Diseño de una base de datos para una *app* de mensajería instantánea

Antonio Sarasa Cabezuelo

PID_00236790

Tiempo mínimo previsto de lectura y comprensión: **6 horas**

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este material didáctico se presenta un caso práctico de utilización de una base de datos orientada a documentos. Para ello se ha elegido un dominio que es familiar para muchas personas como son los servicios de mensajería instantánea que se instalan y se utilizan en los teléfonos móviles estilo WhatsApp o Telegram.

Estos servicios de mensajería surgieron hace unos años y revolucionaron el mercado de la mensajería electrónica. Antes de la existencia de estos servicios, el envío de mensajes electrónicos a través del móvil se realizaba exclusivamente a través del envío de SMS. Este mecanismo constituía una línea de negocio de las compañías telefónicas, dado que en general el servicio no era gratuito, teniendo que pagar una cuota el cliente por el servicio o bien por SMS enviado.

La mensajería electrónica instantánea constituye un hito en este ámbito, dado que aprovechando la conexión a internet que tiene un teléfono móvil se permite el envío gratuito de mensajes sin la necesidad de utilizar el servicio de SMS ofrecido por las compañías telefónicas. La mayoría de las *apps* que han surgido en este ámbito son gratuitas o requieren el pago de una cuota anual simbólica. Con el tiempo estos servicios han ido incrementando las posibilidades con respecto a la comunicación añadiendo otras funcionalidades tales como: llamadas o el envío de fotos, vídeos, documentos, etc. La expansión ha sido tal, que se han convertido en un elemento cotidiano e indispensable de la vida digital diaria; al mismo nivel que el disponer de un número de teléfono móvil o una dirección de correo electrónico.

La información que genera y requiere gestionar el uso de este servicio es de magnitudes enormes, de gran complejidad y de carácter semiestructurado. En este sentido, y tal como se verá en el estudio del caso, el empleo de una base de datos NoSQL de tipo documental como es MongoDB permite reducir el grado de complejidad que hay que gestionar y hacerlo manejable.

Se ha elegido MongoDB como base de datos orientada a documentos debido a la gran aceptación que ha tenido en los últimos años tanto en los ámbitos académicos como industriales. Desde el punto de vista teórico, constituye un buen ejemplo de estudio dado que reúne las características más típicas de este tipo de bases de datos NoSQL. Asimismo, se trata de un producto comercial estable con actualizaciones periódicas, un rico soporte de documentación oficial en línea, *drivers* para los principales lenguajes de programación y herramientas de desarrollo complementarias. También cabría destacar la adopción de esta solución por parte de grandes compañías (eBay, McAffee, Adobe, entre

otras) e instituciones (la ciudad de Chicago, el Gobierno de Gran Bretaña...) para muy diversas necesidades, tal como puede consultarse en la propia web de MongoDB.

Objetivos

En los materiales que forman parte de este caso práctico, el estudiante encontrará las herramientas y contenidos necesarios para lograr los siguientes objetivos:

1. Realizar el proceso de diseño de una base de datos orientada a documentos desde la descripción de los datos hasta el diseño físico, pasando por el diseño conceptual y el diseño lógico.

2. Adquirir un conocimiento general sobre las bases de datos orientadas a documentos, sus principales propiedades y las ventajas que ofrecen en determinadas situaciones frente a las bases de datos relacionales.

3. Identificar en qué situaciones es más adecuado almacenar un conjunto de datos en una base de datos orientada a documentos.

4. Conocer MongoDB como ejemplo de sistema gestor de base de datos orientado a documentos.

5. Saber utilizar los comandos que ofrece MongoDB para recuperar o actualizar información almacenada.

1. Descripción del dominio

En esta sección se describirá el dominio del caso práctico que se va a desarrollar. Se presentará el contexto en el que se enmarca el caso práctico, los datos que se gestionarán y cómo se van a explotar los datos almacenados.

1.1. Contexto

Una empresa de desarrollo de *apps* quiere crear una *app* que facilite un servicio gratuito de mensajería instantánea a todos los usuarios que se instalen la *app* y se registren en el servicio.

Previamente a la implementación de la *app*, se ha realizado un estudio de mercado de este tipo de *apps* y se ha podido comprobar que todas ellas ofrecen como servicio básico el envío de mensajes entre un usuario y sus contactos. Las principales diferencias entre las diversas *apps* analizadas se deben a servicios complementarios al básico tales como llamadas telefónicas, tipos de emoticones que se pueden enviar, geolocalización de los contactos, etc.

La empresa desarrolladora ha decidido realizar la implementación de la *app* en varias fases en las que se introducirá de manera incremental nuevas funcionalidades y servicios. De esta forma, se puede analizar la respuesta por parte de los usuarios a la nueva *app*, y a partir de la retroalimentación (o *feedback*) recibida actualizar la *app*, para adaptarla así mejor a los requerimientos de los usuarios. En este sentido, se puede considerar un proyecto en continuo desarrollo y cambio.

En la primera fase del desarrollo se va a limitar a ofrecer un servicio básico de mensajería instantánea en la que un usuario podrá enviar mensajes de texto, fotos o vídeos a usuarios individuales que son contactos suyos o bien a un grupo de usuarios.

Respecto al ámbito de uso de la *app*, se ha pensado que su distribución se realizará en varias fases. De la misma forma que su implementación, se quiere analizar cómo es la acogida entre los usuarios por zonas geográficas, y en el caso de ser un éxito, entonces ir considerando otras zonas geográficas. Dado que la empresa desarrolladora tiene sede en Zaragoza, se ha pensado que en esta primera fase de desarrollo se cubra a todos los usuarios de la península ibérica. Es importante observar que este requisito constituye una limitación con respecto al uso de la *app*. Así, solo los usuarios que tengan teléfonos registrados en poblaciones de la península ibérica, podrán descargarse la *app* y registrarse en ella, por lo que no podrán formar parte de sus contactos ni comunicarse con ninguna persona que no cumpla esta condición.

El objetivo de este caso práctico es diseñar una base de datos que permita almacenar todos los datos que se necesitan gestionar en la *app* de mensajería instantánea. En la elección de la tecnología que se va a usar, así como en el diseño propiamente de la base de datos, se prestará especial interés a los requisitos referidos tanto al tipo de explotación de datos que se va a llevar a cabo como a las necesidades futuras que se tendrán en las sucesivas versiones que se van a desarrollar de la *app*.

1.2. Datos gestionados

Los siguientes apartados describen los datos que la *app* va a necesitar gestionar y almacenar en la primera fase de desarrollo. En esta fase se permitirá al usuario enviar mensajes a usuarios individuales que formen parte de sus contactos o bien a un grupo de usuarios. El contenido de los mensajes que pueden enviarse se limita a texto, fotos o vídeos. Un usuario podrá bloquear a otros usuarios y ser bloqueado por otros usuarios. Así pues, para esta primera versión de la *app*, se requiere almacenar información acerca de los usuarios, las relaciones que existen entre los usuarios y de los mensajes que se envían.

1.2.1. Usuario

Para ser usuario de la *app* será necesario bajarse la *app*, instalarla en el teléfono móvil y registrarse. El registro requerirá que el usuario introduzca una serie de datos personales: *nombre*, *apellidos*, *teléfono móvil utilizado para el registro*, *email de contacto*, *edad*, *ciudad* y *país de residencia*. Los cuatro primeros datos serán obligatorios para darse de alta, y los restantes optativos. Asimismo, la *app* asignará al usuario un identificador único en el momento del registro. Cuando un usuario se registre, antes de poder utilizar su cuenta, recibirá en el correo electrónico indicado en el registro un mensaje de confirmación. El usuario deberá pulsar sobre el mismo para finalizar la fase de registro y poder utilizar el servicio. En ese momento se producirá la creación de la cuenta del usuario. Además de almacenar los datos de registro mencionados, el sistema añadirá otros campos de información: *fecha* e *instante de creación*.

Para añadir o modificar la información propia del usuario, la *app* ofrecerá el enlace «Modificar perfil de usuario» que, al pulsarlo, conducirá al usuario a una ventana en la que le aparecerán sus datos personales y podrá modificarlos. El único dato que no podrá modificar será su número de teléfono móvil.

Cabe observar que de cara al almacenamiento y registro de usuarios, el número de teléfono móvil del usuario servirá como identificador del usuario. Es por ello que un usuario podría tener más de una cuenta en la *app* siempre y cuando utilizara para ello dos números de teléfono y dos terminales diferentes. La única restricción es que un mismo terminal no podrá albergar más de una

cuenta de usuario dado que el registro de la cuenta comprobará que el número introducido por el usuario coincide con el número de teléfono de la tarjeta SIM que se encuentra en el terminal del teléfono móvil.

1.2.2. Relaciones entre usuarios

Un usuario podrá establecer diversas relaciones con el resto de usuarios de la *app* que se van a describir a continuación.

Cualquier usuario puede añadir a otro usuario como contacto propio. Para ello la *app* ofrecerá una opción de «Añadir contacto». Cuando el usuario pulse sobre dicha opción, le aparecerá un formulario en el que introducirá el teléfono del usuario que quiere añadir como contacto. La operación se realizará con éxito siempre que el usuario esté registrado en la *app* y no exista un bloqueo entre ambos usuarios en cualquiera de los sentidos. Una vez que un usuario es contacto de otro usuario, se pueden enviar mensajes entre ellos y formar parte de grupos pertenecientes a uno u otro usuario. Cuando un usuario se convierte en contacto de otro se almacenará información acerca de la *fecha y hora de alta como contacto*. Mientras que un usuario no es contacto del otro no es posible el envío de mensajes entre ambos. Si un usuario que se trata de añadir como contacto ya lo fuera, entonces la acción de añadir no produciría efecto alguno. Entre las opciones que ofrecerá la *app* se encontrará la de visualizar a todos los contactos que tiene un usuario en un momento dado.

La *app* también ofrecerá la posibilidad de eliminar un contacto. Para ello, el usuario deberá dirigirse al listado de contactos y buscar el contacto que desea eliminar. Una vez situado sobre el contacto, al presionar sobre el contacto le aparecerá una ventana de confirmación. Al pulsar sobre «Aceptar» en dicha ventana, el contacto será eliminado de la lista de contactos del usuario, así como de todos los grupos de usuarios en los que apareciera el usuario eliminado y que fueran propiedad del usuario que realiza la eliminación. Además, la eliminación del contacto producirá la eliminación de todos los mensajes que se hubieran intercambiado entre ambos usuarios.

Tal como se ha mencionado, un usuario puede bloquear a otros usuarios y puede ser bloqueado por otros usuarios. Para bloquear a un usuario, la *app* ofrecerá la opción de «Bloquear usuario». Cuando el usuario pulse sobre dicha opción, le aparecerá un formulario en el que introducirá el teléfono del usuario que quiere bloquear. La operación se realizará con éxito siempre que el usuario estuviera registrado en la *app*, y no existiera un bloqueo previo. Si existe un bloqueo previo, entonces la acción de bloquear no produciría efecto alguno. Si el bloqueo se realiza con éxito, entonces el usuario bloqueado no podrá añadir al usuario bloqueante como contacto suyo, y por lo tanto no se podrán intercambiar mensajes entre los mismos. Además, se almacenará información acerca de la *fecha y hora del bloqueo del usuario*.

Cabe observar que si el usuario bloqueado era un contacto, entonces desaparecerá como contacto del usuario y será eliminado de todos los grupos de los que formará parte y fueran propiedad del usuario. Además, en los grupos que no son propiedad del usuario pero en los cuales se encuentren los dos usuarios, la *app* impedirá que ambos usuarios puedan comunicarse ni ver los mensajes que envía cualquiera de los dos al grupo común. Asimismo, el bloqueo de un contacto producirá la eliminación de todos los mensajes que se hubieran intercambiado entre ambos usuarios.

También existirá la posibilidad de desbloquear a un usuario; para ello la *app* ofrece la opción de «Desbloquear usuario». Cuando el usuario pulse sobre dicha opción, le aparecerá un formulario en el que introducirá el teléfono del usuario que quiere desbloquear. La operación se realizará con éxito siempre que el usuario estuviera registrado en la *app* y existiera un bloqueo previo. En caso de no estar registrado en la *app* o no estar bloqueado, la acción no producirá ningún efecto. Además, se almacenará información acerca de la *fecha y hora del desbloqueo del usuario*.

Por último, un usuario podrá crear grupos de usuarios privados con ciertos usuarios que forman parte de sus contactos. Para ello, la *app* ofrecerá la opción «Crear grupo de contactos». Cuando el usuario pulse sobre dicha opción, le aparecerá un formulario en el que podrá introducir un nombre para el grupo de usuarios que va a crear, y a continuación aparecerá un campo en el que podrá ir introduciendo el nombre o teléfono de los contactos que quieren que formen parte del grupo. Una vez introducidos el *nombre del grupo* y los *contactos*, pulsará sobre un botón de «Aceptar» que dará de alta al grupo. Cuando se crea un grupo se almacena información adicional acerca del grupo: *identificador del grupo* (que será asignado por la *app*), *fecha y hora de creación*, y *teléfono móvil del propietario del grupo*. Los contactos que han sido añadidos como miembros de un grupo, recibirán un mensaje de aceptación de manera que puedan confirmar que quieren pertenecer al grupo o bien declinar la inclusión.

De la misma forma que ocurre con los contactos, un usuario podrá acceder a un listado con todos los grupos que ha creado o de los que forma parte pero son propiedad de otros usuarios. Asimismo, un usuario podrá darse de baja de un grupo del que es miembro. Para ello deberá dirigirse al listado de grupos y buscar el grupo del que desea darse de baja. Una vez situado sobre el grupo, al presionar sobre el mismo le aparecerá la opción «Darse de baja». Si confirma la baja, entonces el usuario dejará de ser miembro del grupo y el nombre del grupo desaparece del listado de grupos de los que es miembro. En este proceso solo hay una excepción, y es que los propietarios de grupos no pueden darse de baja de los grupos que son de su propiedad.

El propietario de un grupo solo podrá modificar las características de un grupo en cuanto al nombre del grupo y a los integrantes del mismo, o bien eliminar el grupo. Para ello el propietario deberá dirigirse al listado de grupos, y buscar el grupo que desea actualizar o eliminar. Una vez situado sobre el grupo, al pre-

sionar sobre el mismo le aparecerán dos opciones: «Eliminar» o «Modificar». Si pulsa sobre opción de «Eliminar», el grupo será eliminado, desapareciendo de la lista de grupos del usuario y de todos los usuarios que formaban parte del grupo. Si pulsa sobre la opción de «Modificar», le aparecerá una ventana con toda la información del grupo, la cual podrá actualizar: cambiar el nombre del grupo, y añadir o eliminar integrantes. En el caso de la actualización, los mensajes que se han intercambiado en el grupo permanecerán inalterados, y en el caso de la eliminación del grupo, todos los mensajes asociados al grupo también serán eliminados.

Un usuario podrá intercambiarse mensajes con cualquiera de los usuarios que forman parte de sus contactos, pero nunca podrá hacerlo con un usuario que no sea un contacto previamente. También podrá intercambiar mensajes con todos los grupos de los que forme parte, con independencia de que sea propietario o no del grupo. Los mensajes enviados se almacenarán tanto en la cuenta del emisor como en la cuenta del receptor. Cabe observar que en el caso del envío de un mensaje a un grupo, se considerará al grupo como un único destinatario (y no como múltiples destinatarios) que recibe un único mensaje (aunque existan copias del mensaje en las cuentas de cada uno de los usuarios que forman parte del grupo).

Los mensajes, una vez que han sido enviados, no pueden ser eliminados de la cuenta del destinatario con independencia de que el mensaje haya sido enviado a un usuario individual o a un grupo. Por otra parte, cuando se añade un nuevo usuario a un grupo, este podrá visualizar solo los mensajes que se intercambien en el grupo desde el momento en que produzca su alta. Por último, cuando un usuario deja de ser contacto de otro usuario o bien deja de ser miembro de un grupo, mantendrá en su cuenta todos los mensajes que se han intercambiado hasta ese momento.

1.2.3. Mensajes enviados

Respecto a los mensajes que son intercambiados, pueden ser de tres categorías, según el tipo del contenido que es intercambiado. Así, se podrán enviar mensajes constituidos únicamente de texto plano. No existirá una longitud máxima para el texto enviado. El texto no podrá tener asociado ningún tipo de estilo o complemento (es decir, no se podrá escribir en negrita, cursiva, color...). Solo detectará como texto especial cuando en el texto aparezca una URL, la cual se mostrará de un color azulado, y permitirá pulsar sobre el enlace para que este sea abierto en un navegador web que esté disponible en el teléfono móvil.

Otro tipo de mensajes son los que incluyen fotos. Para poder intercambiar una foto, será necesario que la foto se encuentre en el teléfono móvil, ya sea porque se trata de una foto que ha realizado el usuario con su teléfono y la tiene almacenada, o bien porque el usuario la ha recibido previamente en un mensaje emitido por otro usuario y se encuentra en la *app*.

Por último, se podrán enviar mensajes que contengan vídeo. Este caso se gestiona de forma similar a las fotos. En este sentido, se podrán enviar vídeos realizados por el usuario con el teléfono móvil y por lo tanto se encuentren ya almacenados en el propio teléfono, o bien vídeos que le han sido enviados por otros usuarios y se encuentran en la *app*.

Para enviar un mensaje de cualquiera de los tipos, el usuario deberá seleccionar de la lista de contactos al contacto al que desea enviar el mensaje, o bien de la lista de grupos a los que pertenece, al grupo al que desea enviar el mensaje. Una vez seleccionado, le aparecerá una ventana donde podrá introducir el texto del mensaje y/o seleccionar la foto y/o vídeo que desea enviar. A continuación, pulsará sobre «Enviar», realizándose el envío. Los mensajes enviados o recibidos quedan asociados al contacto o grupo con el cual se intercambiaron.

Con respecto al reenvío de mensajes, bastará situarse en un contacto o grupo y seleccionar el mensaje que se quiere reenviar. Una vez seleccionado al presionar sobre el mismo, aparecerá la opción de reenviar. Cuando se pulse, mostrará una nueva ventana en la que se podrá seleccionar los contactos o grupos al que se quiere reenviar el mensaje. A continuación se pulsará sobre «Reenviar», realizándose el reenvío del mensaje.

A cada mensaje enviado se le asocia un conjunto de campos de información: *fecha y hora de publicación*. Asimismo, se almacenan otros datos representativos del mensaje según la categoría a la que pertenezca el mensaje. Si se trata de un mensaje de texto, se guardará la *longitud* del mismo. Si se trata de una foto se guardará la *dimensión*, *fecha de realización*, *calidad*, *formato*, *tamaño* y *título* que tenga la foto. Y si se trata de un vídeo, se almacenará *fecha de realización*, *calidad*, *duración*, *temática*, *formato* y *título del vídeo*. No se puede garantizar que algunos de los datos mencionados se puedan recuperar en todas las fotos o vídeos, pues pueden que existan o no (estos datos se capturan de los metadatos que se encuentran asociados a los propios recursos), y además puede que en versiones futuras de la *app* se guarden otros campos de información diferentes a los mencionados con fines estadísticos.

Por otro lado, los mensajes se almacenarán en el teléfono móvil del usuario y del destinatario, por lo que ocuparán memoria del terminal. Así, el límite máximo de mensajes que podrán intercambiarse vendrá dado por la cantidad de memoria disponible en el mismo. En este sentido, un usuario podrá eliminar los mensajes que se ha intercambiado, pero esta eliminación solo afectará a la

copia de los mensajes que se encuentran en su cuenta, pero no así a la copia de los mensajes que se encuentran en la cuenta del destinatario (estos serán gestionados por ese usuario).

Hay que tener en cuenta que en esta fase solo se han contemplado tres tipos de mensajes; sin embargo, en fases posteriores, las versiones de la *app* podrán intercambiar otro tipo de mensajes tales como información de un contacto, información de geolocalización, documentos de diferentes tipos tales como pdf, Word, etc. Es por ello que se deberá tener en cuenta las futuras ampliaciones de la *app* para realizar el diseño de la misma, así como de la base de datos que almacenará la información gestionada.

1.3. Explotación de los datos

De toda la información descrita en el apartado anterior, la empresa solo podrá gestionar una parte de la misma, dado que hay parte de la información descrita que tiene un carácter privado y que solo se almacenará en la memoria de los terminales de los usuarios registrados. En este sentido, la empresa podrá gestionar la siguiente información de cada usuario:

- Toda la información acerca de la cuenta personal de un usuario.
- Toda la información acerca de los grupos a los que pertenece un usuario con independencia de si es propietario del grupo o si solo es miembro invitado.
- Toda la información acerca de quiénes son los contactos de un usuario.
- Toda la información acerca de los usuarios bloqueados y desbloqueados por un usuario.
- Toda la información acerca de los mensajes intercambiados por un usuario con sus contactos o con los grupos a los que pertenece, incluyendo la información propia del tipo al que pertenece: vídeos, fotos o texto.

