqueda usando expresiones regulares para encontrar los valores que no cumplen con este formato para luego modificarlos.

Otras columnas que son necesario modificar, son las utilizados para almacenar el número de documento, identificador en archivo nacional de la memoria y número del registro nacional del fallecido. En la base de datos original estas tienen asignadas como dominio al tipo de dato `VARCHAR(50)`, a diferencia de la nueva base de datos que utiliza `INTEGER`. Estos presentan los mismos problemas que el caso anterior y además deben cumplir con la restricción de unicidad. Lo que llevó a encontraron registros repetidos o que tenían diferencias en algunos de sus valores pero representaban a la misma persona.

Los problema presentados anteriormente son los más generales que se repiten en varios atributos, pero hay otros más puntuales que afectan a pocos registros.

6.5.2. Migración de Entidades Conceptuales y Datos Relacionados

Para poder migrar los registros que representan a las distintas entidades conceptuales, se agregon en la nueva base de datos las columnas auxiliares `old_type_tmp` y `old_id_tmp` en la tabla `objeto`. La columna `old_id_tmp` es

utilizada para almacenar identificadores de los registros que representan las entidades conceptuales en el diseño original. Como se puede recordar estos identificadores son de tipo numérico salvo los de Fuentes, que utilizan cadenas de caracteres. Por esta razón todos los identificadores son guardados en la columna `old_id_tmp` con formato de texto. Por otro lado, la columna `old_type_tmp` se usa para indicar a qué entidad conceptual pertenece el identificador. Para ello almacena el valor correspondiente del atributo `idObj` de la tabla `objetos` del diseño original.

La tabla `objeto`, además de las columnas ya mencionadas, contiene una de tipo `serial` llamada `id_objeto` que asigna automáticamente un identificador numérico a cada nuevo registro. De esta forma cada fila en la tabla contiene: un identificador único entre todos los registro de las entidades conceptuales, el identificador que tenían en la base de datos original y su tipo.

La Figura 6.2 ilustra como quedan los registros en la tabla `objeto` luego de migran a la nueva base de datos identificadores de las tablas `fuentes` (Fuentes) y `af_casos` (Víctimas).

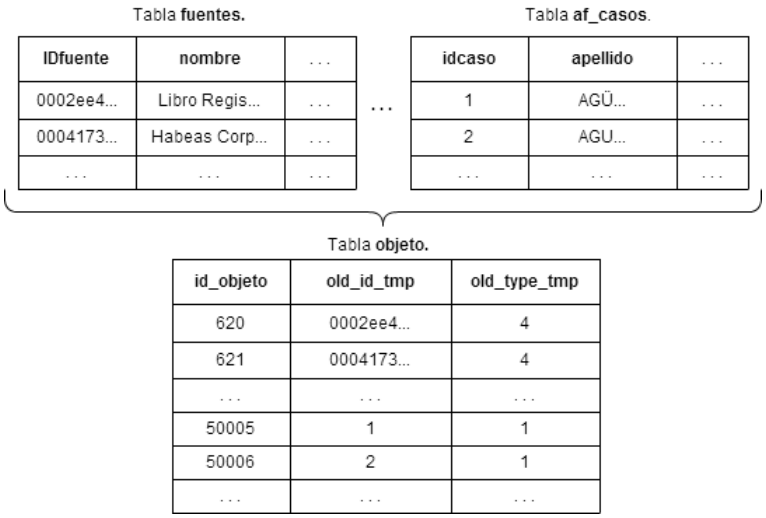

Figura 6.2: Método para conservar los identificadores de las entidades conceptuales en la tabla `objeto`.

Cada vez que se debe guardar un registro en una tabla con campos que se relacionan mediante clave foránea a las entidades conceptuales, es necesario consultar en la tabla `objeto` su antiguo identificador. En base a este, se cambia el valor del identificador que hace referencia por el nuevo que fue asignado en la tabla `objeto`.

La nueva estructura contiene relaciones entre tablas que, a diferencia de la base de datos original, están implementadas mediante el uso de claves foráneas. Cuando se migraron los datos a tablas con estas nuevas restricciones se encontró una gran cantidad de registros huérfanos. La mayoría fueron obviados ya que sin los datos a los que se relacionan carecen de significado.

