FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Un Análisis Comparativo de Bases de Datos para Dispositivos Móviles

AUTORES: Fernando Tesone

DIRECTOR: Mg. Pablo Thomas

CODIRECTOR: -

ASESOR PROFESIONAL: -

CARRERA: Licenciatura en Informática

Resumen

Con el crecimiento en el alcance y uso de internet, de los smartphones, y de las redes sociales, se está produciendo un aumento exponencial en el volumen de datos administrados, pudiendo ser éstos estructurados, semiestructurados, o sin estructura. En este contexto surgen las bases de datos NoSQL, que facilitan el almacenamiento de datos semiestructurados o sin estructura.

Las mejoras en las prestaciones de hardware de los dispositivos móviles llevan a que éstos administren cada vez más información, lo que hace que surjan nuevos sistemas de gestión de bases de datos que se instalan en los dispositivos. Este trabajo de investigación tiene por objetivo realizar un relevamiento de los sistemas de gestión de bases de datos que se instalan en dispositivos móviles existentes en la actualidad, y realizar un análisis de los sistemas más representativos de los tipos más utilizados.

Palabras Clave

Bases de Datos para Dispositivos Móviles, DBMS Relacional, DBMS NoSQL

Conclusiones

Se considera que la librería Room que utiliza Android como capa de abstracción de SQLite lleva a que se genere código más legible que utilizando SQLite sin Room o Couchbase Lite.

Por otro lado, se considera que Couchbase Lite se adapta mejor a modelos de datos con propiedades opcionales o con estructuras no definidas claramente.

El modelo de datos del problema a resolver es determinante para elegir el DBMS más adecuado.

Trabajos Realizados

Se realizó una investigación exhaustiva de bases de datos para dispositivos móviles.

Se realizó una descripción de los todos los DBMSs encontrados para dispositivos móviles.

Se realizó una selección de dos DBMSs representativos del tema.

Se realizó una experimentación utilizando estos dos DBMSs.

Trabajos Futuros

Extender el análisis realizado utilizando otros DBMSs, priorizando aquellos disponibles para Android e iOS, y utilizados en los principales frameworks de desarrollo de aplicaciones móviles.

Fecha de la presentación: MARZO 2021

Un Análisis Comparativo de Bases de Datos para Dispositivos Móviles

Tesone, Fernando

Marzo de 2021

Resumen

Con el crecimiento en el alcance y uso de internet, de los *smartphones*, y de las redes sociales, se está produciendo un aumento exponencial en el volúmen de datos administrados, pudiendo ser éstos estructurados, semi-estructurados, o sin estructura. En este contexto surgen las bases de datos NoSQL, que facilitan el almacenamiento de datos semi-estructurados o sin estructura.

Las mejoras en las prestaciones de hardware de los dispositivos móviles, llevan a que éstos administren cada vez más información, lo que hace que surjan nuevos sistemas de gestión de bases de datos que se instalan en los dispositivos. Este trabajo de investigación tiene por objetivo realizar un relevamiento de los sistemas de gestión de bases de datos que se instalan en dispositivos móviles existentes en la actualidad, y realizar un análisis de los sistemas más representativos de los tipos más utilizados.

Palabras clave: Bases de Datos para Dispositivos Móviles, DBMS Relacional, DBMS NoSQL.