Una vez que se ha fijado la información a la que tendrá acceso la empresa, se va a describir algunas formas de explotación de la información almacenada. Se van a enunciar agrupados por categorías, y en cada categoría se mostrarán a modo de ejemplo algunas de las preguntas que se podrían plantear (no se trata de una lista exhaustiva).Todas las preguntas se plantean sobre un periodo de tiempo referido a un mes de un año concreto.

1.3.1. Actividad en el servicio de mensajería

Se podría responder a preguntas que ofrecieran una visión general sobre la actividad que se genera en el servicio tales como:

- ¿Cuál es la media de mensajes enviados por los usuarios a lo largo de un mes de un año concreto?

- ¿Cuál es la media de mensajes enviados en los grupos de usuarios a lo largo de un mes de un año concreto?

- ¿Cuántos usuarios nuevos se crean a lo largo de un mes de un año concreto? ¿Y cuántos grupos?

- ¿Cuál fue el usuario que más mensajes envió? ¿Y el usuario que menos mensajes envió?

- A lo largo de un mes de un año concreto, ¿cuál fue el día con más actividad en cuanto al envío de mensajes y registro de usuarios? ¿Y el que menos?

1.3.2. Perfil del comportamiento de los usuarios

Se podría responder a preguntas que ofrecieran información acerca del comportamiento que tiene un usuario concreto tales como:

- A lo largo de un mes de un año concreto, ¿con qué usuario mantiene un mayor intercambio de mensajes el usuario? ¿Y con qué grupo?

- ¿Cuántos mensajes ha enviado el usuario a lo largo de un mes de un año concreto?

- ¿En qué fecha de un mes de un año concreto ha enviado más mensajes el usuario?

- ¿Cuál es el usuario que más mensajes envía al usuario? ¿Y el que menos?

- ¿Cuál es el tipo de mensaje (vídeo, foto o texto) que más envía el usuario a lo largo de un mes de un año concreto?

- ¿Cuántos usuarios ha creado el usuario a lo largo de un mes de un año concreto? ¿Y grupos?

- ¿Cuál es la media por día de mensajes que envía el usuario a lo largo de un mes de un año concreto?

- ¿Cuál es el número de bloqueos realizados por el usuario a lo largo de un mes de un año concreto? ¿Y desbloqueos?

1.3.3. Información sobre los mensajes enviados

Se podría responder a preguntas que ofrecieran información acerca de las características de los mensajes que son enviados tales como:

- ¿Cuál es el tipo de mensaje (vídeo, foto o texto) más enviado por los usuarios a lo largo de un mes de un año concreto?
- ¿Cómo son las fotos que envían los usuarios a lo largo de un mes de un año concreto (calidad, formato, dimensiones, etc.)?
- ¿Cómo son los vídeos que envían los usuarios a lo largo de un mes de un año concreto (calidad, formato, duración, etc.)?
- ¿Cuál es la longitud media de los textos que envían los usuarios a lo largo de un mes de un año concreto?
- ¿Cuál es el usuario que más fotos envía a lo largo de un mes de un año concreto? ¿Y vídeos? ¿Y mensajes de texto?
- ¿Cuál es el grupo que más fotos envía a lo largo de un mes de un año concreto? ¿Y vídeos? ¿Y mensajes de texto?

1.3.4. Información sobre los bloqueos que se realizan

Se podría responder a preguntas que ofrecieran información acerca de los bloqueos que se producen entre los usuarios registrados tales como:

- A lo largo de un mes de un año concreto, ¿cuál es la media de bloqueos que se producen? ¿Y la media de desbloqueos?
- A lo largo de un mes de un año concreto, ¿cuál es el usuario que más bloqueos ha realizado? ¿Y cuál es el usuario que más ha sido bloqueado por otros usuarios?
- ¿Cuántos usuarios bloqueados han sido desbloqueados a lo largo de un mes de un año concreto?
- ¿A qué ciudad/es pertenecen la mayoría de los usuarios que han sido bloqueados a lo largo de un mes de un año concreto? ¿Y la ciudad/es que menos usuarios han sido bloqueados?
- ¿A qué ciudad/es pertenecen la mayoría de los usuarios que han sido desbloqueados? ¿Y la ciudad/es que menos usuarios han sido desbloqueados?

1.3.5. Información sobre los grupos

Se podría responder a preguntas que ofrecieran una visión general sobre los grupos que existen en el servicio de mensajería tales como:

- ¿Cuál es el grupo que más mensajes ha intercambiado a lo largo de un mes de un año concreto?

- ¿Cuál es el tipo de mensajes que más se intercambian los usuarios de los grupos a lo largo de un mes de un año concreto?

- A lo largo de un mes de un año concreto, ¿cuántos grupos se crearon? ¿Y cuántos fueron eliminados?

- A lo largo de un mes de un año concreto, ¿a cuántos grupos en media pertenecen los usuarios?, ¿y en media cuántos grupos han sido creados por los usuarios?

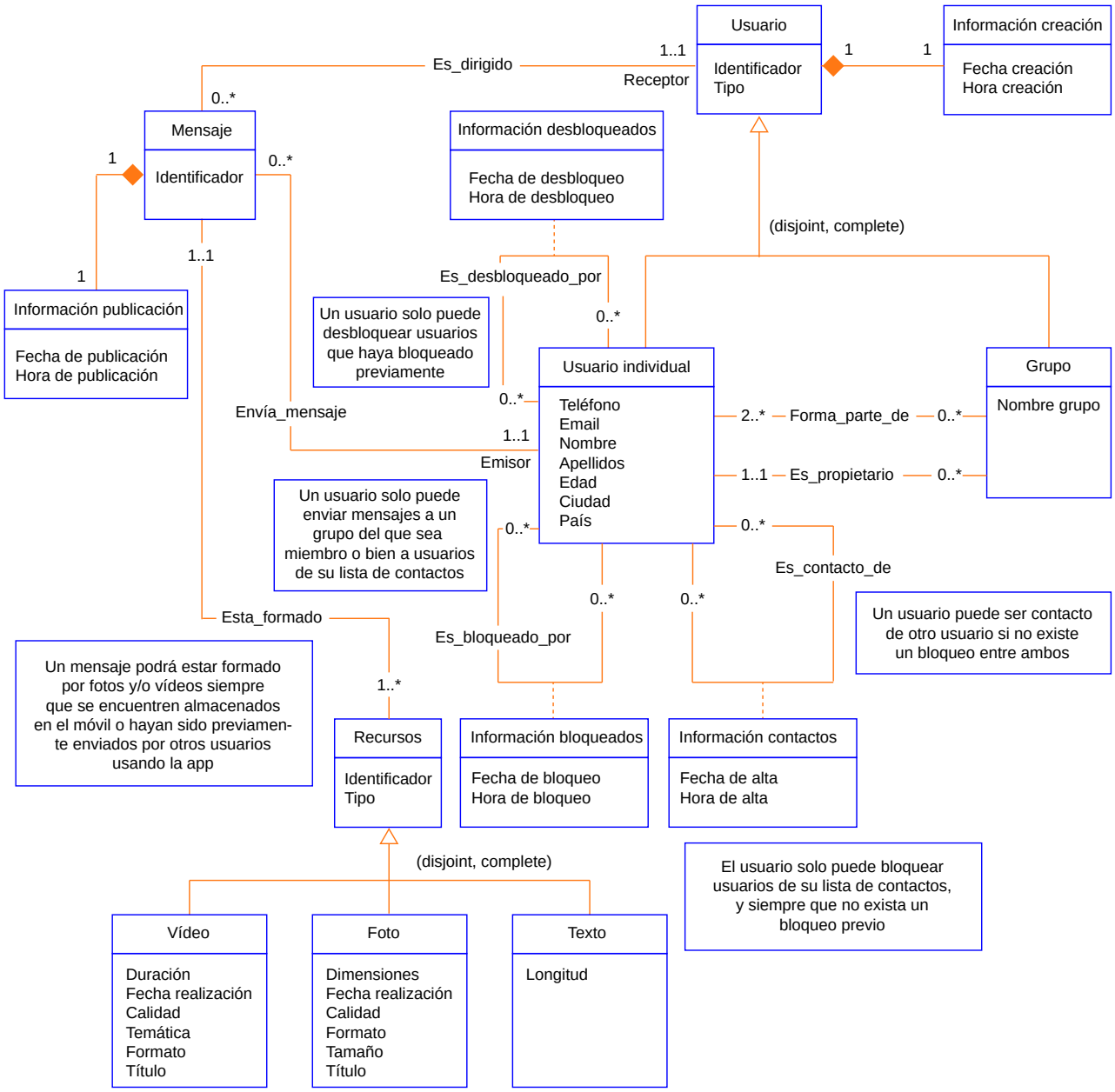
- A lo largo de un mes de un año concreto, ¿cuál es el grupo que más usuarios tiene? ¿Y el que menos?

- A lo largo de un mes de un año concreto, ¿se pueden crear agrupaciones de grupos por la zona geográfica de los miembros que lo componen?

2. Diseño conceptual

Sobre la base del tipo de explotación de datos que se va a realizar y de la descripción del caso de uso realizada, se plantea el siguiente diagrama conceptual en forma de diagrama de clases UML. Se debe observar que el diagrama presentado en la figura 1 no es único, y se podría diseñar otros diagramas igualmente válidos.

Figura 1. Modelo conceptual para la aplicación de mensajería instantánea



Existen dos clases principales en el diagrama, la clase *Usuario* y la clase *Mensaje*, que representan respectivamente a los distintos tipos de usuarios que gestiona la *app* y a los mensajes que se intercambian los usuarios de la *app*.

La clase *Usuario* posee dos atributos. El atributo *identificador* que permite identificar cada instancia de esta clase, y el atributo *tipo* que permitirá identificar a las instancias de las subclases que derivan de cada una de las clases. En este sentido, la clase *Usuario* se especializa en dos subclases, la subclase *Grupo* y la subclase *Usuario individual*, que representan respectivamente a un grupo de usuarios y a un usuario individual. La especialización es disjunta dado que no puede haber instancias que pertenezcan a la vez a ambas subclases, y completa dado que no se considera ninguna otra posible subclase además de las mencionadas. Por otra parte, para almacenar la información de creación de un *Usuario* (fecha y hora de creación), se ha definido una clase denominada *Información creación* que mantiene una relación de composición con la clase *Usuario*, es decir, cada información de creación se compone de 1 y solo 1 usuario, y cada usuario solo tiene asociada una información de creación.

Entre las subclases *Grupo* y *Usuario individual* se definen dos relaciones de tipo asociación: *Forma_parte_de* y *Es_propietario*. La relación *Forma_parte_de* representa que un grupo está formado por 2 o más (2..*) usuarios (no se consideran grupos con menos de 2 usuarios) y que un usuario individual puede pertenecer a 0 o más (0..*) grupos (no es necesario que un usuario forme parte de ningún grupo). La relación *Es_propietario* representa que un usuario es propietario de 0 o más (0..*) grupos y que un grupo tiene uno y solo un propietario (1..1).

En la clase *Grupo* se han definido el atributo *Nombre del grupo* que representa al nombre que se he dado a un grupo cuando se realiza su creación.

La clase *Usuario individual* dispone de los atributos *Teléfono*, *Email*, *Nombre*, *Apellidos*, *Edad*, *Ciudad* y *País*. El atributo *Teléfono* representa el teléfono móvil del usuario que ha introducido en el momento de realizar el registro en la *app*. El atributo *Email* representa el email del usuario que ha introducido en el momento de realizar el registro en la *app*. Los atributos *Nombre* y *Apellidos* representan el nombre y apellidos del usuario. Los atributos *Edad*, *Ciudad* y *País* representan la edad, la ciudad y país del usuario.

En la clase *Usuario individual* se definen tres relaciones de tipo asociación reflexiva o recursiva: *Es_bloqueado_por*, *Es_desbloqueado_por* y *Es_contacto_de*. La relación *Es_bloqueado_por* representa el hecho de que un usuario individual puede bloquear a 0 o más (0..*) usuarios o ser bloqueado por 0 o más (0..*) usuarios. Para almacenar la información de creación de un bloqueo (fecha y hora de bloqueo) se ha definido una clase asociativa denominada *Información bloqueados*. Se ha representado mediante una restricción textual, la restricción de integridad que indica que *un usuario solo puede bloquear a usuarios que pertenecen a su lista de contactos, y siempre que no exista un bloqueo previo*. La relación *Es_desbloqueado_por* representa el hecho de que un usuario individual puede

desbloquear a 0 o más (0..*) usuarios o ser desbloqueado por 0 o más (0..*) usuarios. Para almacenar la información de la realización de un desbloqueo (fecha y hora del desbloqueo) se ha definido una clase denominada *Información desbloqueados*, y se ha representado mediante una restricción textual, la restricción de integridad que indica que *un usuario solo puede desbloquear a usuarios que haya bloqueado previamente*. La relación *Es_contacto_de* representa el hecho de que un usuario individual puede tener 0 o más (0..*) usuarios de contacto o ser contacto de 0 o más (0..*) usuarios. Para almacenar la información de creación de un contacto (fecha y hora de alta) se ha definido una clase denominada *Información contactos*, y se ha representado mediante una restricción textual, la restricción de integridad que indica que *un usuario puede ser contacto de otro usuario si no existe un bloqueo entre ambos*.

La clase *Mensaje* posee el atributo *identificador* que permite identificar cada instancia de esta clase. Para almacenar la información de creación de un mensaje (fecha y hora de publicación) se ha definido una clase denominada *Información publicación* que mantiene una relación de composición con la clase *Mensaje*, es decir, cada información de publicación se compone de 1 y solo 1 mensaje y cada mensaje solo tiene asociada una información de publicación.

Entre las clases *Mensaje* y *Usuario individual* se define la relación *Envía_mensaje* de tipo asociación, que representa que un usuario individual actúa como emisor de 0 o más (0..*) mensajes, y que un mensaje es enviado por un único emisor (1..1). También entre las clases *Mensaje* y *Usuario* se define la relación *Es_dirigido* de tipo *asociación*, que representa que un *Usuario* puede recibir 0 o más (0..*) mensajes, y que un mensaje está dirigido a un único receptor (1..1). Tal como aparece en la descripción del caso, el envío de un mensaje a un grupo se considera como el envío de un único mensaje. Además, se ha representado mediante una restricción textual, la restricción de integridad que indica que *un usuario solo puede enviar mensajes a un grupo del que sea miembro o bien a usuarios de su lista de contactos*.

Por último, se tiene la clase *Recursos*, la cual representa el tipo de mensaje que puede ser enviado. Así entre la clase *Mensaje* y *Recursos* se establece la relación *Esta_formado_por* que representa que un mensaje está formado por uno o más recursos (1..*).

La clase *Recursos* se especializa en las subclases *Vídeo*, *Foto* y *Texto* que representan respectivamente a los diferentes tipos de recursos, es decir vídeos, fotos y textos. La especialización es disjunta dado que no puede haber instancias que pertenezcan a la vez a más de una subclase, y completa dado que no se considera ninguna otra posible subclase además de las mencionadas.

La clase *Vídeo* está formada por los atributos *Duración*, *Fecha realización*, *Calidad*, *Temática*, *Formato* y *Título*. El atributo *Duración* representa la duración del vídeo en alguna unidad temporal estándar. El atributo *Fecha realización* representa la fecha cuando se realizó el vídeo. El atributo *Calidad* representa

la calidad que tiene el vídeo. El atributo *Temática* representa un conjunto de palabras clave que indican la temática del vídeo. El atributo *Formato* indica el formato en el que se encuentra el vídeo (avi, mp4, wm,..). Por último, el atributo *Título* representa el título del vídeo.

La clase *Foto* está formada por los atributos *Dimensiones*, *Fecha realización*, *Calidad*, *Formato*, *Tamaño* y *Título*. El atributo *Dimensiones* representa las dimensiones de la foto en alguna unidad válida. El atributo *Fecha realización* representa la fecha cuando se realizó la foto. El atributo *Calidad* representa la calidad que tiene la foto. El atributo *Formato* indica el formato en el que se encuentra la foto. Por último, el atributo *Título* representa el título de la foto.

La clase *Texto* está formada por el atributo *Longitud* que representa la longitud del texto.

Los atributos de estas clases se obtienen de los metadatos que vienen asociados con cada uno de estos recursos; es por ello que no está garantizado que puedan estar disponibles para todos los recursos, y en muchos casos aparezcan vacíos.

3. Alternativas tecnológicas

Teniendo en cuenta los requisitos especificados en los anteriores apartados, se van analizar los diferentes tipos de bases de datos que podrían utilizarse para implementar la base de datos.

En primer lugar, se van a destacar una serie de afirmaciones que aparecen en la descripción del dominio:

- «La empresa desarrolladora ha decidido realizar la implementación de la *app* en varias fases en las que se introducirá de manera incremental nuevas funcionalidades y servicios [...] En este sentido se puede considerarse un proyecto en continuo desarrollo y cambio.»

- «Respecto al ámbito de uso de la *app*, se ha pensado que su distribución se realizará en varias fases [...] se ha pensado que en esta en esta primera fase de desarrollo se cubra a todos los usuarios de la península ibérica.»

- «No se puede garantizar que algunos de los datos mencionados se puedan recuperar en todas las fotos o vídeos, pues pueden que existan o no (estos datos se capturan a su vez de los metadatos que aparecen asociados a estos recursos), y además puede que en versiones futuras de la *app* se guarden otros metadatos diferentes a los mencionados con fines estadísticos.»

- «Hay que tener en cuenta que en esta fase solo se han contemplado tres tipos de mensajes; sin embargo, en fases posteriores, las versiones de la *app* podrán intercambiar otro tipo de mensajes tales como información de un contacto, información de geolocalización, documentos de diferentes tipos tales como pdf, Word...»

- «Existe un catálogo prefijado de formas de explotar la información que se desea almacenar.»

De estas premisas se puede deducir que la base de datos que mejor encajaría debería cumplir:

a) Sin un esquema de definición de datos fijo. Las restricciones resaltadas anteriormente indican que se trata de un proyecto en continuo desarrollo y cambio, por lo que es posible que los datos que se gestionen cambien con el tiempo. Se menciona que en futuras versiones, además de gestionar los tres tipos de mensajes actuales (vídeo, foto, texto), se añadirán nuevos tipos para los cuales habrá que almacenar nueva información característica de los nuevos tipos.

b) Datos semiestructurados no regulares. Se menciona que la información que se quiere almacenar de los tipos de mensajes puede que no esté completa y haya campos de información que estén vacíos. Además, es probable que se cambien los datos almacenados por otros diferentes con fines estadísticos.

c) Elevada concurrencia de accesos. Considerando la naturaleza de la aplicación, una *app* para mensajería instantánea, se tiene que la cantidad de usuarios que accederán al servicio al mismo instante tratando de realizar consultas y enviar mensajes de forma simultánea será enorme. Es por ello que sería necesario que el sistema de persistencia estuviera preparado para dar respuesta a una elevada concurrencia de accesos.

d) Enorme cantidad de datos. Como consecuencia del requisito anterior, la cantidad de datos que se va a generar y que va a ser necesario gestionar será de dimensiones gigantescas. Por esta razón, sería necesario disponer de un sistema de persistencia que sea capaz de gestionar y manipular de una forma eficiente tales cantidades de datos y que, además, se pueda escalar con facilidad según vaya aumentando el volumen de datos almacenados.

e) Explotación de datos definida. Tal como se ha expuesto en uno de los apartados anteriores, la forma en la que se van a explotar los datos es conocida de antemano, de manera que la información se puede estructurar previamente para facilitar la forma en la que se quiere acceder a la misma. No se va a realizar otro tipo de consultas diferentes a las planificadas.

f) Existencia de relaciones simples. Las relaciones que existen entre los datos no son demasiado complejas ni existen demasiadas.

g) Distribución de los datos. Al tratarse de una aplicación que se va a distribuir en diferentes zonas geográficas, sería fácil llevar a cabo una fragmentación de la información de acuerdo a estas zonas geográficas, de forma que los usuarios de una zona determinada puedan acceder con mayor rapidez a sus datos.

Una vez analizadas las características requeridas por el sistema de persistencia, se van a considerar diferentes alternativas de almacenamiento empezando por las bases de datos relacionales. En primer lugar, la necesidad de tener un esquema de datos previamente definido es incompatible (Mohamed y otros, 2014) con la característica a) que indica que sería mejor no disponer de una estructura fija dado que es probable que esta cambie. La característica b) también supone un problema, dado que si se usara una base de datos relacional, las tablas tendrían filas con muchas columnas llenas de valores nulos al no existir. En este mismo sentido, la posibilidad de tener que cambiar los campos de información que se requieren almacenar o la adicción de nuevos campos haría necesario realizar cambios en las tablas o introducir soluciones muy particulares y poco eficientes para poder adaptarse a los cambios de requisitos de almacenamiento que fueran surgiendo.

Referencia bibliográfica

M. A. Mohamed; O. G. Altrafi; M. O. Ismail (2014). «Relational vs. NoSQL Databases: A Survey». *International Journal of Computer and Information Technology* (vol. 3, núm. 3).