Capítulo 7

Conclusiones y Trabajo Futuro

Si bien se alcanzó el objetivo de realizar la reingeniería de la base de datos de Presentes que es completamente funcional y está documentada, aun no se concreto la adaptación de esta a la interfaz. En el momento de presentar este trabajo se encuentra implementado en el sistema en uso solo lo descrito en el Capítulo 3, que abarca la primer etapa. Esto es consecuencia de la complejidad que presenta para los responsables del sistema adaptar los cambios para migrar de gestor de base de datos, a lo que se suma la limitación del tiempo que disponen para efectivizarlo.

Aspiro poder aplicar las etapas restantes en un futuro cercano, de no ser así considero que el trabajo realizado es significativo ya que se logró esclarecer su funcionamiento, se mejoraron varios aspectos[1] y se hizo un análisis detallado de carencias que presenta. Además hay muchas ideas que se desprenden de este trabajo que pueden ser utilizadas. Ejemplos de mejoras que pueden ser desarrolladas en la base de datos que utiliza actualmente el sistema son: almacenar las Fuentes en un formato semiestructurado, eliminar tablas que restringen posibles valores de columnas, unificar tablas que almacenan información similar, etc.

Ya que la nueva base de datos nunca pudo ser usada en el sistema Presentes, solo se probaron pocas consultas generadas por la interfaz que nos fueron facilitadas para ayudar a migrar el gestor. Estas arrojaron los mismos resultados en la versión de PostgreSQL de la base de datos original que en la nueva. El resto de pruebas se hicieron para corroborar que se migrara correctamente los datos de una base a otra, comparando la similitud de sus registros.

A continuación y salvando las dificultades mencionadas, se describen las mejoras que a mi parecer presenta el trabajo de reingeniería de la base de datos de Presentes.

[1] En la etapa que fue implementada en el sistema se borraron tablas obsoletas, fueron agregadas claves foráneas y se modificó el dominio de columnas.

7.1. Mejoras Alcanzadas

A lo largo de todo el proceso se aplicó una gran cantidad de cambios. La base de datos fue evolucionando por etapas hasta alcanzar un diseño completamente nuevo. Posee una estructura más compacta, con una amplia relación explícita entre los datos. Se mejoró la calidad de la información que almacena eliminando redundancias e inconsistencias, agregando integridad referencial y escalabilidad. Asimismo se incorporaron nuevos sistemas que amplían la funcionalidad de Presentes. En las siguientes secciones se detallan los logros alcanzados con la implementación del nuevo diseño, la mejora en la calidad de los datos y nuevas funcionalidades.

7.1.1. En Cuanto al Diseño

Como resultado de este trabajo hay evidentes mejoras con respecto al cambio de diseño. La base de datos en un comienzo contaba con 75[2] tablas que se vieron reducidas a 32. La cantidad de relaciones implementadas mediante claves foráneas disminuyeron de 91[3] a 41.

Si bien hay una gran disminución en el número de tablas y relaciones, se alcanzó mayor organización y estructuración de la información. Se unificaron tablas que almacenan información similar y se añadieron relaciones explícitas faltantes que, como es el casos de las de entidades conceptuales, eran muy complejas de implementar en la base de datos original. Además, se añadieron restricciones a columnas que evitan información redundante o inconsistente. Las reglas utilizadas para dar nombre a tablas y columnas hacen que el código sea más claro y autodescriptivo, lo que facilita la comprensión y escritura de consultas.

Estas mejoras se ven reflejadas en consultas más simples. La información ya no esta dispersa en tantas tablas lo que facilita las búsquedas de datos relacionados. Además es más sencillo borrar y actualizar registros asociados mediante clave foránea, dado que estas acciones se propagan en cascada gracias a las cláusulas `ON DELETE CASCADE` y `ON UPDATE CASCADE` que indican qué hacer en estos casos.

Se quitaron de varias tablas las columnas que almacenan datos usados por el sistema para presentar formularios de consultas u organizar resultados en la interfaz gráfica de usuario. De esta forma se logra una mayor independencia con el resto de la aplicación.

La nueva base de datos posee la documentación que presenta este informe, donde se exponen diagramas de su diseño y la explicación de los mismos. También, como parte de la documentación, existe un archivo que muestra donde se almacenan los valores de cada campo de la base de datos original en el nuevo diseño. Esto permite una comprensión clara de su funcionamiento, a la vez de facilitar futuras modificaciones.

7.1.2. En Cuanto a los Datos

A lo largo de las tres etapas de este trabajo fue mejorando la calidad de los datos que almacena la base de datos. Se elimina información redundante,

[2] Después de borrar las tablas obsoletas (Sec. 3.2.1).
[3] Después de agregar las laves foráneas faltantes (Sec. 3.2.2).

principalmente por la unificación de tablas que tenían datos o estructuras similares. Se consiguió una mayor integridad referencial de los datos que se relacionan, desechando en el proceso una gran cantidad de registros huérfanos.

Muchos datos fueron modificados para que pertenezcan al dominio asignado a las columnas en el nuevo diseño. Por ejemplo: columnas que debían almacenar valores de horas y eran almacenadas como fechas, distintos valores numéricos que se guardaban como cadena de caracteres, etc. Se reemplazaron por `NULL` una gran cantidad de registros que contenían el "cero" o cadena de caracteres de longitud cero para representar que no tienen asignado un valor.

Estas modificaciones en los datos no sólo permitieron que se almacenen con el correcto dominio según el requerimiento de cada tipo de dato, sino que además posibilita agregar restricciones como la unicidad o claves foráneas en distintas columnas.

Por medio del tipo de dato `jsonb` se sumó la posibilidad de almacenar documentos e información particular de las personas sin necesidad de agregar o modificar tablas específicas para ello.

7.1.3. En Cuanto a la Funcionalidad

La base de datos resultante cuenta con la capacidad de realizar búsquedas mediante el sistema Full Text Search de PostgreSQL en los documentos y causas judiciales. Los resultados que arrojan las búsquedas realizadas con este sistema mejora en varios aspectos a la común. Presenta un gran rendimiento de velocidad al utilizar índices específicos para este sistema. Encuentra palabras derivadas y sinónimos de los términos de búsqueda. Da la posibilidad de ordenarlos según su relevancia, analizando la cantidad de ocurrencias y cercanía de estos en los documentos. También se puede adaptar el sistema a requerimientos específicos de búsqueda permitiendo agregar diccionarios y cambiar algoritmos de reconocimiento de palabras.

Otra funcionalidad que se agrega es el historial implementado mediante un sistema que utiliza un esquema paralelo al usado por las tablas de Presente, con funciones y triggers que permiten llevar un registro minucioso de todos los cambios en los datos de forma automática y que es totalmente independiente de la interfaz.

También se añade la utilización del sistema nativo de PostgreSQL para la administración de los usuarios que aporta mayor seguridad para el acceso a la base de datos, dado que la clave es controlada por el gestor.

7.2. Dificultades que se Presentaron en el Desarrollo del Trabajo

El desarrollo de la primera y segunda etapa requirió más tiempo del planificado en un comienzo. Creo haber subestimado la dificultad que implica realizar la ingeniería inversa de Presentes y el cambio de gestor de base de datos.

El mayor inconveniente que afectó el inicio de este trabajo fue la escasa documentación sobre el diseño y su funcionamiento. En un principio sólo contamos con un archivo SQL para generar la estructura de la base de datos. Dada la gran cantidad de tablas, falta de relación mediante claves foráneas, confusos nombres utilizados y la forma de identificar y relacionar las distintas entidades

conceptuales hizo que comprender el funcionamiento de la base de datos resulte una tarea compleja. Ese problema se superó cuando nos otorgaron acceso a los datos y el manual de usuarios del sistema. Se comprobó que ciertas partes no funcionaban como se pensaba y que algunas columnas no almacenaban la información que aparentaban.

Otro de los problemas fueron las constantes modificaciones de la estructura de Presentes. En el momento que se desarrolló el proceso de migración, la base de datos utilizada en el APM no contaba con un control de versiones de su estructura. Cada vez que requerían pasar a PostgreSQL la base de datos que usaban en ese momento (para adaptar la interfaz), era necesario hacer un recorrido por las tablas y columnas en busca de cambios y así poder actualizar los archivos necesarios para la ejecutar la migración de gestor de base de datos.