Por otro lado, la base de datos requiere soportar una cantidad enorme de accesos concurrentes y la gestión de cantidades inmensas de datos, situación que podría no solucionarse de una forma muy eficiente un sistema gestor de bases de datos relacional que, por lo general, se orienta a un entorno centralizado, y cuya capacidad de procesamiento depende directamente de la máquina en la que se ejecuta. De manera que para conseguir una eficiencia adecuada, habría que realizar un escalado vertical adquiriendo sucesivamente máquinas más potentes. En este caso, es más eficiente realizar un escalado horizontal trabajando sobre *clusters* de máquinas más económicas.

Asimismo, se menciona la necesidad de distribuir los datos geográficamente dado que la *app* se va a utilizar en diferentes áreas geográficas. Esto refuerza la necesidad de trabajar en un entorno distribuido. Como se ha dicho anteriormente, una base de datos relacional tiene como modo de trabajo preferido el centralizado. En este sentido, la distribución, fragmentación y replicación de los datos, que son técnicas relacionadas con bases de datos distribuidas, pueden ser problemáticas en una base de datos relacional dado que no es sencillo acometer un buen diseño de distribución del esquema de datos y esto afectaría al rendimiento global del sistema. Por ejemplo, quizá no se puede conocer *a priori* qué filas y qué tablas van a ser necesarias consultar en cada momento (esta situación podría ser aún más problemática si fuera necesario ejecutar operaciones de combinación sobre tablas que están distribuidas), de manera que posiblemente no es una solución adecuada.

Por último, se menciona que se conoce *a priori* la forma en la que se va a realizar la explotación de los datos. Las bases de datos relacionales organizan el esquema de la base de datos de forma agnóstica. Esto significa que no organizan la información para realizar un tipo de consultas concretas, sino que se orientan a poder dar respuesta a cualquier tipo de consulta que se pueda formular sobre la base de datos. A pesar de ello, sí que existen mecanismos que pueden ayudar a resolver eficientemente ciertos tipos de consultas concretas que se sabe *a priori* que se van a formular, por ejemplo, pueden definir vistas materializadas. En cualquier caso, se trata de un mecanismo que se tiene que utilizar con mesura (o al menos esa es la recomendación) dado los costes de mantenimiento que tienen asociadas.

Así pues, se ha visto que muchas de las características anteriores son incompatibles (o de difícil cumplimiento) con los requisitos necesarios para poder usar una base de datos relacional, razón por la que se desecha esta posibilidad.

La alternativa a las bases de datos relacionales son las bases de datos NoSQL (Redmond y Wilson, 2012). Dentro de este tipo de bases de datos, se pueden considerar dos modelos (Moniruzzaman y Hossain, 2013), el orientado a grafos y el orientado a agregados.

Referencias bibliográficas

A. B. M. Moniruzzaman; S. A. Hossain (2013). «NoSQL Database: New Era of Databases for Big Data Analytics-Classification, Characteristics and Comparison». *ArXiv preprint arXiv*: 1307.0191.

E. Redmond; J. R. Wilson (2012). *Seven Databases in Seven Weeks: a Guide to Modern Databases and the NoSQL Movement*. Raleigh, NC: Pragmatic Bookshelf.

Se va a analizar en primer lugar el modelo orientado a grafos (Vaish, 2013). En este caso, algunas de las incompatibilidades que mostraba el modelo relacional no se encuentran en el modelo orientado a grafos tales como la necesidad de tener un esquema de datos previamente definido o la manipulación de datos semiestructurados. Sin embargo, sí hay algún requisito que hace que no sea una buena elección. Tal como se expresa en el requisito f), las relaciones entre datos son simples y no numerosas; en cambio, un escenario de uso adecuado para una base de datos orientada a grafos es la existencia de relaciones muy complejas y numerosas. Por otro lado, los requisitos c), d) y g) indican la necesidad de realizar al menos una fragmentación horizontal de los datos, realizando su ejecución en un *cluster* de máquinas. Esta necesidad hace incompatible el uso de una base de datos orientada a grafos para el caso que se quiere implementar, dado que este tipo de bases de datos no están pensadas para ejecutarse en entornos distribuidos puesto que no se puede conocer *a priori* qué partes de un grafo se podrían ejecutar en cada elemento del *cluster* asegurándose de que no será necesario tener juntas esas partes del grafo para realizar las consultas. Asimismo, otra característica propia de este tipo de bases de datos es la existencia de poca información en cada uno de los nodos del grafo dado que el principal elemento de información del grafo es las relaciones que existen entre los datos, y que vienen representadas por la estructura del grafo. Por último, también resulta poco adecuada esta elección por la enorme cantidad de accesos concurrentes que se van a realizar puesto que, en términos de un grafo, se traducirán en recorridos sobre el mismo, que pueden ser bastantes costosos dada la naturaleza propia del grafo y los métodos de búsqueda que se emplean. Es por todo ello que tampoco es una buena elección utilizar un modelo orientado a grafos para implementar la base de datos del caso planteado.

Referencia bibliográfica

G. Vaish (2013). *Getting started with NoSQL*. Birmingham: Packt Publishing Ltd.

El último modelo es el orientado a agregados (Pramod y Fowler, 2012). Una base de datos NoSQL de tipo clave-valor, orientada a documentos o bien orientada a columnas, encaja perfectamente con los requisitos expuestos para el caso de estudio. No requiere la definición de un esquema de datos previo, permiten manipular datos semiestructurados, facilitan la fragmentación horizontal de los datos permitiendo la ejecución en un *cluster*, lo cual hace posible llevar a cabo la gestión de enormes cantidades de datos y soportar un elevado acceso concurrente a los mismos, y permiten estructurar los datos en forma de agregados que facilitan la explotación de los datos de formas previamente conocidas (Tiwari, 2011). Por estas razones, será una buena elección el uso de una base de datos NoSQL orientada hacia agregados. En este caso, se va a utilizar una base de datos orientada a documentos de tipo MongoDB (Membrey y otros, 2010).

Referencias bibliográficas

P. Membrey; E. Plugge; D. Hawkins (2010). *The Definitive Guide to MongoDB: the NoSQL Database for Cloud and Desktop Computing*. Nueva York: Apress.

J. S. Pramod; M.Fowler (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Reading, MA: Addison-Wesley Professional.

S. Tiwari (2011). *Professional NoSQL*. Hoboken, Nueva Jersey: John Wiley & Sons.

Se ha elegido MongoDB (Banker, 2011) por los siguientes motivos:

- En primer lugar, implementa un modelo orientado hacia agregados donde la unidad de agregación es el documento.

- Se trata de un producto estable con un soporte técnico y documental. En el sitio web de MongoDB se pueden encontrar desde manuales y tutoriales hasta vídeos de cómo usarlo.

- MongoDB ofrece formación gratuita sobre sus productos.

- Existe una cierta correspondencia entre las consultas que se pueden realizar en MongoDB y las consultas SQL de las bases de datos relacionales. Esto facilita la curva de aprendizaje.

- Existen *drivers* para la mayoría de los lenguajes de programación, lo que facilita la implementación de aplicaciones que lo usen como sistema de persistencia.

- Implementa la *framework map-reduce* (Dayley, 2015), lo que ofrece la posibilidad de realizar procesamientos eficientes de información no estructurada. En este sentido, dispone de conectores para utilizar herramientas de procesamiento masivo como Spark.

- El tipo de documento que gestiona MongoDB, basado en un formato adaptado de JSON (Hows y otros, 2014), hace que sea fácil de utilizar desde los lenguajes de programación, dado que representa a una estructura de datos existente en la mayoría de los de los lenguajes, el registro. En particular, cabe destacar la fácil integración con el lenguaje de programación Python, donde el equivalente al documento es la estructura de datos del diccionario.

- Relacionado con el formato del documento, también es importante mencionar que JSON es uno de los formatos preferidos para distribuir los datos que se utilizan en el contexto del análisis de datos y *big data*.

Referencias bibliográficas

K. Banker (2011). *MongoDB in Action*. Greenwich, CT: Manning Publications Co.

B. Dayley (2015). *Sams Teach Yourself NoSQL with MongoDB in 24 Hours*. Indianápolis, IN: Sams.

D. Hows; P. Membrey; E. Plugge (2014). *MongoDB Basics*. Nueva York: Apress.

4. Diseño lógico

Fijada la base de datos que se va a utilizar, y teniendo en cuenta la forma en la que se va a llevar a cabo la explotación de los datos, se va a realizar el diseño lógico de la base de datos.

La unidad de estructuración de MongoDB (Chodorow, 2013) son los documentos.

Referencia bibliográfica

K. Chodorow (2013). *MongoDB: the Definitive Guide*. Newton, MA: O'Reilly Media, Inc.

> Un documento físicamente es un formato adaptado del formato de datos JSON que contiene una colección de pares (clave, valor). Los valores admitidos se corresponden con un conjunto de tipos básicos tales como cadenas de caracteres, números, booleanos o *arrays*, o bien podría ser a su vez una colección de pares (clave, valor).

Los documentos se agrupan en colecciones.

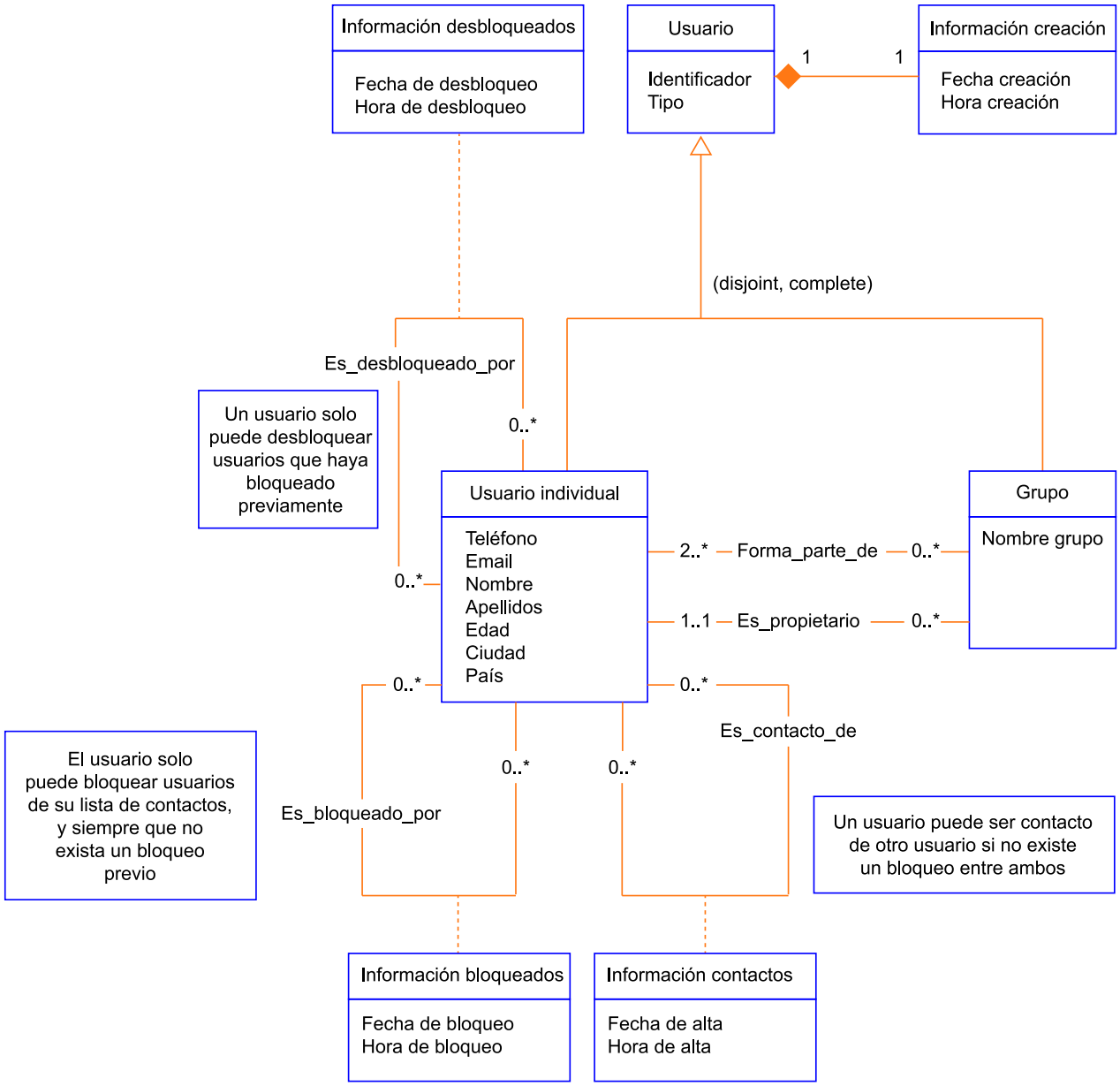
> Una colección consiste en una agrupación de documentos que tienen una relación semántica o que simplemente están relacionados. Sin embargo, una colección podría contener documentos que no estuvieran relacionados o que tuvieran estructuras diferentes, dado que no existe ningún mecanismo que realice esta comprobación. Se trata simplemente de un mecanismo de organización interna.

Por último, las colecciones se enmarcan dentro de una base de datos, pudiendo existir una o más colecciones dentro de una misma base de datos.

Así pues, el diseño lógico de una base de datos orientada a documentos consiste en decidir qué tipos de documentos son necesarios y cómo se organizan en colecciones. En este sentido, la estructura de los documentos está directamente relacionada con las formas en las que va a ser explotada la información y con el modelo del dominio. Por esta razón, antes de realizar el diseño lógico, se deben fijar sobre qué información se van a realizar las consultas y qué tipos de consultas se van a realizar.

Para ilustrar el diseño lógico, se va a elegir la siguiente parte del dominio (figura 2):

Figura 2. Parte del modelo conceptual tomado del original



Esta parte del dominio se caracteriza por:

- Existen tres relaciones reflexivas sobre la clase *Usuario individual*: *Es_contacto_de, Es_bloqueado_por* y *Es_desbloqueado_por*.

- Existe una relación de composición entre la clase *Usuario* y la clase *Información creación*.

- Existe una relación de generalización entre la clase *Usuario* y las clases *Usuario individual* y *Grupo*.

- Existen varias clases asociativas para almacenar datos en las relaciones *Es_contacto_de, Es_bloqueado_por* y *Es_desbloqueado_por*.

- Existen dos relaciones entre *Usuario individual* y *Grupo*: *Forma_parte_de* y *Es_propietario*.

- Aparecen varias restricciones de integridad expresadas en forma de restricciones textuales.

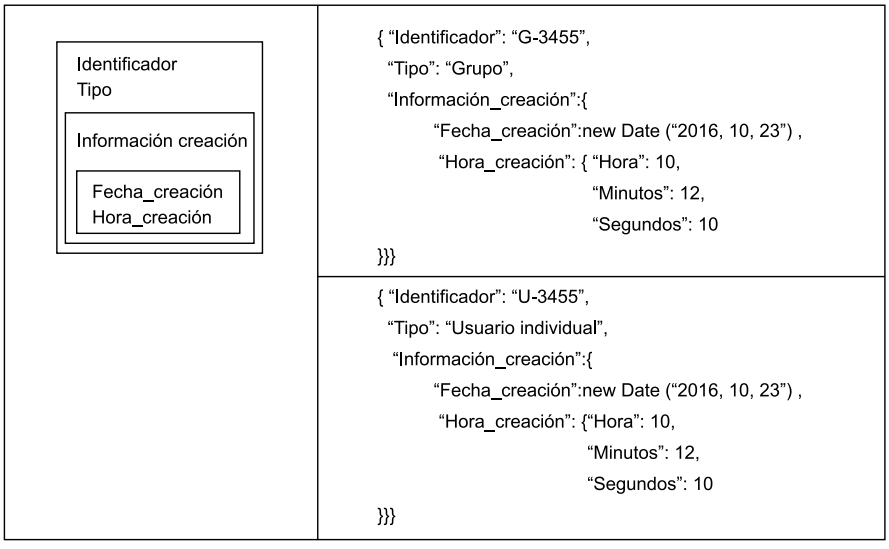
Una diferencia fundamental con respecto al diseño lógico, entre el modelo de documentos y el modelo relacional, es que en el modelo relacional existen reglas definidas acerca de cómo realizar la transformación de un diagrama conceptual en una base de datos relacional. Sin embargo, en las bases de datos orientadas a documentos, no existen reglas, y muchas de las decisiones de diseño que se toman se deben a la forma en que se va a consultar la información.

Se van a considerar las diferentes alternativas de diseño de acuerdo a la representación conceptual y al tipo de consultas de información que se pueden plantear:

1) ¿Cómo se transforman las relaciones de composición?

La relación de composición entre la clase *Usuario* con la clase que hace referencia a la información de creación de instancias concretas de esa clase podría transformarse en un documento referido a un usuario concreto que tuviera en su interior un subdocumento que representa a la información de creación de ese usuario concreto con los campos *Fecha* y *Hora de creación*. En la figura 3 se muestra el diseño para la clase *Usuario* y un ejemplo de documento.

Figura 3. Transformación de la composición definida en la clase *Usuario*

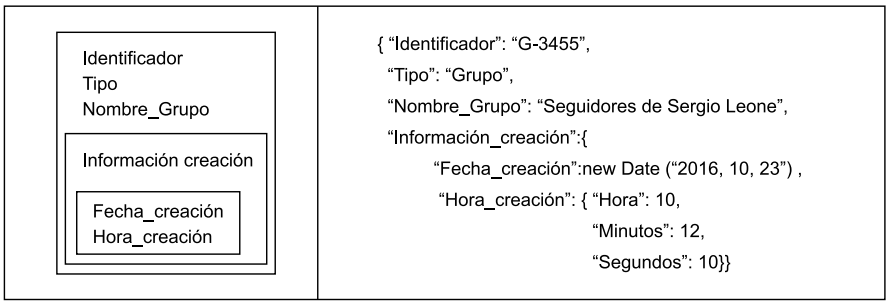


2) ¿Cómo se transforma la relación de generalización?

La relación de generalización entre la clase *Usuario* y las clases *Usuario individual* y *Grupo* se puede transformar en dos tipos de documentos. Por una parte, documentos referidos a la clase *Usuario individual* que contendrán como

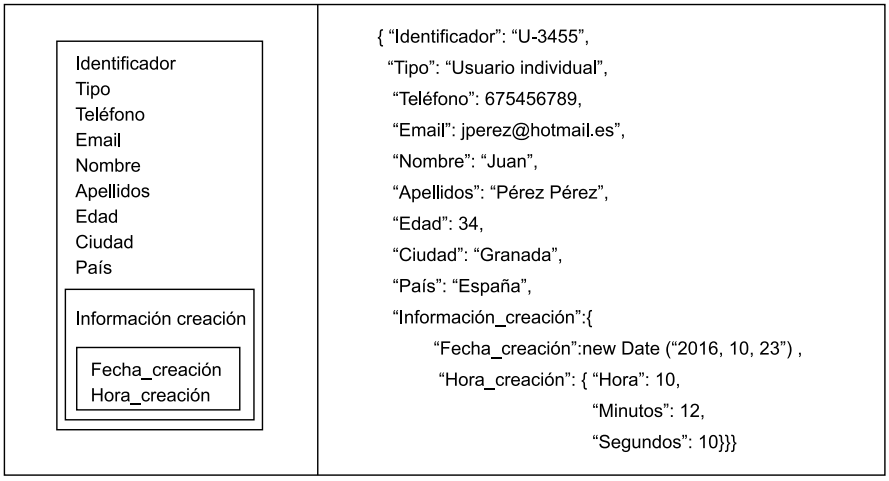
campos de información todo los que aparecen en la clase más los campos de información heredados de la superclase *Usuario*. De forma similar, existirían documentos referidos a la clase *Grupo* con campos de información propios de esta clase más los campos de información heredados de la superclase *Usuario*. En la figura 4 se muestra el diseño de los documentos para la clase *Grupo* y un ejemplo de documento.

Figura 4. Transformación de la generalización para la clase *Grupo*



Y en la figura 5 se muestra el diseño para la clase *Usuario individual* y un ejemplo de documento.

Figura 5. Transformación de la generalización para la clase *Usuario*



3) ¿Cómo se transforman las relaciones reflexivas?

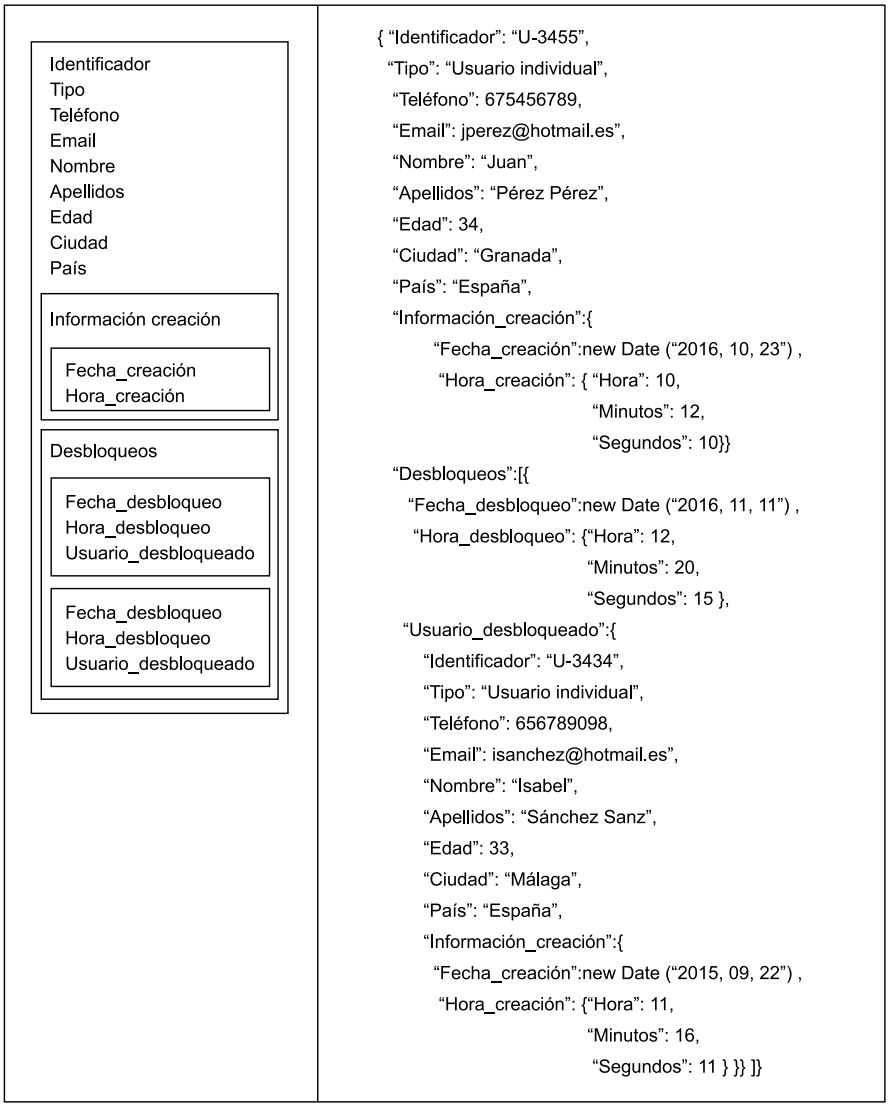
La transformación de las relaciones reflexivas resulta menos trivial, dado que en ellas sí existe una dependencia importante con respecto al tipo de explotación de la información que se va a realizar. Se va a discutir, como ejemplo ilustrativo, la relación *Es_desbloqueado_por* y los resultados serán aplicables al resto de relaciones reflexivas definidas.

Esencialmente, se podrían distinguir tres opciones para transformar la relación *Es_desbloqueado_por*:

a) Almacenar la información de la relación en el usuario que realiza el desbloqueo. Esta opción será interesante siempre y cuando el acceso a la información de los desbloqueos se realice a través de los usuarios que han realizado algún desbloqueo y se quieran hacer consultas tales como: ¿cuántos desbloqueos ha realizado un usuario concreto en un mes de un año concreto?, ¿cuántos desbloqueos realizó el usuario en un día de un mes de un año concreto?, ¿un usuario concreto ha sido desbloqueado por un usuario?, ¿cuál es la media de desbloqueos por día que realiza un usuario concreto en un mes de un año concreto?, etc. En todos los ejemplos, el acceso a la información se realiza a través de un usuario concreto que ha realizado algún desbloqueo. Es por esta razón que interesa disponer de la información acerca de los desbloqueos agrupada por los usuarios que llevan a cabo la acción de desbloquear. En esta opción de diseño, los documentos afectados por la transformación de la relación reflexiva son los documentos referidos a instancias de la clase *Usuario individual* que representan a usuarios que realizan algún desbloqueo. Estos documentos deberán incluir un campo de información específico acerca de todos los desbloqueos realizados, de manera que cada desbloqueo incluye la *Fecha* y *Hora del desbloqueo* y un subdocumento que representa al usuario que ha sido desbloqueado. En la figura 6 se muestra cómo quedaría el diseño de los documentos usando esta opción.

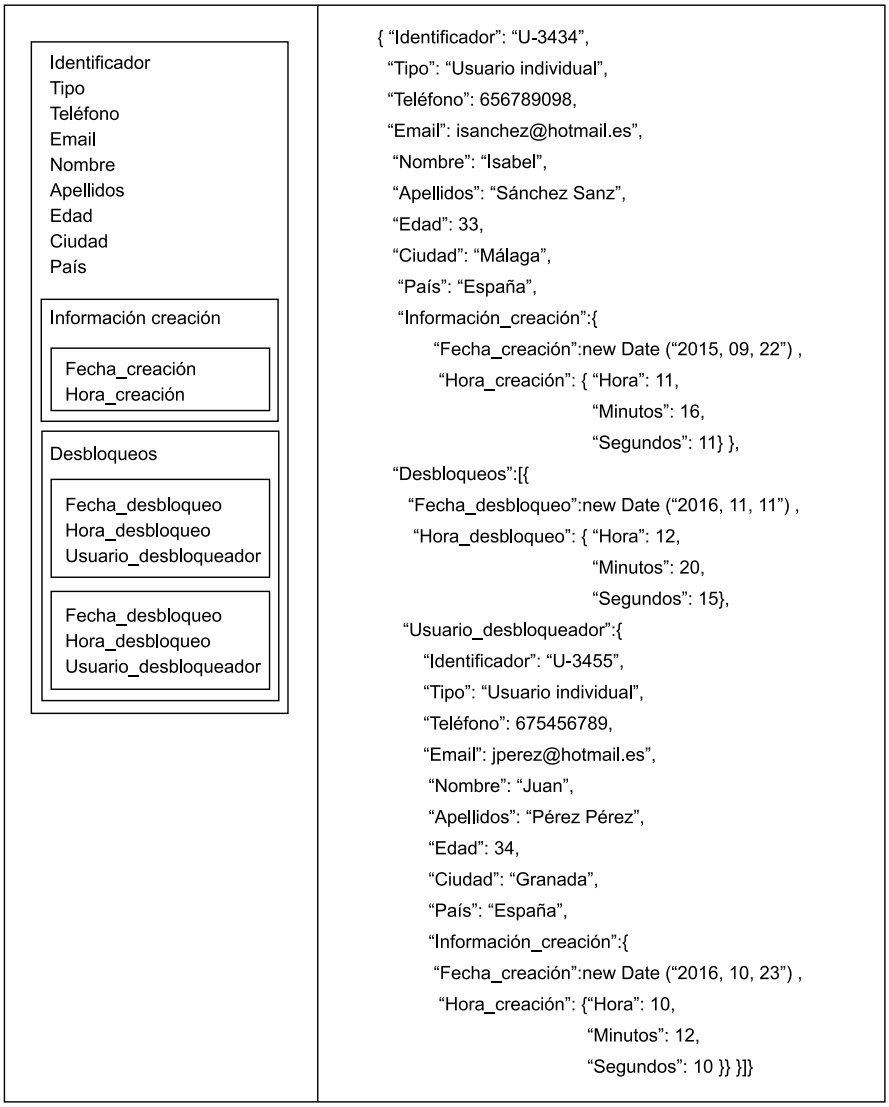
b) Almacenar la información de la relación en el usuario que es desbloqueado. Esta opción será interesante siempre y cuando el acceso a la información de los desbloqueos se realice a través de los usuarios que han sido desbloqueados, y se quieran hacer consultas tales como: ¿qué usuarios han desbloqueado al usuario en un mes de un año concreto?, ¿cuántos usuarios han desbloqueado a un usuario concreto en un mes de un año concreto?, ¿un usuario concreto ha desbloqueado al usuario?, etc.

Figura 6. Transformación de la relación reflexiva (caso a)



En todos los ejemplos, el acceso a la información se realiza a través de un usuario concreto que ha sido desbloqueado. Es por esta razón que interesa disponer de la información acerca de los desbloqueos agrupada por los usuarios que han sido desbloqueados. En esta opción de diseño, los documentos afectados por la transformación de la relación reflexiva son los documentos referidos a instancias de la clase *Usuario individual* que representan a usuarios que han sido desbloqueados. Estos documentos deberán incluir un campo de información específico acerca de todos los desbloqueos realizados, de manera que cada desbloqueo incluye la *Fecha* y *Hora del desbloqueo* y un subdocumento que representa al usuario que ha realizado el desbloqueo. En la figura 7 se muestra cómo quedaría el diseño de los documentos usando esta opción.

Figura 7. Transformación de la relación reflexiva (caso b)

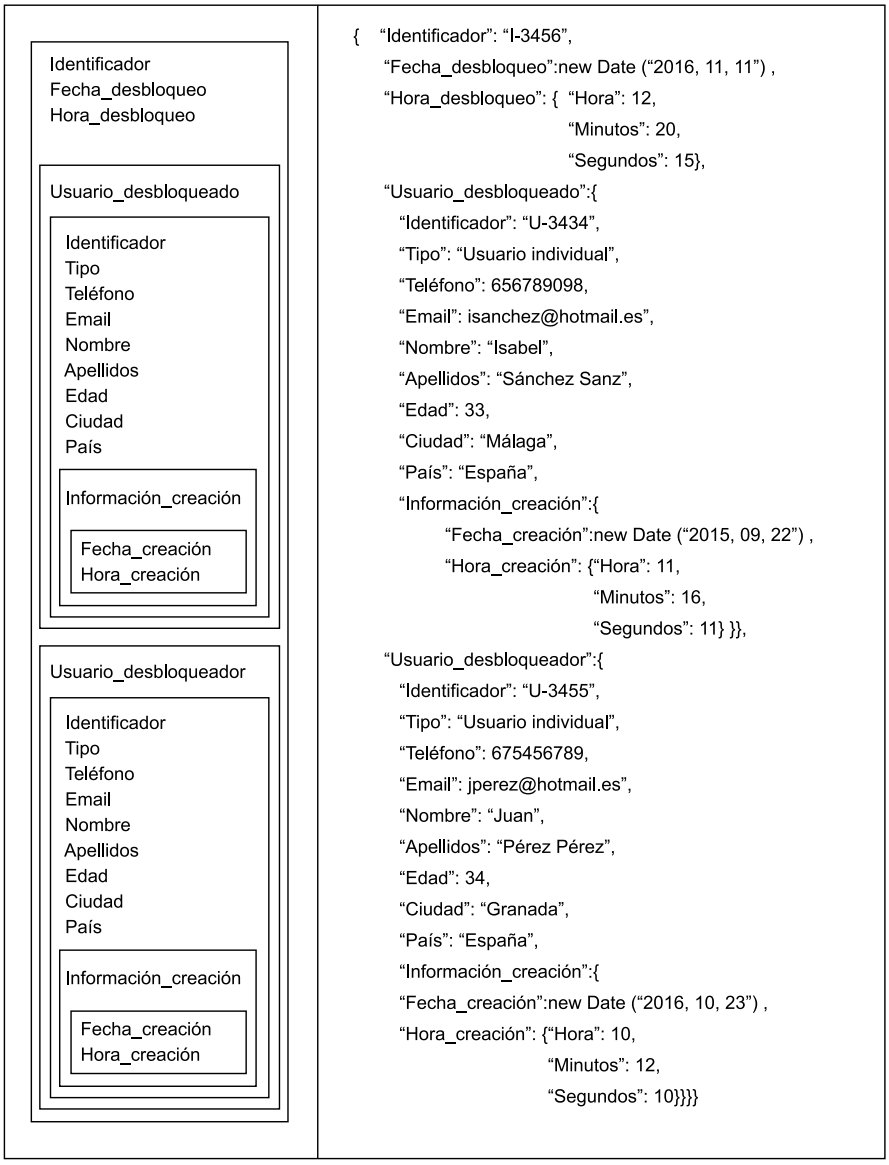


c) Almacenar la información en un documento sobre desbloqueos realizados. Esta opción será interesante siempre y cuando el acceso a la información de los desbloqueos se realice a través de los desbloqueos que se han realizado y se quieran hacer consultas tales como: ¿cuántos desbloqueos se han realizado en una fecha de un mes de un año concreto?, ¿hay algún desbloqueo de un usuario concreto en un mes de una fecha concreta?, ¿en qué fecha se han realizado más desbloqueos en un mes de un año concreto?, etc. En todos los ejemplos, el acceso a la información se realiza a través de los desbloqueos realizados. Es por esta razón que interesa disponer de la información acerca de los desbloqueos agrupada por los desbloqueos realizados. En esta opción de diseño, se crea un nuevo tipo de documento referido a cada desbloqueo realizado que incluirá un identificador del desbloqueo, la información del desbloqueo: *Fecha* y *Hora del desbloqueo* y un subdocumento que representa al usuario que ha realizado el desbloqueo y otro subdocumento que representa al usuario que ha sido desbloqueado. En la figura 8 se muestra cómo quedaría el diseño de los documentos usando esta opción.

Obsérvese que cada una de las opciones propuestas es dependiente del tipo de consulta que se quiere realizar sobre la información. Por ejemplo, si se va a consultar información acerca de los bloqueos que ha realizado un usuario concreto a lo largo de un mes, las opciones de diseño b) y c) no son una buena solución. En el caso b) habría que procesar cada uno de los documentos de los usuarios que han sido desbloqueados y comprobar si el usuario desbloqueador es el usuario buscado. Situación similar ocurre si se hubiera elegido la opción c) para realizar dicha consulta, dado que en este caso para localizar los desbloqueos realizados por un usuario concreto habría que procesar cada documento asociado a un desbloqueo y comprobar si el usuario desbloqueador es el usuario buscado. En este ejemplo, la opción de diseño a) facilita la recuperación de la información, pues basta acceder al usuario buscado y contabilizar la longitud de la lista de desbloqueos asociada.

Por otra parte, cabe señalar que si las consultas que se van a realizar sobre los documentos propuestos en los casos anteriores no van hacer uso de la información acerca de los usuarios bloqueados (o los desbloqueados), entonces una alternativa de diseño aplicable a los casos anteriores consiste en sustituir los subdocumentos que hacen referencia a los usuarios bloqueados (o los desbloqueados) por sus identificadores únicamente. De esta forma, los identificadores actuarían como claves foráneas para otros tipos de consultas.

Figura 8. Transformación de la relación reflexiva (caso c)



Tal como se comentaba anteriormente, los razonamientos expuestos son aplicables al resto de relaciones reflexivas. Así pues, se podría crear una base de datos de colecciones de documentos de acuerdo al tipo de consultas que se han analizado, tal como se muestra en la tabla 1.

Tabla 1. Colecciones de documentos

Tipo de consulta	Tipo de documento	Colección
Desbloqueos realizados por un usuario	Documento tipo a)	*Usuarios_desbloqueadores*
Desbloqueos realizados sobre un usuario	Documento tipo b)	*Usuarios_desbloqueados*
Desbloqueos realizados	Documento tipo c)	*Desbloqueos*
Bloqueos realizados por un usuario	Documento tipo a)	*Usuarios_bloqueadores*
Bloqueos realizados sobre un usuario	Documento tipo b)	*Usuarios_bloqueados*
Bloqueos realizados	Documento tipo c)	*Bloqueos*

Tipo de consulta	Tipo de documento	Colección
Contactos realizados por un usuario	Documento tipo a)	*Contactos_usuarios*
Contactos realizados	Documento tipo c)	*Contactos*

Obsérvese que en el caso de la relación reflexiva *Es_contacto_de* solo son necesarias dos colecciones. Una colección que almacena documentos sobre cada usuario y en particular un campo de información con todos los contactos de un usuario, y una colección que almacena documentos de los contactos realizados.

4) ¿Cómo se transforman las relaciones entre las clases *Grupo* y *Usuario individual*?

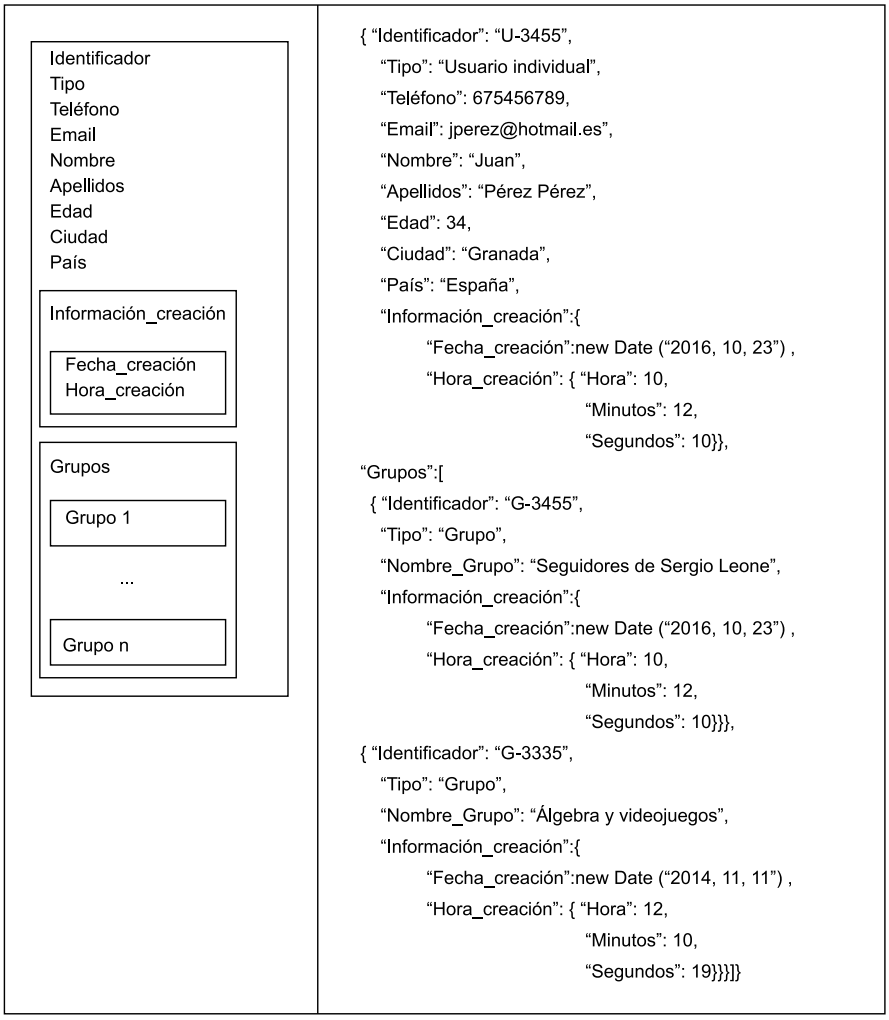
En este caso, también existe una dependencia importante con respecto al tipo de explotación de la información que se va a realizar. Se va discutir, como ejemplo ilustrativo, la relación *Forma_parte_de* y argumentaciones similares se podrán aplicar a la relación *Es_propietario*.

Se pueden considerar dos opciones para transformar la relación *Forma_parte_de*:

a) Almacenar la información de los grupos en cada usuario que es miembro de un grupo. Esta opción será interesante siempre y cuando el acceso a la información de los grupos se quiere realizar desde cada usuario concreto y se quieren hacer consultas tales como: ¿a qué grupos pertenece un usuario?, ¿a cuántos grupos pertenece un usuario?, ¿el usuario pertenece a grupo concreto?, etc. En todos los ejemplos, el acceso a la información se realiza a través de un usuario.

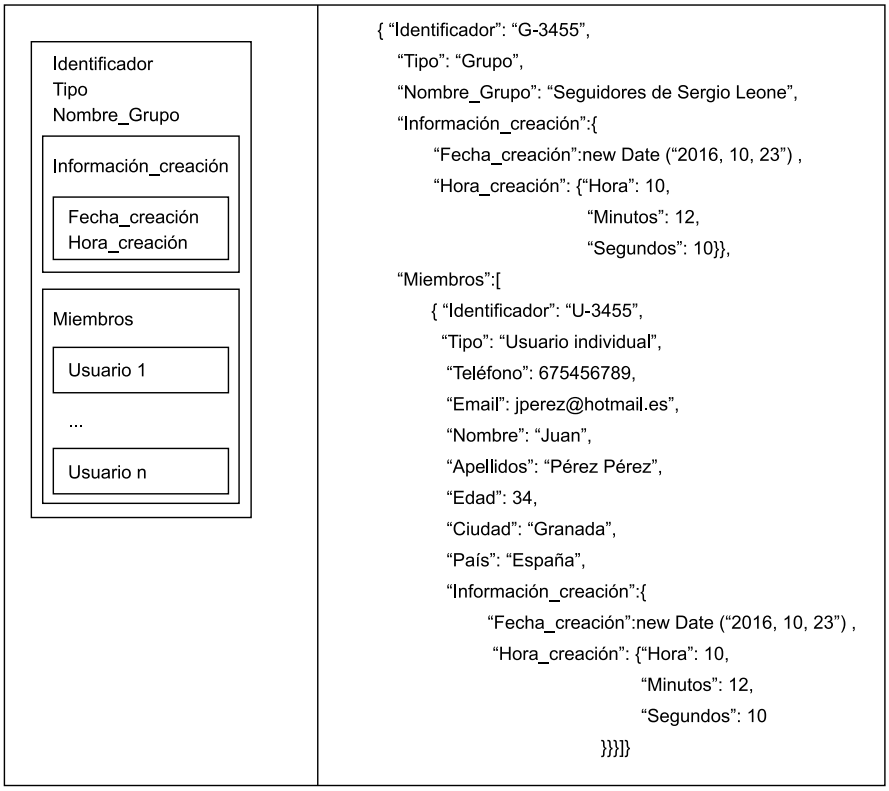
Es por esta razón que interesa disponer de la información acerca de los grupos asociada a cada usuario. En esta opción de diseño, se ven afectados los documentos que representan instancias de la clase *Usuario individual* que deberán incluir un campo de información adicional sobre los grupos de los que es miembro el usuario. En la figura 9 se muestra cómo quedaría el diseño de los documentos usando esta opción.