Casi todos los registros presentaban alguna clase de problema. En todas las etapas llevo bastante tiempo encontrar y arreglar esas fallas. Si bien se mejoró en gran medida la calidad de los datos, seguramente quedan muchos errores e inconsistencias que no fueron encontrados. En ningún momento fue un objetivo solucionar exhaustivamente estos errores, ya que los datos con que contamos no están actualizados. Además que la base de datos cuenta con otras dos instancias, seguramente con distintos tipos de problemas. Por estas razones los arreglos solo fueron hechos para que los datos puedan ser almacenados en las estructuras que se desarrollaron en las distintas etapas.

7.3. Trabajo Futuro

La base de datos producto de este trabajo tiene un importante potencial para ampliar su funcionalidad. Se estudiaron diversos desarrollos de software libre con que cuenta PostgreSQL en materia de geolocalización y sistemas que implementan bases de datos distribuidas. Por otro lado, al tener un diseño documentado con integridad en sus datos e independiente del resto del sistema facilita la realización de una reingeniería de la interfaz.

7.3.1. Interfaz

El próximo gran paso en el desarrollo de Presentes es realizar la reingeniería completa del resto del sistema. La falta de modularización entre los distintos componentes del programa acarrea grandes problemas para su mantenimiento y desarrollo. En particular para este trabajo es el principal obstáculo que impidió concretar el cambio de gestor de base de datos.

Una opción para implementar una nueva interfaz es mediante un *framework* que utilice el patrón de diseño MVC (*Model View Controller*) que divide al sistema en: una parte encargada de comunicarse con la base de datos (Model), la lógica de la aplicación (Controller) y la presentación gráfica de información para usuarios (View).

En el nuevo diseño se usó el método MOR como forma de relacionar las entidades conceptuales entre sí. Para desarrollar la parte encargada de comunicarse con la base de datos se puede utilizar ese mismo método de manera inversa. Es decir que a partir de la base de datos de Presentes se generan clases cuyos objetos almacenan sus datos persistentes en la base de datos. El software Hibernate[4],

[4] http://hibernate.org/

distribuido bajo licencia GNU LGPL, posee una herramienta que permite mapear tablas y relaciones de una base de datos a objetos generados automáticamente en el lenguaje java. Por cada tabla crea una clase con funciones `Set()` y `Get()` y un archivo XML de metadatos de mapeo, lo que permite manejar los registros de una base de datos relacional como objetos. De esta forma se crea una capa que simplifica la interacción con la base de datos, dado que se deja de utilizar el lenguaje SQL y se suman las ventajas de la programación orientada a objetos.

Una característica que se puede agregar a la interfaz aprovechando el historial de cambios en los registros que almacena la base de datos es el control de versiones, donde se liste el historial de cambios de un registro y se permita volver a versiones anteriores.

7.3.2. Geolocalización

En el APM se han desarrollado sistemas que utilizan datos geoespaciales. Este tipo de dato puede ser usado en la base de datos de Presentes para agregar información que indique referencialmente la ubicación geográfica donde sucedió un hecho, domicilio de las personas, etc. Para ello PostgreSQL es una excelente herramienta ya que cuenta con un desarrollo de larga data en materia de geolocalización a través del modulo PostGis[5] distribuido bajo licencia GNU GPL. Este módulo extiende al gestor permitiendo almacenar objetos georreferenciados[6] como puntos, líneas o polígonos de acuerdo a las especificaciones OpenGIS[7]. Además dispone de funciones para el procesamiento y análisis de estos objetos.

Existe una gran cantidad de programas GIS (*Geographic Information Systems*) tanto de licencia libre como comercial que permiten visualizar, editar y analizar datos almacenados en PostgreSQL con el módulo PostGIS.

7.3.3. Base de datos distribuida

Como se comentó anteriormente, Presentes cuenta con otras instancias ejecutándose en distintos servidores, en cada uno de estos se administran los datos en forma independiente.