Figura 9. Ejemplo de documento de usuario con información de grupos



b) Almacenar la información en un documento sobre grupos creados. Esta opción será interesante siempre y cuando el acceso a la información de los grupos se realice directamente desde los grupos y se quieran llevar a cabo consultas tales como: ¿cuántos usuarios hay en un grupo concreto?, ¿pertenecen a grupo concreto unos determinados usuarios?, ¿cuáles son los usuarios que pertenecen a un grupo concreto?, etc. En todos los ejemplos, el acceso a la información se realiza a través de los grupos. Es por esta razón que interesa disponer de la información agrupada por los grupos existentes. En esta opción de diseño, se ven afectados los documentos que representan instancias de la clase *Grupo* que deberán incluir un campo de información adicional sobre los usuarios que son miembros de ese grupo. En la figura 10 se muestra cómo quedaría el diseño de los documentos usando esta opción.

Figura 10. Ejemplo de documento sobre grupos



Tal como se comentaba anteriormente, razonamientos similares se podrían aplicar a la relación *Es_propietario*. Así pues, se podría crear en la base de datos colecciones de documentos de acuerdo al tipo de consultas que se han analizado, tal como se muestra en la tabla 2.

Tabla 2. Ejemplo de documento sobre grupos

Tipo de consulta	Tipo de documento	Colección
Grupos a los que pertenece un usuario	Documento tipo a)	*Usuarios_grupos*
Grupos definidos	Documento tipo b)	*Grupos_usuarios*
Grupos de los que es propietario un usuario	Ampliar documento tipo a)	*Usuarios_grupos*
Propiedad de un grupo	Ampliar documento tipo b)	*Grupos_usuarios*

5. Diseño físico

En este apartado se va a considerar la base de datos discutida en el apartado anterior y se van a mostrar algunos ejemplos de manipulación de documentos. Tal como se ha comentado, se va a utilizar una base de datos NoSQL orientada a documentos de tipo MongoDB.

5.1. Creación de la base de datos

A diferencia de lo que ocurre en las bases de datos relacionales en las que es necesario definir previamente el esquema de definición de datos, en MongoDB el proceso de creación de una base de datos es algo trivial que se reduce a dar un nombre a la base de datos. Asimismo, se puede crear un conjunto de colecciones para agrupar los documentos que están relacionados.

Para dar un nombre (y por tanto, crear la base de datos) en MongoDB se utiliza el comando *use*. La base de datos del caso de uso se denominara *Mensajería*:

```
use Mensajería
```

Para crear una colección en MongoDB se debe indicar el nombre de la misma y utilizar el comando *db.createCollection("Nombre_Colección")*. Por ejemplo, para crear las colecciones que se obtenían a partir de la relación *Es_desbloqueado_por* se podría realizar de la siguiente manera:

a) Colección *Usuarios_desbloqueadores*:

```
db.createCollection("Usuarios_desbloqueadores")
```

b) Colección *Usuarios_desbloqueados*:

```
db.createCollection("Usuarios_desbloqueados")
```

c) Colección *Desbloqueos*:

```
db.createCollection("Desbloqueos")
```

De esta forma, se crea la metaestructura de la base de datos. Se puede consultar tanto la base de datos actual mediante el comando *db* y las colecciones que existen mediante el comando *show collections*.

5.2. Añadir documentos

Una vez creadas la base de datos y las colecciones, se pueden añadir documentos a cada una de las colecciones. Para añadir documentos, se puede hacer de varias formas. Una forma sería definir cada documento en una variable, y a continuación utilizar el método *insert*:

```
db.<Nombre_Colección>.insert( <Nombre_documento>)
```

Usando este mecanismo también se pueden añadir varios documentos a la vez; basta indicar como argumento del método la lista de los nombres de las variables que tienen definidos los documentos:

```
db.<Nombre_colección>.insert( [documento1,...,documenton])
```

Otra forma alternativa de añadir documentos a una colección consiste en pasar como argumento del método la definición del propio documento en vez del nombre de una variable que contiene la definición.

Por ejemplo, en el siguiente ejemplo se va a añadir un documento a la colección *Desbloqueos*. Para ello, se define una variable que contendrá el documento que quiere añadir.

```
documento= {    "Identificador": "I-3456",
    "Fecha_desbloqueo":new Date ("2016, 11, 11"),
    "Hora_desbloqueo": {
             "Hora": 12,
             "Minutos": 20,
             "Segundos": 15
             },
    "Usuario_desbloqueado":{
             "Identificador": "U-3434",
             "Tipo": "Usuario individual",
             "Teléfono": 656789098,
             "Email": isanchez@hotmail.es",
             "Nombre": "Isabel",
             "Apellidos": "Sánchez Sanz",
             "Edad": 33,
             "Ciudad": "Malaga",
             "País": "España",
             "Información_creación":{
                    "Fecha_creación":new Date ("2015, 09, 22") ,
                    "Hora_creación": {
                           "Hora": 11,
                           "Minutos": 16,
                           "Segundos": 11
                           }}
```

```
                    },
      "Usuario_desbloqueador":{
              "Identificador": "U-3455",
              "Tipo": "Usuario individual",
              "Teléfono": 675456789,
              "Email": jperez@hotmail.es",
              "Nombre": "Juan",
              "Apellidos": "Pérez Pérez",
              "Edad": 34,
              "Ciudad": "Granada",
              "País": "España",
              "Información_creación":{
                     "Fecha_creación": new Date ("2016, 10, 23"),
                     "Hora_creación": {
                            "Hora": 10,
                            "Minutos": 12,
                            "Segundos": 10
                            }}
               }}
```

A continuación, para insertarlo en la colección *Desbloqueos* se realiza la siguiente llamada:

```
db.Desbloqueos.insert(documento)
```

Alternativamente se podría haber invocado el método *insert* con la definición del documento.

5.3. Modificación de documentos

De manera general, para actualizar un documento se usa el comando *update*:

```
db.<Nombre_Colección>.update(<condición>,<actualización>)
```

donde *<condición>* representa la condición de búsqueda usada para recuperar la información y *<actualización>* establece qué modificaciones deben realizarse sobre el documento o documentos que se han recuperado.

5.3.1. *$set* y *$unset*

El operador *$set* establece un valor para un campo dado, y, si el campo dado no existe, lo crea.

```
{ $set: { <campo1>: <valor1>, ... } }
```

Para ilustrar su uso, se va a considerar la colección *Grupos_usuarios*.

1) Actualización sobre un campo directo

Se indica el nombre del campo. Por ejemplo, se quiere modificar el documento de la colección *Grupos_usuarios* que tiene como identificador *G-3455* y renombrarlo como *Fans de Sergio Leone*. Se podría realizar de la siguiente manera:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3455"
    },
    {
        $set:{"Nombre_Grupo" : "Fans de Sergio Leone"}
    }
);
```

2) Actualización sobre un campo de un subdocumento embebido

Usando la notación dot (.) se indica la ruta hasta el campo que se quiere modificar. Por ejemplo, se quiere realizar una modificación en el mismo documento sobre el campo *Información_creación*. En el subcampo *Fecha* se quiere cambiar el valor actual por *24/01/2014*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3455"
    },
    {
        $set:{"Información_creación.Fecha_creación" : new Date ("2014, 01, 24") }
    }
);
```

3) Actualización sobre un valor del *array*

Se usa la notación dot (.) y se indica la posición mediante el índice que ocupa el valor que quiere ser modificado. Por ejemplo, se quiere realizar una modificación en el mismo documento sobre el *array Miembros*. Se quiere cambiar el valor del teléfono del primer elemento del *array* por el teléfono *645387723*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3455"
    },
    {
        $set:{"Miembros.0.Teléfono" : 645387723}
    }
```

```
);
```

4) Creación de un nuevo campo que no existía

El operador *$set*, cuando intenta realizar una actualización, si no encuentra el campo especificado entonces lo crea. Por ejemplo, se quiere añadir al mismo documento anterior, un campo de información denominado *Num_miembros* que representa el número de miembros que tiene el grupo y asociarle el valor *10*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3455"
    },
    {
        $set:{"Num_miembros" : 10}
    }
);
```

El operador *$unset* elimina el campo especificado. Para especificar el campo, se sigue la misma notación que la usada con el operador *$set*. Sin embargo, el valor que se le asigne al campo no impacta para realizar la operación.

```
{$unset: { <campo1>: "",...} }
```

Por ejemplo, se quiere eliminar el campo *Num_miembros*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update (
    {
        "Identificador": "G-3455"
    },
    {
        $unset:{"Num_miembros" :""}
    }
);
```

En el caso de que el campo especificado no exista, no hace nada, y si se trata de un elemento de un *array*, entonces, en vez de eliminar el *array*, mantiene el campo y sustituye el valor eliminado por el valor *null*. Por ejemplo, se quiere eliminar el primer valor del *array Miembros*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update (
    {
        "Identificador": "G-3455"
    },
```

```
    {
        $unset:{"Miembros.0" :""}
    }
);
```

5.3.2. *$addToSet*

Este operador añade un valor a un *array*, salvo que el valor ya esté presente en dicho *array* en cuyo caso no hace nada. Si se omite el campo que se quiere actualizar, entonces crea un campo de tipo *array* con el valor especificado, y si el valor indicado es un *array*, lo añade como si fuera un elemento único al *array*. Y si el campo indicado no es un *array*, entonces falla la operación.

```
{$addToSet: {<campo1>: <valor1>,...} }
```

Para ilustrar su uso, se va a considerar la colección *Usuarios_grupos*.

1) Añadir un valor que no existe en un *array*

Hay que considerar el documento asociado al usuario con identificador *U-3455*, y se quiere añadir al *array Grupos*, el siguiente grupo:

```
g={ "Identificador": "G-3485",
  "Tipo": "Grupo",
  "Nombre_Grupo": "Chistes y humor",
  "Información_creación":{
            "Fecha_creación":new Date ("2014, 10, 22") ,
            "Hora_creación": {
                      "Hora": 11,
                      "Minutos": 22,
                      "Segundos": 15
                      }}
},
```

Se podría realizar con el siguiente comando:

```
db.'Usuarios_grupos'.update(
    {
        "Identificador": "U-3455"
    },
    {
        $addToSet: {"Grupos": g }
);
```

2) Añadir varios elementos a la vez

El operador *$addToSet* permite añadir varios elementos a la vez si se combina con el operador *$each*. Por ejemplo,hay que considerar nuevamente el documento del ejemplo anterior, y se quieren añadir dos grupos a la vez:

g1={ “Identificador”: “G-2385”,
“Tipo”: “Grupo”,
“Nombre_Grupo”: “Inteligencia artificial”,
“Información_creación”:{
“Fecha_creación”:new Date (“2013, 11, 20”) ,
“Hora_creación”: {
“Hora”: 12,
“Minutos”: 24,
“Segundos”: 11
}}
}
	g2={ “Identificador”: “G-32345”,
	“Tipo”: “Grupo”,
	“Nombre_Grupo”: “Política”,
	“Información_creación”:{
	“Fecha_creación”:new Date (“2012, 11, 22”) ,
	“Hora_creación”: {
	“Hora”: 15,
	“Minutos”: 12,
	“Segundos”: 25
	}}
	}

Se podría realizar con el siguiente comando:

```
db.'Usuarios_grupos'.update(
    {
        "Identificador": "U-3455"
    },
    {
        $addToSet:   { "Grupos": {$each: [g1, g2]}}});
```

5.3.3. *$pop*

Este operador elimina el primer o último campo de un *array*. Si se elimina el último campo que tuviera un *array*, entonces se le asigna el *array* vacío. Para especificar el campo que se desea eliminar, se indica el nombre del *array* y como valor toma -1 para indicar el primer elemento y 1 para indicar el último elemento.

```
{$pop: {<campo>: <-1 | 1>,...} }
```

Para ilustrar su uso, se va a considerar la colección *Usuarios_grupos*. Por ejemplo, se quiere eliminar el último grupo del *array Grupos* del documento asociado al usuario con identificador *U-3455*. Se podría realizar con el siguiente comando:

```
db.'Usuarios_grupos'.update (
    {
        "Identificador": "U-3455"
    },
    {
      $pop:   {"Grupos":1}
});
```

5.3.4. *$pull*

Este operador elimina de un *array* todas las instancias de un valor o valores que encajen con la condición especificada.

```
{ $pull: { <campo1>: <valor|condición>, <campo2>: <valor|condición>, ... } }
```

Si se especifica una condición y los elementos del *array* son documentos embebidos, entonces se aplica la condición como si cada elemento fuera un documento. Por otro lado, si el valor especificado es un *array*, se eliminan solo los elementos del *array* que encajen exactamente con el valor especificado incluyendo el orden de los elementos. Y si el valor fuera un documento, entonces se eliminan todos los valores del *array* que sean documentos con los mismos campos y valores sin importar el orden en que aparecen.

Para ilustrar su uso se va a considerar la colección *Grupos_usuarios*.

1) Eliminar todos los elementos que son iguales a un valor dado

Considerar el documento asociado al grupo con identificador *G-3485*. Se quiere eliminar todos los elementos del *arrayMiembros* que tomen el siguiente valor:

```
m={ "Identificador": "U-3455",
   "Tipo": "Usuario individual",
   "Teléfono": 675456789,
   "Email": jperez@hotmail.es",
   "Nombre": "Juan",
   "Apellidos": "Pérez Pérez",
   "Edad": 34,
   "Ciudad": "Granada",
   "País": "España",
   "Información_creación":{
            "Fecha_creación":new Date ("2016, 10, 23") ,
            "Hora_creación": {
                      "Hora": 10,
                      "Minutos": 12,
                      "Segundos": 10
                      }
}
}
```

Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3485"
```

```
    },
    {
        $pull:{"Miembros":m}
    }
);
```

2) Eliminar elementos de un *array* de documentos

Considerar el documento asociado al grupo con identificador *G-3485*. Se quiere eliminar todos los elementos del *array Miembros* donde el valor del subcampo *Teléfono* toma el valor *675456789*. Podría realizarse con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {"Identificador" : "G-3485"
    },
    {
    $pull:{"Miembros":{"Teléfono": 675456789}}
    },);
```

Una variación sería eliminar del documento asociado al grupo con identificador *G-3485* todos los elementos del *array Miembros* donde, al menos, el valor del subcampo *Teléfono* toma el valor *675456789* y el valor del subcampo *Edad* toma el valor *34*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3485"
    },
    {
        $pull:{"Miembros":{$elemMatch :{ "Teléfono":675456789, "Edad":34}}}
    }
);
```

3) Eliminar todos los elementos de acuerdo a una condición dada

Considerar el documento asociado al grupo con identificador *G-3485*. Se quiere eliminar todos los elementos del *array Miembros* donde el valor del subcampo *Edad* es mayor que *34*. Se podría realizar con el siguiente comando:

```
db.'Grupos_usuarios'.update(
    {
        "Identificador" : "G-3485"
    },
    {
        $pull:{"Miembros":{"Teléfono":{$gt:34}}}
    }
```

```
);
```

5.3.5. *$push*

Este operador añade un valor especificado a un *array* dado.

```
{$push: {<campo1>: <valor1>,...} }
```

Si el campo se omite, entonces se crea un campo de tipo *array* para incluir el elemento especificado en el valor. Si el valor es un *array*, entonces añade el mismo como si fuera un elemento único, y si se quiere añadir cada elemento del valor de forma separada, entonces se usa el modificador *$each*.

Este modificador tiene asociado un conjunto de modificadores que pueden usarse para complementar el comportamiento del modificador:

- *$each*: Permite añadir más de un valor al *array*.
- *$slice*: Limita el número de elementos del *array*.
- *$sort*: Ordena los elementos del *array*.
- *$position*: Especifica la localización en el *array* en la cual insertar los nuevos elementos. Por defecto, *$push* añade al final del *array*.

Para ilustrar su uso, se va a considerar la colección *Usuarios_desbloqueadores*:

1) Añadir un valor a un *array*

Considerar el documento asociado al usuario con identificador *U-3455*; se quiere añadir al *array Desbloqueados*, el siguiente usuario que ha sido desbloqueado:

```
d={
   "Fecha_desbloqueo":new Date ("2016, 11, 11"),
   "Hora_desbloqueo": {
         "Hora": 12,
         "Minutos": 20,
         "Segundos": 15
         },
   "Usuario_desbloqueado":{
         "Identificador": "U-3434",
         "Tipo": "Usuario individual",
         "Teléfono": 656789098,
         "Email": isanchez@hotmail.es",
         "Nombre": "Isabel",
         "Apellidos": "Sánchez Sanz",
```

```
          "Edad": 33,
          "Ciudad": "Malaga",
          "País": "España",
          "Información_creación":{
                "Fecha_creación":new Date ("2015, 09, 22");
                "Hora_creación": {
                      "Hora": 11,
                      "Minutos": 16,
                      "Segundos": 11
                      }
                }
    }
```

Se podría realizar con el siguiente comando:

```
db.'Usuarios_desbloqueadores'.update(
    {
        "Identificador" : "U-3455"
    },
    {
        $push:{"Desbloqueados": d }
    }
);
```

2) Añadir múltiples valores a un *array*

Considerar el documento asociado al usuario con identificador *U-3455*; se quiere añadir al *array Desbloqueados* los siguientes usuarios que han sido desbloqueados:

```
d1={
    “Fecha_desbloqueo”:new Date (“2016, 11, 11”) ,
     “Hora_desbloqueo”: {
                        “Hora”: 12,
                        “Minutos”: 20,
                        “Segundos”: 15
                        },
   “Usuario_desbloqueado”:{
      “Identificador”: “U-3434”,
      “Tipo”: “Usuario individual”,
      “Teléfono”: 656789098,
      “Email”: isanchez@hotmail.es”,
      “Nombre”: “Isabel”,
      “Apellidos”: “Sánchez Sanz”,
      “Edad”: 33,
      “Ciudad”: “Málaga”,
      “País”: “España”,
      “Información_creación”:{
        “Fecha_creación”:new Date (“2015, 09, 22”) ,
         “Hora_creación”: {
                        “Hora”: 11,
                        “Minutos”: 16,
                        “Segundos”: 11
                        }
       }
    }
  }
```

```
d2={
    “Fecha_desbloqueo”:new Date (“2015, 11, 11”) ,
     “Hora_desbloqueo”: {
                        “Hora”: 13,
                        “Minutos”: 21,
                        “Segundos”: 15
                        },
   “Usuario_desbloqueado”:{
      “Identificador”: “U-3431”,
      “Tipo”: “Usuario individual”,
      “Teléfono”: 656783398,
      “Email”: isanz@hotmail.es”,
      “Nombre”: “Israel”,
      “Apellidos”: “Sanz Sanz”,
      “Edad”: 32,
      “Ciudad”: “Madrid”,
      “País”: “España”,
      “Información_creación”:{
        “Fecha_creación”:new Date (“2012, 09, 21”) ,
         “Hora_creación”: {
                        “Hora”: 12,
                        “Minutos”: 26,
                        “Segundos”: 21
                        }
       }
    }
  }
```

Se podría realizar con el siguiente comando:

```
db.'Usuarios_desbloqueadores'.update(
    {
        "Identificador" : "U-3455"
    },
    {
        $push:{"Desbloqueados": { $each :[d1,d2]}}
    }
);
```

5.4. Ejemplos de eliminación de documentos

Para eliminar un documento, se utiliza el comando:

```
db.<Nombre_Colección>.remove(<condición>)
```

Considerar la colección *Usuarios_grupos*. Se quiere eliminar aquel documento tal que el campo de información *Teléfono* tome el valor *675456789*, y el campo de información *Ciudad* tome el valor *Granada*. Se podría hacer con el comando:

```
db.'Usuarios_grupos'.remove({
        "Teléfono" : 675456789,
        "Ciudad": "Granada",
```

```
        });
```

5.5. Ejemplos de consulta de documentos

Para buscar información dentro de un documento se utiliza el comando *find*:

```
db.<Nombre_Colección>.find(<condición>,<salida>)
```

donde <*condición*> representa la condición de búsqueda usada para recuperar la información y <*salida*> establece qué forma debe tener el resultado devuelto.

La condición de búsqueda expresa qué documentos se quieren recuperar. Para ello, se utilizan pares (clave, valor) que identifican dichos documentos. En caso de no indicar ningún par, se recuperan todos los documentos.

En general, en la <*salida*> se especifica qué campos de información de los resultados devueltos se quieren mostrar. Para ello, se listan los campos y toman el valor 1 si se quiere mostrar o el valor 0 si no se quiere mostrar el campo. En caso de no listar ningún campo, entonces serán devueltos todos los campos.

Se va a considerar la colección *Usuarios_grupos* para ilustrar las consultas.

1) Búsqueda directa

Mostrar el documento de la colección *Usuarios_grupo* tal que el campo de información *Teléfono* tome el valor *675456789* y el campo de información *Ciudad* tome el valor *Granada*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{
        "Teléfono" : 675456789,
        "Ciudad": "Granada",
},
{"Nombre":1, "Apellidos":1}
)
```

2) Búsqueda en subagregados

a) Buscar el documento que en el subcampo *Fecha_creación* del campo *Información_creación* tome el valor de *23/10/2016*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{"Información_creación.Fecha_creación":new Date("2016, 10, 23")},
{"Nombre":1, "Apellidos":1}
)
```

b) Buscar los documentos que en el subcampo *Hora* del subcampo *Hora_creación* del campo *Información_creación* tome un valor mayor que *10*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{"Información_creación.Hora_creación_Hora":{$gt:10}},
{"Nombre":1, "Apellidos":1}
)
```

c) Buscar los documentos que en los subcampos *Hora*, *Minutos* y *Segundos* del subcampo *Hora_creación* del campo *Información_creación* tomen los siguientes valores respectivamente: *10*, *12* y *10*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{"Información_creación.Hora_creación_Hora":{
                                    "Hora": 10,
                                    "Minutos": 12,
                                    "Segundos": 10
},
{"Nombre":1, "Apellidos":1}
)
```

3) Búsqueda en *arrays*

a) Buscar los documentos donde haya algún elemento del *array Grupos* que tome el valor *Seguidores de Sergio Leone* en el campo de información *Nombre_grupo* y el valor *Grupo* en el campo de información *Tipo*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{"Grupos":{$elemMatch:
{"Nombre_grupo":"Seguidores de Sergio Leone","Tipo":"Grupo"}},
{"Nombre":1, "Apellidos":1}
)
```

b) Buscar los documentos donde haya algún elemento del *array Grupos* que en el subcampo *Hora* del subcampo *Hora_creación* del campo *Información_creación* tome un valor mayor que *10*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{"Grupos.Información_creación.Hora_creación_Hora":{$gt:10}},
{"Nombre":1, "Apellidos":1}
)
```

c) Buscar los documentos donde el *array Grupos* contenga alguno de los siguientes grupos. Mostrar el nombre y apellidos del usuario.

g1={ “Identificador”: “G-3455”,
“Tipo”: “Grupo”,
“Nombre_Grupo”: “Seguidores de Sergio Leone”,
“Información_creación”:{
“Fecha_creación”:new Date (“2016, 10, 23”) ,
“Hora_creación”: {
“Hora”: 10,
“Minutos”: 12,
“Segundos”: 10
}}
}
	g2={ “Identificador”: “G-3335”,
	“Tipo”: “Grupo”,
	“Nombre_Grupo”: “Álgebra y videojuegos”,
	“Información_creación”:{
	“Fecha_creación”:new Date (“2014, 11, 11”) ,
	“Hora_creación”: {
	“Hora”: 12,
	“Minutos”: 10,
	“Segundos”: 19
	}}
	}

```
db.'Usuarios_grupos'.find(
{"Grupos":{$in:[ g1,g2]}},
{"Nombre":1, "Apellidos":1}
)
```

d) Buscar los documentos donde el *array Grupos* contenga exactamente los grupos *g1* y *g2*. Mostrar el nombre y apellidos del usuario.

```
db.'Usuarios_grupos'.find(
{"Grupos":g1,"Grupos":g2},
{"Nombre":1, "Apellidos":1}
)
```

4) Ordenación y contar resultados

a) Contar el número de documentos donde el campo de información *Ciudad* toma el valor de *Granada*.

```
db.'Usuarios_grupos'.find(
{"Ciudad":"Granada"},{}).count()
```

b) Ordenar de forma ascendente los documentos de la colección *Usuarios_grupos* por el campo *Teléfono*.

```
db.'Usuarios_grupos'.find({}).sort({"Teléfono":-1})
```

5.6. Creación de índices

Para crear un índice se utiliza el comando:

```
db.<Nombre_Colección>.createIndex(<claves>)
```

donde cada clave se especifica como *<Nombre_campo:<1,-1>>*. Si toma el valor 1 indica que la ordenación de los valores en el índice se realiza de forma ascendente y -1 indica que se realiza de forma descendente.

Por ejemplo, si se quisiera crear un índice sobre los documentos de la colección *Usuario_grupos* sobre los campos *Teléfono* de forma ascendente y *Edad* de forma descendente, se podría hacer de la siguiente forma:

```
dbUsuarios_grupos.createIndex({"Teléfono":1,"Edad":-1})
```

Para crear un índice sobre el campo de un subagregado se utiliza la notación dot (.). Por ejemplo, si se quiere crear un índice ascendente sobre el campo *Fecha_creación* del subagregado *Información_creación* se podría hacer de la siguiente manera:

```
dbUsuarios.createIndex({"Información_creación.Fecha_creación":1})
```

Y si se quiere usar todos los campos del subagregado, basta indicar el nombre del mismo. Por ejemplo, si se quiere crear un índice ascendente sobre el subagregado *Información_creación*, se podría hacer de la siguiente manera:

```
dbUsuarios.createIndex({"Información_creación":1})
```

5.7. Agregación

La *framework* de agregación es un conjunto de operaciones que permiten procesar los datos de forma que los resultados se obtienen mediante la combinación de sus valores. Las operaciones agregadas se aplican sobre colecciones de documentos y devuelve los valores procesados en forma de uno o más documentos. Se realiza mediante el método *aggregate*:

```
db.<Nombre_colección>.aggregate( [ { <Fase_1> }, ..., { <Fase_n> } ] )
```

donde *<Fase_1>...<Fase_n>* son conjuntos de operaciones que se aplican de forma ordenada por fases. Algunas de las operaciones disponibles son:

a) *$project*: Modifica los datos de entrada, añadiendo, eliminando o recalculando campos para que la salida sea diferente. Por ejemplo, excluir o incluir campos:

```
{"$project": {<campo>: 1, <campo>: 1}}
```

Crear nuevos campos usando los valores de otros campos mediante la notación $ y el nombre:

```
{"$project": {<campo_nuevo>: "$otro_campo", <campo_nuevo>: <expresion>}}
```

b) *{"$match": <expresion>}*: filtra la entrada dejando solo los que cumplan las condiciones establecidas.

c) *{"$limit": <numero>}*: restringe el número de resultados al número indicado.

d) *{"$skip": <positivo_entero>}*: ignora un número determinado de registros, y devuelve los siguientes.

e) *{ "$group": { "_id": { <campo1>: <expresion>, ... },<campo>: { <operador>: <expresion> }, ... } }*: agrupa documentos según una determinada condición. Se pueden elegir varios campos como identificador.

f) *{"$sort": <expresion>}*: ordena un conjunto de documentos según el campo especificado.

g) *{"$out": <coleccion>}*: almacena los resultados en la colección indicada. Crea la colección si no existe, la reemplaza atómicamente si existe y si hay errores no hace nada.

Por ejemplo, se quiere obtener los identificadores de los usuarios tal que la fecha de creación del usuario fue el *23/10/2016* a partir de los documentos de la colección *Usuarios_grupos*:

```
db.Usuarios_grupos.aggregate([{$match:{"Información_creación.Fecha_creación":new Date("2016, 10,
23")}}, {$project:{"Identificador":1}}])
```

Las operaciones que se llevan a cabo son las siguientes:

- En primer lugar, se obtienen aquellos documentos de la colección *Usuarios_grupos* en los que el campo *Fecha_creación* del subagregado *Información_creación* toma el valor *23/10/2016* mediante el operador *$match*.

- A continuación, la salida de la operación anterior (documentos que cumplen la condición indicada) filtra los campos y solo se va a mostrar el campo *Identificador*.

También existe un conjunto de métodos que se utilizan para llevar a cabo una única operación:

- *db.collection.count (<query>)*: devuelve el número de documentos que cumplen la condición especificada en la *query* (consulta).

- *db.collection.distinct (field, query)*: devuelve el número de documentos diferentes según el campo *field* de los documentos que cumplen la condición especificada en *query*.

- *db.collection.group ({key, reduce, initial [, keyf] [, cond] [,finalize]})*: esta operación toma el conjunto de documentos que cumplen la *query* y los agrupa en función de un determinado campo o campos. Devuelve un *array* de documentos con los resultados procesados para cada grupo.

Por ejemplo, si se quiere contar cuántos usuarios de la colección *Usuarios_grupos* viven en *Madrid* sería necesario ejecutar la siguiente sentencia:

```
db.Usuarios_grupos.count({"Ciudad":"Madrid"})
```

5.8. Transacciones

En MongoDB las operaciones de escritura son atómicas en lo que respecta al documento, incluso cuando se modifican múltiples documentos embebidos dentro de un documento. Sin embargo, cuando la escritura afecta a varios documentos independientes, entonces la modificación de cada documento es atómica, aunque entre documento y documento pueden intercalarse otras operaciones. Para aislar una operación que afecta a múltiples documentos, se puede usar el operador *$isolated*. Este operador previene que otros procesos se intercalen entre la operación de escritura una vez que se comienza la modificación del primer documento. Sin embargo, esta operación no ofrece la atomicidad «todo o nada» propia de las transacciones ACID disponibles en las bases de datos relacionales.

Por ejemplo, se quiere modificar de forma aislada en todos los documentos de la colección *Usuarios_grupos* el campo *Fecha_creación* del subagregado *Información_creación* con valor *23/10/2016* y asignarle el valor *null*. Se puede hacer de la siguiente manera:

```
db.Mensajes.update({"Información_creación.Fecha_creación":
new Date("2016, 10, 23") ,$isolated:1},
{$set:{" Información_creación.Fecha_creación ":null}},
{multi:1})
```

Resumen

En este material didáctico se ha realizado el diseño de una base de datos que permite almacenar información de un sistema de mensajería instantánea.

El documento comienza con una contextualización y descripción informal del caso de uso planteado. A continuación, se presenta un diseño conceptual del caso, que representa formalmente el caso propuesto. A partir del mismo y de la contextualización, se realiza una discusión acerca de las diferentes alternativas tecnológicas que podrían utilizarse en este caso, haciendo hincapié en las ventajas y desventajas de cada una de ellas. Se selecciona como mejor alternativa la opción de una base de datos orientada a documentos. A continuación, en el apartado sobre el diseño lógico, se discute paso a paso cómo realizar la transformación de una parte del diseño conceptual del caso propuesto en una base de datos orientada a documentos de tipo MongoDB. Y en el apartado sobre diseño físico se plantean ejemplos que ilustran cómo explotar los datos de la base de datos desarrollada. Por último, en el apartado de actividades se plantean un conjunto de consultas adicionales.

El material incluye tres anexos. En el primer anexo se explica cómo instalar la base de datos documental MongoDB; en el segundo anexo se explica cómo instalar una herramienta de ayuda; por último, en el anexo tercero, se proporciona una guía básica de sentencias de consulta en MongoDB.

Actividades

A continuación, se muestra un conjunto de ejercicios propuestos que servirán para fijar los conceptos introducidos:

1. Recuperar todos los documentos de la colección *Usuarios_grupos* que pertenecen a usuarios con más de *30* años y que viven en *Barcelona*. Mostrar la información personal del usuario. Ordenar los resultados por el número de teléfono del usuario.

2. Recuperar todos los documentos de la colección *Usuarios_grupos* que representen a usuarios que fueron creados en una fecha posterior al *23/06/2012*. Mostrar la información personal del usuario. Ordenar los resultados por el número de teléfono del usuario.

3. Recuperar todos los documentos sobre *Grupos_usuarios* tal que tengan algún miembro que viva en *Valencia* o *Barcelona*. Mostrar el identificador y nombre del grupo.

4. Recuperar todos los documentos de la colección *Desbloqueos* tales que un usuario haya desbloqueado a un usuario con identificador *U-3434* después de *01/01/2015*. Mostrar nombre y apellidos del usuario desbloqueador, ordenados por el nombre de manera ascendente y apellidos de manera descendente.

5. Añadir un nuevo campo denominado *Licencia* a todos los documentos de la colección *Grupos_usuarios* que hayan sido creados con fecha posterior al *01/01/2015* e insertar el siguiente documento:

```
{"nombre":"Creative Commons",
 "Aplicación": "Universal",
  "Ámbito": "Nacional"}}
```

6. Buscar los documentos de la colección *Grupos_usuarios* que tengan miembros de *Extremadura*. Se pide eliminar en esos documentos los usuarios que sean de *Toledo*.

7. Se pide actualizar los documentos de la colección *Usuarios_grupos*. Usando una única sentencia, se quiere eliminar el campo *Fecha_creación* del subagregado *Información_creación* y añadir un nuevo campo denominado *Fecha_creación* que tome el valor *null*.

8. Se han actualizado todos los documentos de la colección *Grupos_usuarios* añadiendo un nuevo campo denominado *Comunidades* que contiene una lista de todas las comunidades autónomas a las que pertenecen sus miembros. Se sabe que los miembros de algunos grupos que contienen miembros de Aragón no han sido actualizados Se pide actualizar el campo *Comunidades* de los documentos en las condiciones especificadas pero sin repetir el valor de Aragón.

Bibliografía

Banker, K. (2011). *MongoDB in Action*. Greenwich, CT: Manning Publications Co.

Chodorow, K. (2013). *MongoDB: the Definitive Guide*. Newton, MA: O'Reilly Media, Inc.

Dayley, B. (2015). *Sams Teach Yourself NoSQL with MongoDB in 24 Hours*. Indianápolis, IN: Sams.

Hows, D.; Membrey, P.; Plugge, E. (2014). *MongoDB Basics*. Nueva York: Apress.

Membrey, P.; Plugge, E.; Hawkins, D. (2010). *The Definitive Guide to MongoDB: the NoSQL Database for Cloud and Desktop Computing*. Nueva York: Apress.

Mohamed, M. A.; Altrafi, O. G.; Ismail, M. O. (2014). «Relational vs. NoSQL Databases: A Survey». *International Journal of Computer and Information Technology* (vol. 3, núm. 3).

Moniruzzaman, A. B. M.; Hossain, S. A. (2013). «NoSQL database: New Era of Databases for Big Data Analytics-Classification, Characteristics and Comparison». *ArXiv preprint arXiv*: 1307.0191.

Pramod, J. S.; Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Reading, MA: Addison-Wesley Professional.

Redmond, E.; Wilson, J. R. (2012). *Seven Databases in Seven Weeks: a Guide to Modern Databases and the NoSQL Movement*. Raleigh, NC: Pragmatic Bookshelf.

Tiwari, S. (2011). *Professional NoSQL*. Hoboken, Nueva Jersey: John Wiley & Sons.

Vaish, G. (2013). *Getting Started with NoSQL*. Birmingham: Packt Publishing Ltd.

Anexo

I. Guía de instalación de MongoDB

En este anexo se va a explicar la instalación de MongoDB sobre Windows y Linux Ubuntu.

1) Instalación sobre Windows

Para realizar la instalación, se necesita el programa de instalación con extensión *.msi* que se puede encontrar en el sitio oficial de descargas de MongoDB.

Los pasos para instalarlo son:

a) Hacer doble clic sobre el instalador (figura 11) y seguir los pasos que se indica pulsando sobre *Next* en cada pantalla (figura 12).

Figura 11. Instalador de MongoDB

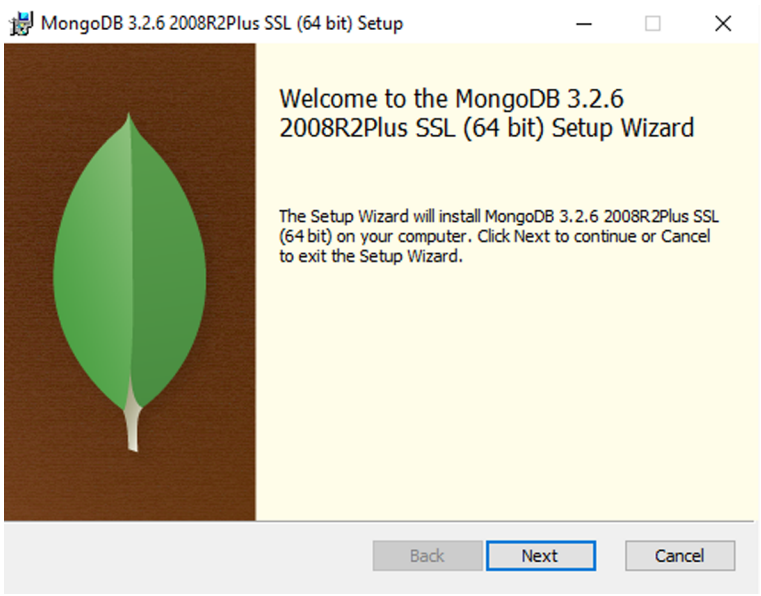
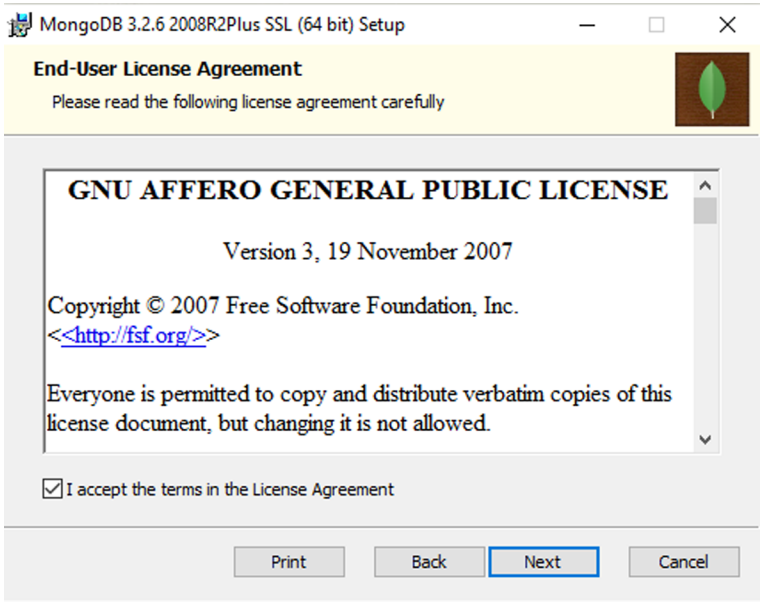
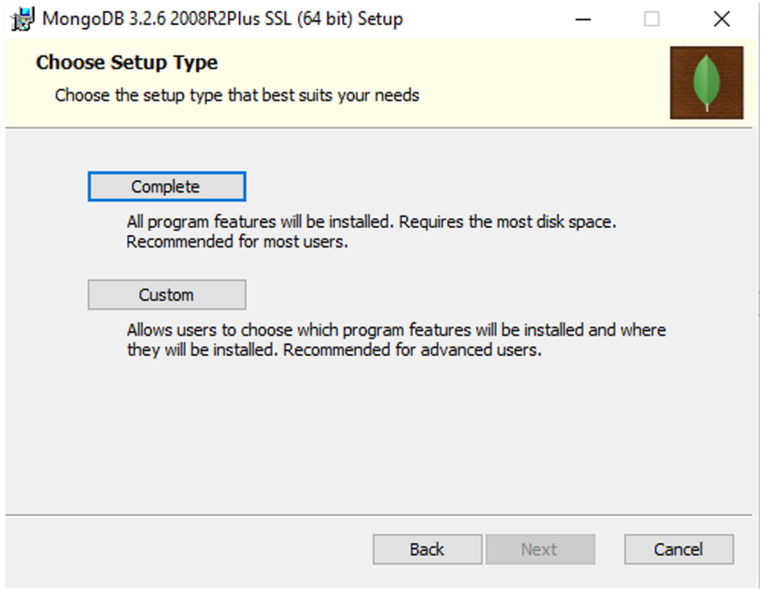


Figura 12. Comienzo del instalador



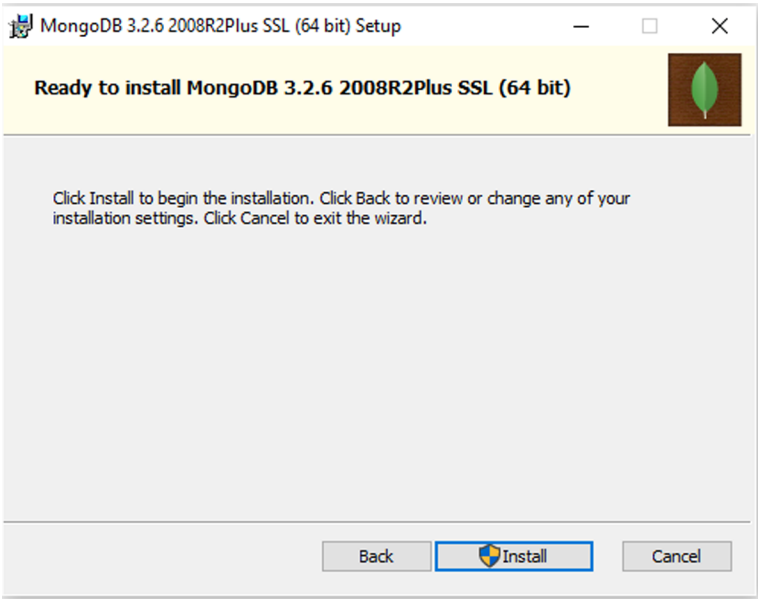
En la siguiente pantalla se elige la opción *Complete* (figura 13).

Figura 13. Configuración de instalación



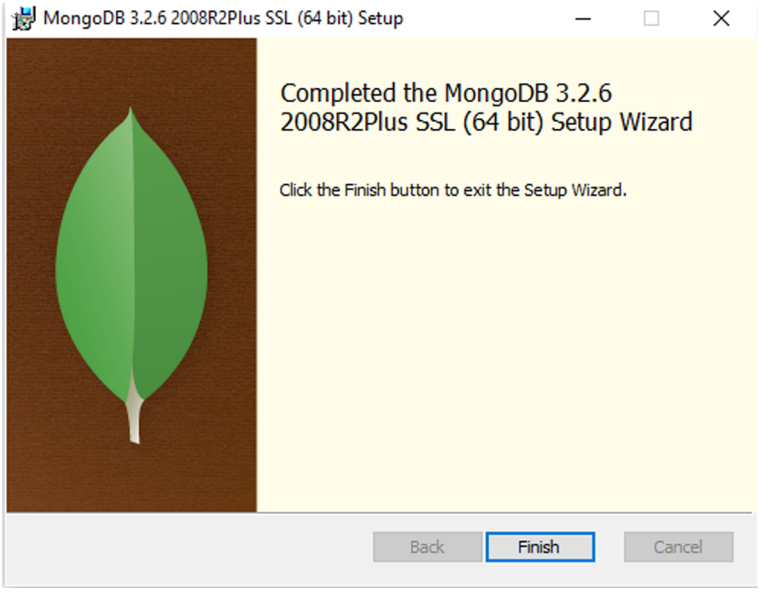
En la siguiente pantalla se pulsa sobre *Install* (figura 14).

Figura 14. Instalación



Cuando finaliza la instalación, aparece la siguiente pantalla (figura 15).

Figura 15. Finalización de instalación



b) En el subdirectorio *bin* del directorio seleccionado, para llevar a cabo la instalación, se encuentran todas las aplicaciones que se han instalado. Entre las aplicaciones instaladas, hay dos aplicaciones básicas para trabajar con MongoDB:

- *mongo*; permite usar el *Shell* o línea de comandos para interactuar con MongoDB. Desde este *Shell* se puede hacer cualquier operación sobre MongoDB.

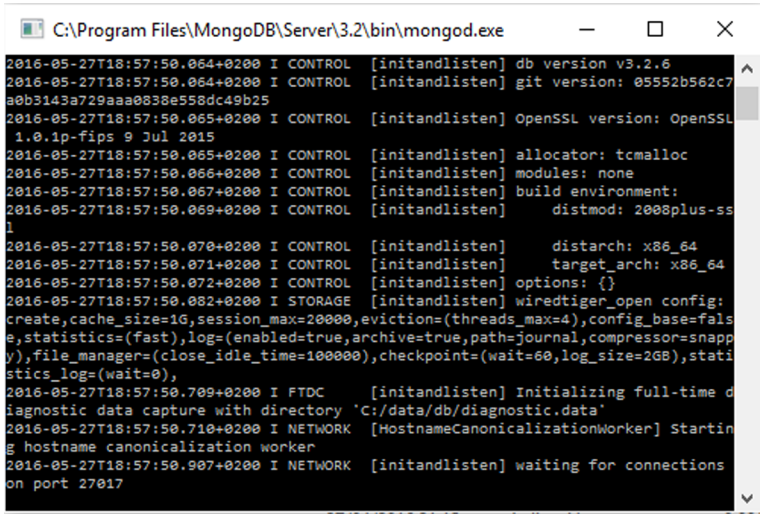
- *mongod*; es el servidor de la base de datos. Antes de poder ejecutar el servidor, es necesario crearse un directorio en el sistema que se usará para almacenar los archivos de la bases de datos. Por defecto, MongoDB intentará

almacenar los archivos en el directorio *C:\data\db*. Es por ello que hay que crear esa estructura de directorios en el sistema.

Una vez que se encuentra instalada, se puede ejecutar siguiendo los pasos siguientes:

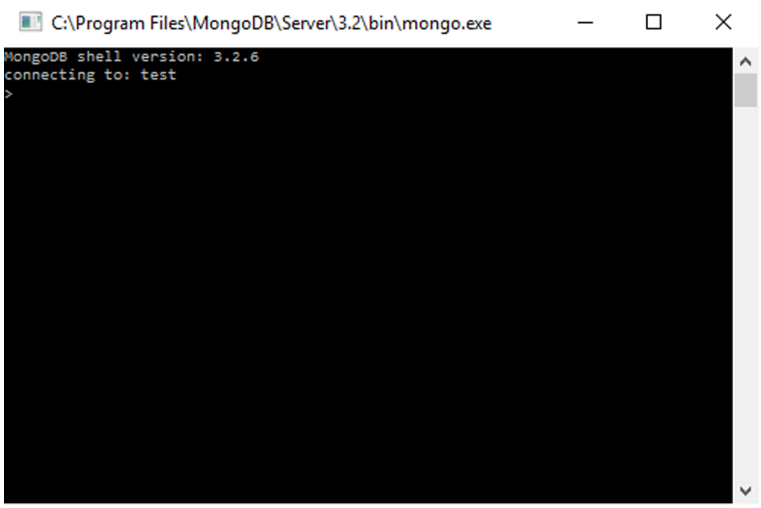
a) Ejecutar el servidor: Para ello, hay que situarse en el directorio *bin* donde se ha realizado la instalación y se pulsa sobre el archivo denominado *mongod* para lanzar el servidor (figura 16).

Figura 16. Servidor de MongoDB



b) Ejecutar la *Shell* de comandos: Para ello, se pulsa sobre el archivo denominado *mongo* situado en el directorio *bin* (figura 17).

Figura 17. *Shell* de MongoDB



Cuando se ejecuta la *Shell* con la configuración por defecto, el servidor se conecta a una base de datos por defecto denominada *test*. Una característica importante de MongoDB es que cuando se intenta conectar a una base de datos que no existe, MongoDB la crea automáticamente. Algunos de los comandos más útiles para empezar a trabajar sobre MongoDB son:

- *show dbs*: muestra los nombres de las bases de datos que existen.
- *show collections*: muestra las colecciones de la base de datos actual.
- *use <db nombre>*: establece la base de datos que se va a usar.

2) Instalación sobre Linux Ubuntu

Para realizar la instalación se necesitan los paquetes con extensión .deb que se pueden encontrar en el sitio oficial de descargas de MongoDB:

- *mongodb-org*: Es un paquete que realizará la instalación automática de los restantes paquetes.

- *mongodb-org-server*: Contiene el proceso *mongod* y su configuración asociada, así como una serie de *scripts* de inicio.

- *mongodb-org-mongos*: Contiene el proceso *mongod*.

- *mongodb-org-shell*: Contiene la *Shell* de comandos *mongod*.

- *mongodb-org-tools*: Contiene las herramientas: *mongoimport*, *bsondump*, *mongodump*, *mongoexport*, *mongofiles*, *mongooplog*, *mongoperf*, *mongorestore*, *mongostat* y *mongotop*.

Los pasos para instalarlo son:

a) Importar la clave pública usada por el sistema de gestión de paquetes: Las herramientas de gestión de paquetes aseguran la consistencia y autenticidad de los paquetes obligando a los distribuidores a firmar los paquetes con claves GPG. Para importar la clave GPG pública de MongoDB, se utiliza el comando:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
```