En ocasiones los datos que se guardan en una instancia deben ser trasladados a otra, lo que se realiza sin utilizar un método automático. Esto no solo implica el trabajo de migrar registros de una base a otra, sino que previamente se debe verificar que estos datos o similares[8] no se encuentren ya en la base de datos de destino. Probablemente muchos de los registros en la instancia del APM que se encontraron con información redundante o huérfanos fueron ocasionados por estas migraciones.

Una solución para evitar este trabajo es utilizar una base de datos distribuida donde cada nodo sea una réplica de los otros. PostgreSQL cuenta de forma nativa con esta capacidad[9], también hay varias herramientas de software libre que se pueden utilizar. De esta forma los cambios generados en cualquier instancia se verían reflejados inmediatamente en las otras. Lo que resultaría muy eficaz en

[5] http://postgis.net/source

[6] http://es.wikipedia.org/wiki/Georreferenciación

[7] http://www.opengeospatial.org/standards/as

[8] Por ejemplo, una misma persona puede estar almacenada de forma distinta en una instancia que en otra.

[9] http://www.postgresql.org/docs/9.4/static/warm-standby.html

cuanto al trabajo colectivo de las organizaciones que lo utilizan, para lo cual tendrían que acordar compartir por completo los datos que genera cada una.

Apéndice A

Preproyecto de Trabajo Especial

"Presentes" es un software utilizado en el Archivo Provincial de la Memoria para la investigación del accionar del terrorismo de estado en la provincia de Córdoba. Contiene una base de datos que condensa y organiza información recopilada a lo largo de años de constante investigación. Actualmente, la implementación del sistema no fue desarrollada mediante un proceso de producción de software adecuado. En este proyecto proponemos realizar una reingeniería de "Presentes" utilizando metodologías basadas en Procesos de Ingeniería del Software, incluyendo la realización de una especificación de requerimientos, un diseño y una implementación utilizando herramientas de código abierto.

A.1. Contexto

Desde el año 2001 la organización H.I.J.O.S. Córdoba viene desarrollando un software destinado a facilitar la investigación del accionar del terrorismo de estado en la Provincia de Córdoba. Este sistema informático, denominado "Presentes", contiene una base de datos que condensa y organiza información recopilada a lo largo de años de constante investigación.

Recientemente, en carácter de donación, "Presentes" fue entregado al Archivo Provincial de la Memoria[1], con el propósito de ser utilizado en tareas de coordinación de investigaciones, articulación de trabajos en común y exposición de información en los Sitios de Memoria bajo la órbita de la Comisión Provincial de la Memoria.

Para hacer uso de "Presentes", el Archivo tiene la necesidad de realizar un conjunto importante de modificaciones y extensiones al software. El sistema no fue desarrollado mediante un proceso de producción de software adecuado, con todos los problemas que de esto derivan. En particular, la base de datos del sistema presenta importantes deficiencias, producto de la falta de un diseño estructurado correctamente documentado. A esto, se agrega que las prestaciones del motor de base de datos utilizado resultan limitadas para responder a las nuevas funcionalidades que se pretenden añadir al sistema.

En consecuencia, y en el contexto del Convenio de Colaboración[2] celebrado

[1] Legislatura de la Provincia de Córdoba, Argentina, Ley N° 9826, 2006.
[2] Resolución Decanal N° 248/2012.

el presente año entre la Facultad de Matemática Astronomía y Física y el Archivo Provincial de la Memoria, proponemos realizar una reingeniería del sistema informático "Presentes", teniendo en cuenta requerimientos relevados y planteados por uno de sus desarrolladores, el Sr. Marcelo Yornet, coordinador del área de digitalización del Archivo Provincial de la Memoria y miembro del área de investigación de la organización H.I.J.O.S.

A.2. Objetivo

El objetivo de este trabajo es la realización de una reingeniería del sistema informático "Presentes". Se espera obtener como resultado un nuevo sistema informático en funcionamiento, correctamente documentado y con un proceso de desarrollo detallado que permita la realización de subsecuentes mejoras y actualizaciones.

Se aplicarán metodologías basadas en Procesos de Ingeniería del Software , que incluirán la realización de una especificación de requerimientos y un diseño debidamente documentados, y una implementación utilizando herramientas de código abierto.

A.2.1. Plan de trabajo

- Formación preliminar y elección de modelo de Proceso de Producción de Software.
- Especificación de requerimientos del software, con un documento de requerimientos como resultado.
- Diseño del software, con un documento de diseño como resultado.
- Implementación.
- Testeo y simulación de casos de uso concretos.
- Evaluación de resultados y conclusiones.

Apéndice B

Vistas y Tablas Eliminadas

B.1. Vistas:

- qryrepre_dest
- af_qryfuente
- listado
- listado sin fecha
- profesion
- af_listado
- #mysql50#listado sin fecha
- estadisticas
- listaupdate
- consulta1
- fuen_qrytest
- est_por_meses

B.2. Tablas:

- actosproc
- actualizaciones
- af_tareas
- af_testimonios
- caidas
- codifuentes
- codnomjuz
- destinos
- fuen_boletines_pc
- fuen_carpmemos_pf
- fuen_docvar_pf
- fuen_ent_lg
- fuen_entrevista_apm
- fuen_libcop_pf
- fuen_libroguardia_pc
- fuen_morgue
- fuen_ordendia_pc
- fuentes_autrem
- fuentes_jud
- fuentes_test
- legales
- permisos
- repre_dest
- roles
- u_usuarios
- #mysql50#codigos familia
- #mysql50#codigos provincia

Apéndice C

Claves Foráneas Implementadas

af_apodos.idcaso ⟶ af_casos.idcaso
af_casos.idprov ⟶ ‘codigos provincia.COD
af_casos.estado ⟶ estados.idestado
af_casos.tipo_documento ⟶ tipo_docs_cod.id
af_cce.idcaso ⟶ af_casos.idcaso
af_cce.idCCE ⟶ cce.idCCE
af_datper.estcivil ⟶ estciviles.idestcivil
af_datper.idcaso ⟶ af_casos.idcaso
af_datper.idprovnac ⟶ ‘codigos provincia‘.COD
af_datper.provtrab ⟶ ‘codigos provincia‘.COD
af_fotos.idcaso ⟶ af_casos.idcaso
af_hipotesis.idcaso ⟶ af_casos.idcaso
af_militancias.idcaso ⟶ af_casos.idcaso
af_militancias.idorga ⟶ orgas.idorga
af_relaciones.idcaso ⟶ af_casos.idcaso
af_relaciones.idprovincia ⟶ ‘codigos provincia‘.COD
af_relaciones.idrelacion ⟶ ‘codigos familia‘.idrelacion
af_secuestros.idcaso ⟶ af_casos.idcaso
af_secuestros.provdes ⟶ ‘codigos provincia‘.COD
archivos.tipoObj ⟶ objetos.idObj
autores.tipoAutor ⟶ tipo_autor.id_tipodeautor
autores.tipoObjeto ⟶ objetos.idObj
causa_actor.idcausa ⟶ causas.idcausa
causa_actor.relacion ⟶ causa_actrel.idrel
causa_actor.tipoObj ⟶ objetos.idObj
causa_indice.idcausa ⟶ causas.idcausa
causa_indice.idfuente ⟶ fuentes.IDfuente
causa_indice.importancia ⟶ codimportancia.idImp
causa_indice.tipo ⟶ causa_tipos.idTipo
causas.idJuzgado ⟶ juzgados.idjuzgado
cce.FUERZA ⟶ fuerzas.idfuerza
cce.prov ⟶ ‘codigos provincia‘.COD