b) Crear un archivo lista para MongoDB: Se crea el archivo lista */etc/apt/sources.list.d/mongodb-org-3.2.list* mediante el comando:

```
echo "deb http://repo.mongodb.org/apt/ubuntu precise/mongodb-org/3.2 multiverse"
```

c) Actualizar la base de datos local de paquetes. Se ejecuta el comando:

```
sudo apt-get update
```

d) Instalar los paquetes de MongoDB. Se instala la última versión estable de MongoDB ejecutando el comando:

```
sudo apt-get install -y mongodb-org
```

Una vez que se encuentra instalada, se puede ejecutar siguiendo los pasos siguientes:

a) Ejecutar MongoDB. Para ello se ejecuta el comando:

```
sudo service mongod start
```

b) Verificar que MongoDB ha comenzado su ejecución correctamente. Se ejecuta el comando:

```
[initandlisten] waiting for connections on port <port>
```

donde *<port>* es el puerto en el que se ejecuta, que por defecto es 27017.

c) Parar MongoDB. Para ello, se ejecuta el comando para el proceso *mongod*:

```
sudo service mongod stop
```

d) Reiniciar MongoDB. Se ejecuta el comando:

```
sudo service mongod restart
```

e) Conexión a la *Shell* de comandos de MongoDB. Se ejecuta el comando:

```
mongo
```

II. Guía de instalación de Robomongo

Es una herramienta que permite gestionar gráficamente bases de datos MongoDB. En el momento de escribir este material, solo está disponible para máquinas de 32 bits. Presenta entre otras funcionalidades:

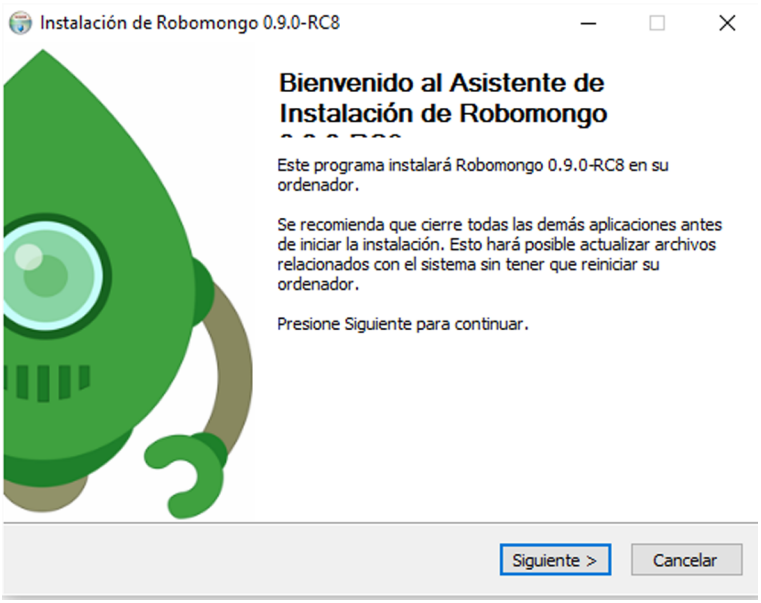
- Múltiples conexiones a las bases de datos, separadas por pestañas.
- Resaltado de sintaxis y autocompletado del código.
- Distintos modos de visualización.

A continuación, se va a explicar cómo llevar a cabo la instalación en Windows y Linux.

1) Instalación en Windows

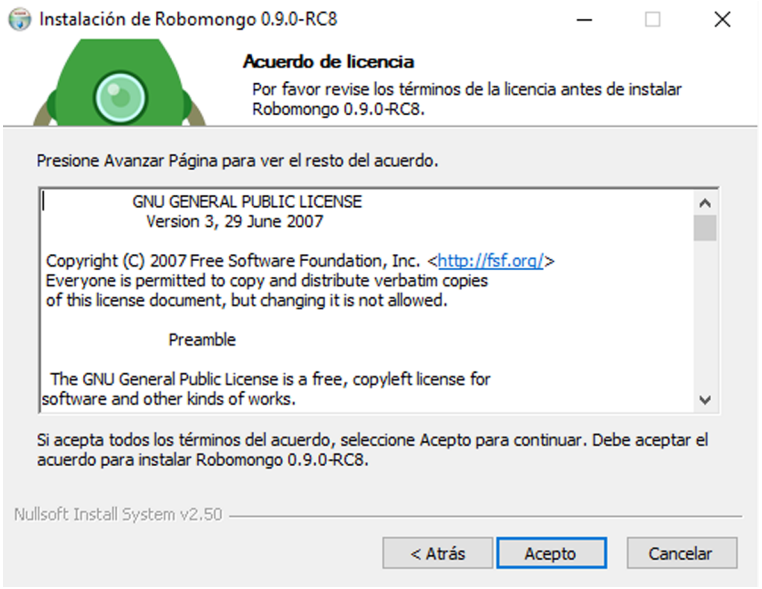
En primer lugar, hay que descargase el instalador de la página oficial. Una vez descargado, se pulsa sobre el mismo y se siguen los pasos (figura 18).

Figura 18. Pantalla de instalación



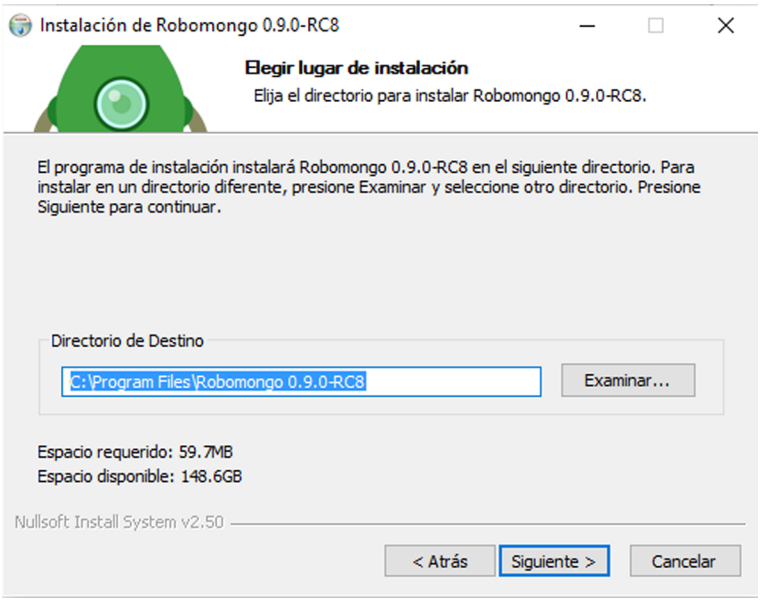
En la siguiente pantalla se pulsa sobre *Acepto* (figura 19).

Figura 19. Pantalla de inicio de instalación



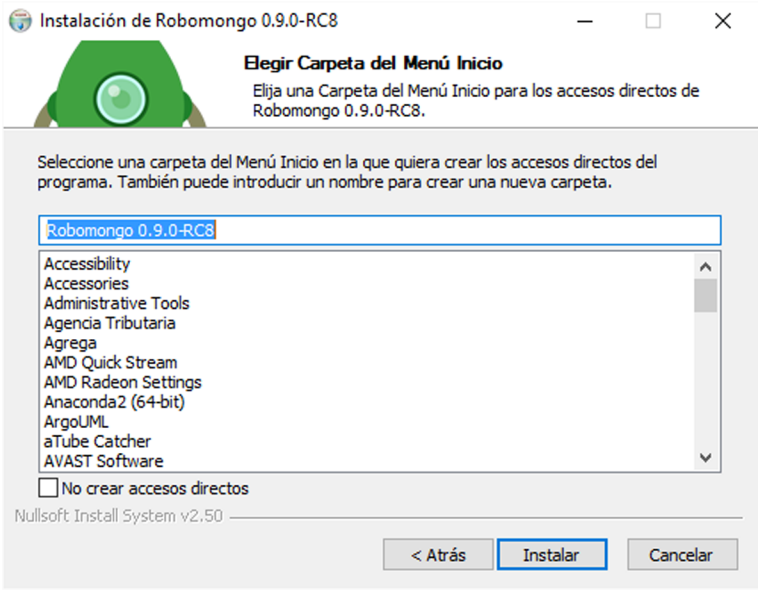
En la siguiente pantalla se pulsa sobre *Siguiente* (figura 20).

Figura 20. Pantalla de selección de directorio



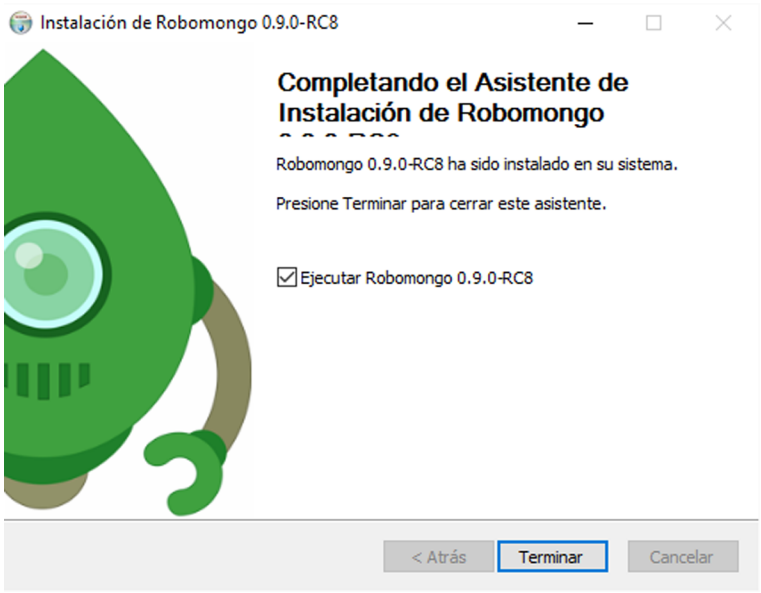
En la siguiente pantalla se pulsa sobre *Instalar* (figura 21).

Figura 21. Pantalla de configuración



Una vez que se ha completado el proceso, aparece la pantalla siguiente (figura 22).

Figura 22. Fin de la instalación



Una vez que se ha completado la instalación, se ejecuta, y aparece la siguiente pantalla (figura 23). Antes de seguir, hay que ejecutar el servidor *mongod* de MongoDB.

Figura 23. Pantalla de inicio Robomongo

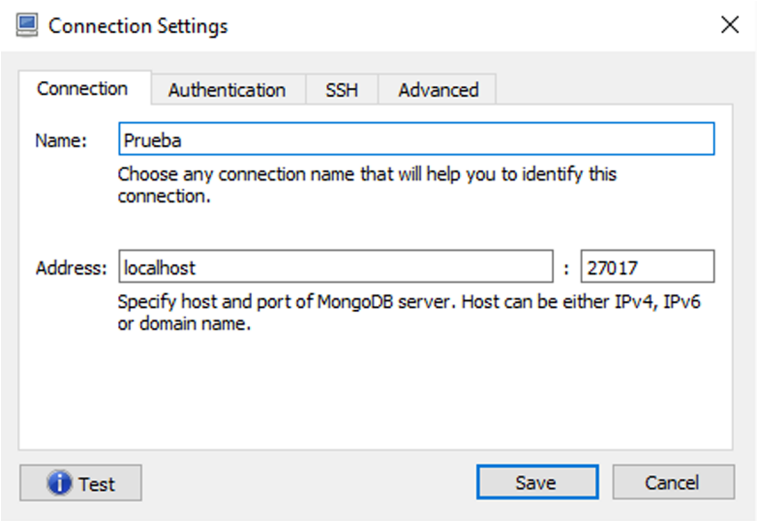
Lo primero que hay que hacer es crear una conexión pulsando sobre *Create*, lo que hace que aparezca una ventana donde hay que rellenar:

- En la pestaña *Connection*: Nombre de la conexión y dirección (por defecto *localhost*).
- En la pestaña *Authentication*: Usuario que realizará la conexión.
- En la pestaña *Advanced*: Base de datos por defecto.
- En la pestaña *SSL*: Habilitación del protocolo SSL para establecer conexiones seguras.

- En la pestaña *SSH*: Realizar conexiones a través de un túnel SSH.

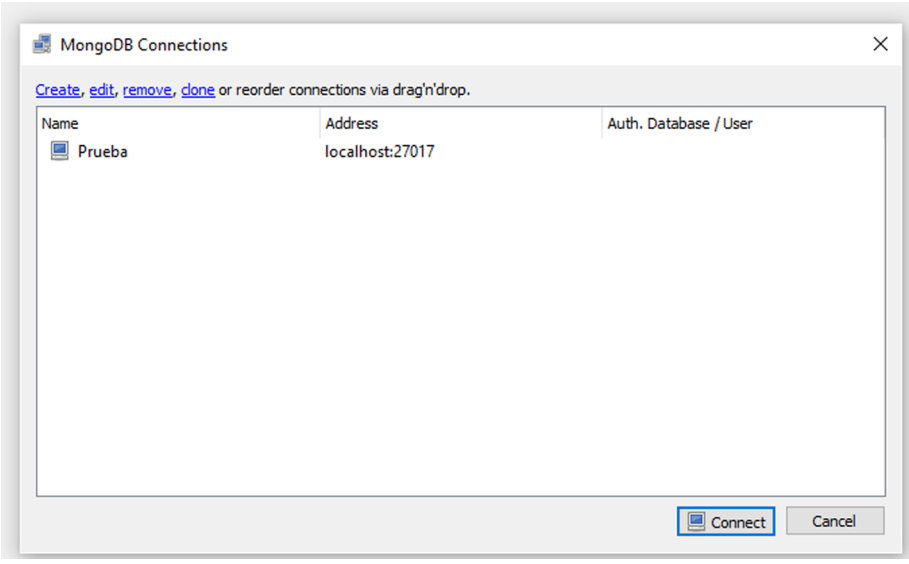
También se puede probar la conexión pulsando sobre *Test*. A continuación se pulsa sobre *Save* para guardar los datos (figura 24).

Figura 24. Guardar datos de la conexión



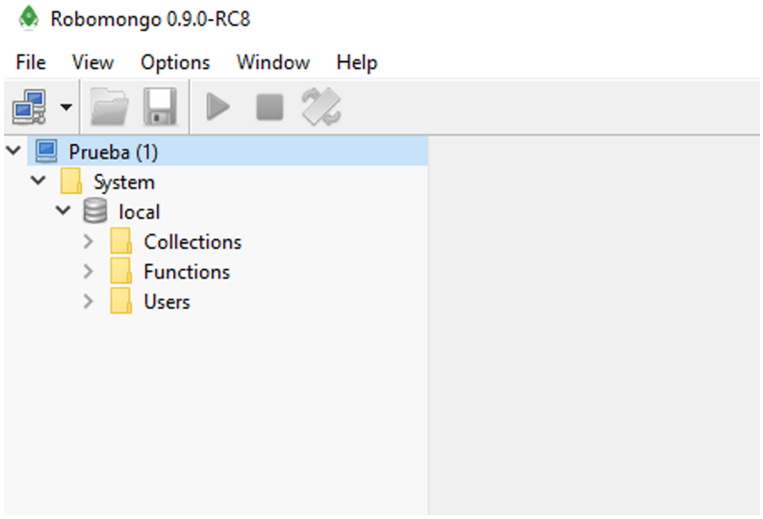
En la pantalla principal aparece la conexión creada. A continuación se selecciona la conexión y se pulsa sobre *Connect* (figura 25).

Figura 25. Conectarse a la base de datos



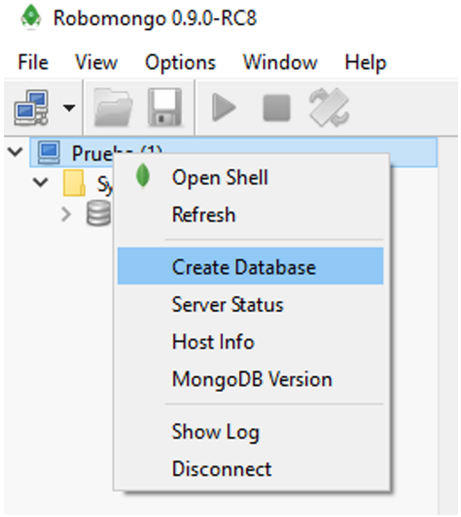
A continuación aparece la siguiente ventana con la conexión (figura 26).

Figura 26. Conexión realizada



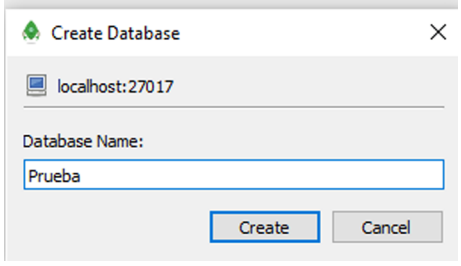
Ahora se puede crear una base de datos pulsando con el botón derecho del ratón sobre el nombre de la conexión y eligiendo la opción *Create Database* (figura 27).

Figura 27. Creación de una base de datos



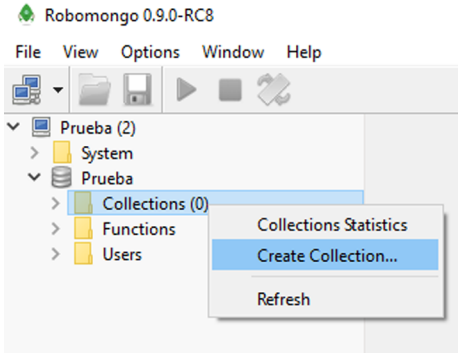
Se le da un nombre y se acepta (figura 28).

Figura 28. Nombre de una base de datos



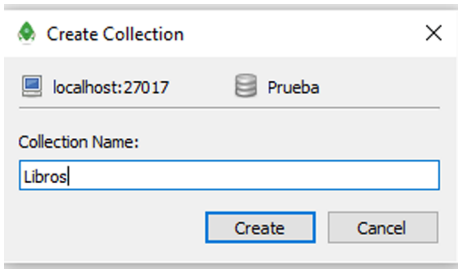
Para crear una colección, se pulsa con botón derecho en la carpeta denominada *Collections* y se elige la opción de *Create Collection* (figura 29).

Figura 29. Creación de una colección



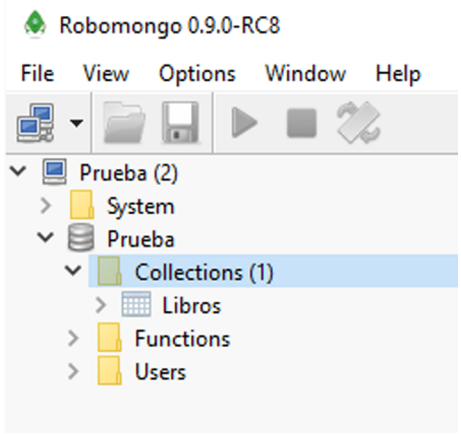
Se introduce el nombre de la colección y se acepta (figura 30).

Figura 30. Nombre de la colección



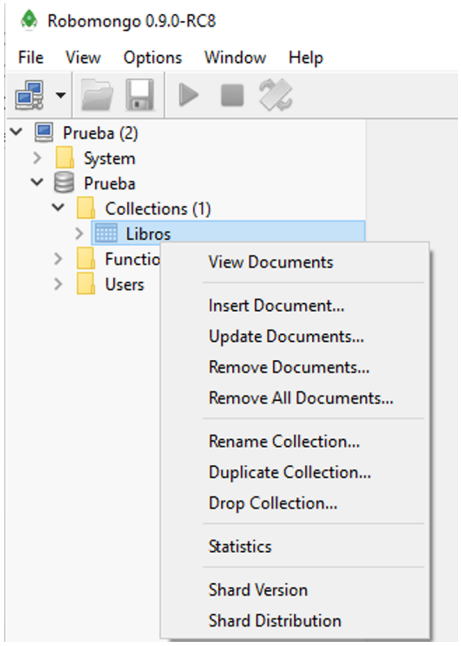
En la ventana principal aparece la nueva colección (figura 31).

Figura 31. Colección creada



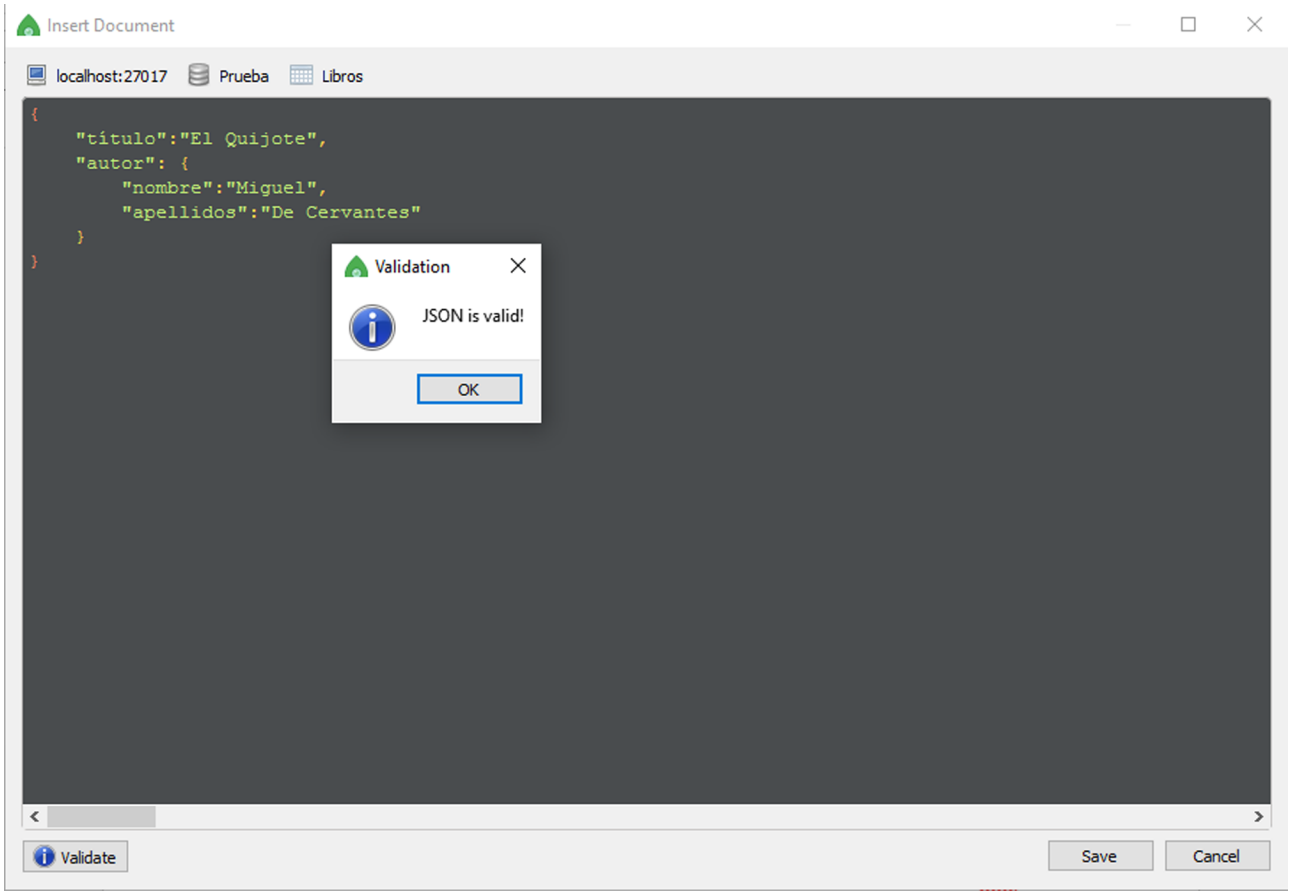
Para insertar un documento se pulsa sobre la colección con botón derecho del ratón y se selecciona la opción *Insert document* (figura 32).

Figura 32. Inserción de un documento



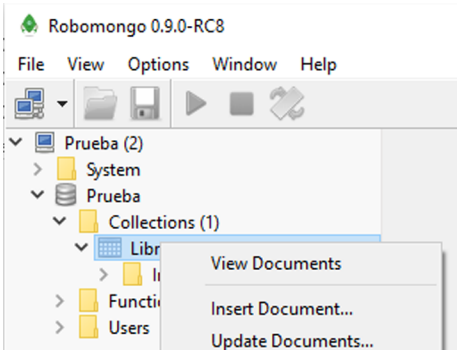
Aparece una ventana de edición en la que se introduce el documento que se quiere añadir. Con el botón *Validate* se puede comprobar que la sintaxis es correcta, y para guardar se pulsa sobre *Save* (figura 33).

Figura 33. Validación de la sintaxis



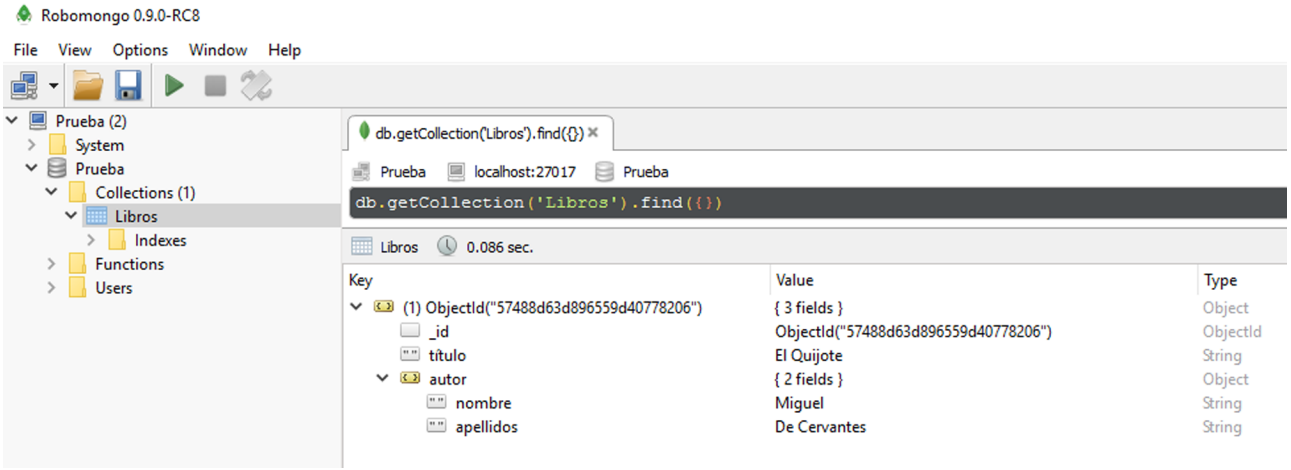
Para ver los documentos de una colección, se pulsa con el botón derecho sobre la misma y se elige la opción *View documents* (figura 34).

Figura 34. Ver documentos de una colección



Y aparecen los documentos que tiene la colección (figura 35).

Figura 35. Documentos de la colección



2) Instalación en Linux

En primer lugar, hay que descargase el paquete *.deb* para distribuciones en Debian como Ubuntu desde la página oficial de la herramienta. Una vez descargado, se accede al directorio donde se encuentre el paquete descargado y se ejecuta. Se hace clic sobre *Instalar paquete*, se introduce la contraseña del usuario y se acepta. En ese momento comenzará la instalación.

III. Guía de sentencias básicas de consultas en MongoDB

1) Comandos referidos a la base de datos

a) Visualizar las bases de datos existentes:

```
show dbs
```

b) Visualizar la base de datos actual:

```
db
```

c) Crear una base de datos:

```
use <Nombre_Base_Datos>
```

d) Eliminar una base de datos actual:

```
db.dropDatabase()
```

2) Comandos referidos a las colecciones

a) Crear una colección:

```
db.createCollection("Nombre_Colección")
```

b) Conocer las colecciones existentes:

```
show collections
```

c) Acceder a una colección concreta se usa el comando:

```
db.getCollection("Nombre_Colección")
```

o

```
db.<Nombre_Colección>
```

d) Eliminar una colección:

```
db.<Nombre_Colección>.drop()
```

3) Comandos referidos a los documentos

a) Para crear un documento se utiliza el comando *"insert"*:

```
db.<Nombre_Colección>.insert( <Nombre_documento> )
```

donde *<Nombre_documento>* es el nombre de una variable que contiene un documento. Por ejemplo, considerar el siguiente ejemplo de documento:

```
documento1={
  "Título": "El Quijote",
  "Autor": {
           "Nombre":"Miguel",
           "Apellidos":"De Cervantes"}}
```

Para insertarlo en una colección llamada libros se haría así:

```
db.libros. insert( documento1)
```

También se podría insertar directamente poniendo como argumento del método el documento propiamente dicho. Por ejemplo:

```
db.libros.insert({
  "Título": "El Quijote",
  "Autor": {
           "Nombre":"Miguel",
           "Apellidos":"De Cervantes"
}})
```

Además, se puede insertar más de un documento a la vez si se llama al método con un conjunto de nombres de variables que contienen documentos encerrados entre corchetes:

```
db.<Nombre_colección>.insert( [documento1,...,documenton])
```

Por ejemplo:

documento1={
“Título”: “El Quijote”,
“Autor”: {
“Nombre”:”Miguel”,
“Apellidos”:”De Cervantes”}}
	documento2={
	“Título”: “La Celestina”,
	“Autor”: {
	“Nombre”:”Fernando”,
	“Apellidos”:”De Rojas”}}

```
db.libros.insert ( [documento1,documento2])
```

b) Para buscar información dentro de un documento, se utiliza el comando *find*:

```
db.<Nombre_Colección>.find(<condición>,<salida>)
```

donde *<condición>* representa la condición de búsqueda usada para recuperar la información y *<salida>* establece qué forma debe tener el resultado devuelto.

Por ejemplo, considerar la anterior colección *libros* y los documentos que se han insertado en el ejemplo anterior. Se quiere recuperar los documentos donde el título es *El Quijote* y de los documentos recuperados solo se quiere mostrar la información de su autor:

```
db.libros.find({"Título":"El Quijote"}, {"Titulo":0,"Autor":1, "_id":0})
```

En la descripción de la salida algunos campos toman el valor *0* y otros *1*. Esto permite indicar a MongoDB qué campos del resultado se quieren mostrar y cuáles se quieren dejar ocultos. En este caso, se le ha indicado que solo debería devolver como resultado de la consulta el campo autor del documento que cumple las condiciones de búsqueda. Por tanto, devolvería el siguiente resultado:

```
{"Autor": {
    "Nombre":"Miguel",
    "Apellidos":"De Cervantes"}
}
```

Cuando se quiere realizar una consulta sobre un campo que está dentro de otro campo como es el caso del campo *Nombre* que es un subcampo del campo *Autor*, entonces se hace referencia al subcampo mediante la notación *campo.subcampo*. Por ejemplo, se quieren recuperar los títulos de autores cuyo nombre sea *Miguel*:

```
db.libros.find({"Autor.Nombre":"Miguel"}, {"Título":1,"Autor":0,"_id":0})
```

También es posible realizar una búsqueda en la que se utilicen varias condiciones de búsquedas. En este caso, las condiciones deben aparecer separadas por comas. Por ejemplo, se quieren recuperar los documentos de los autores cuyo nombre sea *Miguel* y el título sea *El Quijote*:

```
db.libros.find({"Autor.Nombre":"Miguel","Título":"El Quijote"},
                {"Título":1,"Autor":1,"_id":0})
```

Cuando se están buscando documentos en los que un determinado campo puede tomar varios valores, se puede usar el operador *$in* mediante el cual se puede indicar el conjunto de valores que podría tomar el campo. Por ejemplo, se quieren recuperar los documentos de los autores cuyo nombre sea *Miguel* o *Fernando*:

```
db.libros.find({"Autor.Nombre":{$in:["Miguel","Fernando"]}},
                {"Título":1,"Autor":1,"_id":0})
```

Cuando en una consulta se espera que se recuperen muchos resultados, puede ser interesante ordenar los resultados de acuerdo al valor que toma alguno de los campos. En estos casos, se puede usar el método *sort* que toma como parámetros el campo o campos por los que se quiere ordenar los resultados, y el valor *1* o *-1* para indicar si la ordenación se realiza en orden ascendente o descendente. Por ejemplo, se quieren recuperar los documentos referidos a autores cuyo nombre es *Miguel* ordenando los resultados por orden alfabético ascendente de *Título*:

```
db.libros.find({"Autor.Nombre":"Miguel"},{ "_id":0}).sort({"Título":1})
```

c) Para actualizar un documento, se usa el comando *update*:

```
db.<Nombre_Colección>.update(<condición>,<actualización>)
```

donde *<condición>* representa la condición de búsqueda usada para recuperar la información y *<actualización>* establece qué modificaciones deben realizarse sobre el documento o documentos que se han recuperado. Por ejemplo, se quiere recuperar el documento asociado al autor cuyo nombre es *Miguel* y se quiere modificar su campo *Título* para que en vez de almacenar *El Quijote*, almacene *Novelas Ejemplares*:

```
db.libros.update({"Autor.Nombre":"Miguel"},{$set:{"Título":"Novelas Ejemplares"}})
```

Por defecto, se modifica el primer documento que se encuentra en la base de datos que cumple las condiciones, pero si hubiera más de un documento cumpliendo las condiciones indicadas no se modificarían. Si se quiere que se modifiquen todos los documentos que cumplen las condiciones de búsqueda, es necesario añadir el campo *multi*. El ejemplo anterior sería:

```
db.libros.update({"Autor.Nombre":"Miguel"},{$set:{"Título":"Novelas
                  Ejemplares"}},{"multi":true})
```

Cuando se realiza una actualización, se pueden añadir nuevos campos al documento usando el operador *$addToSet*. Por ejemplo, se quiere recuperar el documento asociado al autor cuyo nombre es *Miguel* y se quiere añadir un campo *Idioma* para que almacene el idioma en el que está escrito:

```
db.libros.update({"Autor.Nombre":"Miguel"},
                  {$addToSet:{"Idioma":"Castellano"}})
```

También se considera una actualización, la eliminación de un campo de un documento. Para ello se utiliza el operador *$unset*. Por ejemplo, se va a eliminar el campo *Idioma* del documento asociado al autor cuyo nombre es *Miguel*:

```
db.libros.update({"Autor.Nombre":"Miguel"},
                  {$unset:{"Idioma":""}})
```

d) Por último, si se quiere eliminar un documento o documentos de una colección, se usa el comando *remove*:

```
db.<Nombre_Colección>.remove(<condición>)
```

donde *<condición>* representa la condición de búsqueda usada para recuperar la información. Por ejemplo, si se quieren eliminar todos los documentos de la colección libros, se haría:

```
db.libros.remove({})
```

Y si se quieren eliminar el documento asociado al autor con nombre *Miguel*:

```
db.libros.remove({"Autor.Nombre":"Miguel"})
```