cce.tipoObjeto ⟶ objetos.idObj
fuentes_ap.tipoObjeto ⟶ objetos.idObj
fuentes_refint.idfuente ⟶ fuentes.IDfuente
grados.fuerza ⟶ fuerzas.idfuerza
logs.tipoObjeto ⟶ objetos.idObj
logs.usuario ⟶ usuarios.usuario
orgas.tipoObjeto ⟶ objetos.idObj
rep_apodo.idrepre ⟶ represores.idrepre
rep_cce.idCCE ⟶ cce.idcce
rep_cce.idrepre ⟶ represores.idrepre
rep_destinos.idrepre ⟶ represores.idrepre
rep_enfermedad.idrepre ⟶ represores.idrepre
rep_fotos.idcaso ⟶ represores.idrepre
rep_licencias.idrepre ⟶ represores.idrepre
repre_datper.estcivil ⟶ estciviles.idestcivil
repre_datper.idprovnac ⟶ ‘codigos provincia‘.COD
repre_datper.idrepre ⟶ represores.idrepre
repre_grado.fuerza ⟶ fuerzas.idfuerza
repre_grado.grado ⟶ grados.idgrado
repre_grado.idrepre ⟶ represores.idrepre
represores.fuerza ⟶ fuerzas.idfuerza
represores.tipo_documento ⟶ tipo_docs_cod.id
represores.tipoObjeto ⟶ objetos.idObj
reprevict.fuerza ⟶ fuerzas.idfuerza
reprevict.idrepre ⟶ represores.idrepre
reprevict.idvict ⟶ af_casos.idcaso
reprevict.lugar ⟶ cce.idCCE
reprevict.participacion ⟶ participacion.idparticipacion
searchhistory.tipoObjeto ⟶ objetos.idObj
searchhistory.usuario ⟶ usuarios.idusuario
tarea_obj.tipoObjeto ⟶ objetos.idObj
tarea_usuarios.grupo ⟶ u_grupos.idgrupo
tarea_usuarios.idusuario ⟶ usuarios.idusuario
tareas.estado ⟶ cod_estadotareas.idEstado

Apéndice D

Dominios y Tipos de datos Enumerados

D.1. Dominios

Tipo de documentos de identidad: `tipo_documento`

Vínculos entre personas y victimas: `vinculo`

Relación entre personas y causas: `relacion_causa`

Los distintos tipos de autores: `tipo_autor`

Acto en el que un represor puede haber participado: `participacion`

Genero de una persona: `sexo`

Estado civil: `estado_civil`

País: `pais`

Provincia: `provincia`

Las distintas fuerzas armadas: `fuerza`

Estado Victima: `estado_victima`

Tipos archivos: `tipo_archivo`

Juzgados: `juzgado`

Tipo de acto procesal: `tipo_acto_procesal`

D.2. Tipos de Datos Enumerados

Estado en que se encuentran las tarea: `estado_tarea`

Importancia de las tareas: `importancia`

Bibliografía

[Amb02] S. W. Ambler. *Mapping Objects to Relational Databases*. 2002.

[BK04] C. Bauer and G. King. *Hibernate in action*. Manning Publications Co., 2th edition, 2004.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] E. F. Codd. Relational completeness of database sublanguages. In R. Rustin, editor, *Data Base Systems*. Prentice-Hall, New Jersey, 1972.

[Con11] The Unicode Consortium. The unicode standard. Technical report, Unicode Consortium, Mountain View, CA, 2011.

[EM99] Andrew Eisenberg and Jim Melton. Sql: 1999, formerly known as sql3. *SIGMOD Rec.*, 28(1):131–138, 1999.

[EMK+04] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[EN07] R. Elmasri and S.B. Navathe. *Fundamentos de Sistemas de Base de Datos*. Addison-Wesley, 5th edition, 2007.

[Fin01] Mary A. Finn. Fighting impedance mismatch at the database level. 2001.

[Gro99] Postgres Global Development Group. *Tutorial de PostgreSQL*. 1999.

[Gro14] PostgreSQL Global Development Group. *PostgreSQL 9.4beta2 Documentation*, 2014.

[jso13] *The JSON Data Interchange Format*. ECMA International, first edition edition, 2013.

[MSS01] A Mendelzon, T. Schwentick, and D Suciu. Foundations of semistructured data. 2001.

[Ora14] Oracle y/o afiliados. *MySQL 5.6 Reference Manual: Including MySQL Cluster NDB 7.3-7.4 Reference Guide*, 2014.

[Por80] Martin Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14:130–137, 1980.

[PS+05] R. C. Pare, , L. A. C. Santillan, D. Costal Costa, M. G. Ginesta, C. M. Escofet, and O. P. Mora. *Bases de datos.* Fundacio per a la Universitat Oberta de Catalunya, first edition edition, 2005.

[SKSP02] A. Silberschatz, H.F. Korth, S. Sudarshan, and F.S. Pérez. *Fundamentos de bases de datos.* McGraw-Hill, 4th edition, 2002.