Índice general

1. Introducción **4**

1.1. Motivación 4

1.2. Objetivos 6

1.3. Metodología 7

1.4. Organización 9

2. Trabajos relacionados **10**

3. Bases de Datos para Dispositivos Móviles **14**

3.1. DBMSs Relacionales 17

3.1.1. Selección de DBMS Relacional 19

3.2. DBMSs NoSQL 20

3.2.1. Documentales 21

3.2.2. Otros tipos 26

3.2.3. Selección de DBMS NoSQL 28

4. Experimentación con los DBMSs seleccionados **29**

4.1. SQLite 31

4.2. Couchbase Lite 37

5. Análisis de Resultados **44**

5.1. SQLite 44

5.2. Couchbase Lite 46

6. Conclusiones y Trabajo Futuro **48**

Índice de figuras

4.1. Diagrama conceptual de la PoC 31

4.2. Diagrama correspondiente al modelo lógico para RDBMS . . 31

4.3. Diagrama correspondiente al modelo físico para SQLite . . . 31

4.4. Diagrama correspondiente al modelo físico para Couchbase Lite 38

Capítulo 1

Introducción

En este capítulo se presenta la motivación de esta tesina, y se detallan los objetivos, la metodología de trabajo, y la organización del documento

1.1. Motivación

Las bases de datos surgen de la necesidad de almacenar, organizar y recuperar información interrelacionada, y tienen un origen previo a la creación de las computadoras. En la década de 1960, a partir de un incremento en la popularidad y disminución de costos de las computadoras, y de la informatización de las bases de datos, comienzan a desarrollarse los primeros sistemas de gestión de bases de datos (DBMS) [1]. Estos sistemas jugaron un rol fundamental en el desarrollo de software y el crecimiento de productos de software, ya que proveían una forma eficiente de desarrollar aplicaciones complejas, al eliminar la necesidad de programar el almacenamiento y recuperación de información para cada aplicación, la posibilidad de compartir información entre distintas aplicaciones y/o usuarios, y el desarrollo de interfaces y lenguajes estandarizados, entre otras características [2].

Inicialmente se desarrollaron dos modelos de base de datos: el modelo

de red y el modelo jerárquico [1]. En 1970, Edgar Codd introduce un nuevo modelo de base de datos, el modelo relacional, que a partir de entonces, y hasta la actualidad, se volvió el modelo dominante [1, 3].

Hasta la década de 1990 el uso de internet estaba dedicado principalmente a actividades científicas y académicas, y bajo desarrollo y control estatal, fundamentalmente en Estados Unidos. En la primera mitad de la década se produjeron dos cambios que comenzaron a popularizar su uso: la extensión de internet al ámbito privado, que llevó a su uso comercial, y el desarrollo de la World Wide Web en 1993 por parte de Tim Berners-Lee, que facilitó el uso de internet a personas no especializadas. Estos dos factores, sumado a los avances en el desarrollo de hardware, llevaron a un incremento en las ventas de computadoras personales hacia la segunda mitad de la década de 1990 [4].

En 1997 se acuña por primera vez el término *smartphone*, como forma de denominar a dispositivos móviles con capacidad de instalar y ejecutar programas de software (*apps*), pudiendo considerarse una computadora de mano con teléfono incorporado, y siendo una evolución de los teléfonos móviles surgidos a principios de la década de 1990, y luego de los denominados *feature phones* (teléfonos móviles con ciertas características integradas, como cámara de fotos y/o video, reproducción de archivos multimedia, entre otras). Sin embargo, no es hasta 2007, con la presentación del *Apple iPhone*, y hasta 2008, con la introducción del sistema operativo *Android*, que la industria de los *smartphones* cambiaría radicalmente, incrementando la popularidad del uso de dispositivos móviles, llegando al 80 % de la población en algunos países [4, 5, 6]. Este crecimiento en el uso traería acoplado una diversificación de plataformas.

Para maximizar la presencia en el mercado, las *apps* deben estar disponibles en múltiples plataformas, por lo que los desarrolladores de software deben optar por realizar desarrollos nativos, específicos de cada plataforma, o desarrollos multiplataforma —con la posibilidad de utilizar distintos enfoques, entre los que se pueden mencionar híbrido, interpretado, de com-

pilación cruzada— [7], que permiten utilizar el mismo código fuente para generar aplicaciones que estén disponibles en más de una plataforma.

Con el crecimiento en el alcance y uso de internet y de los dispositivos móviles, sumado a la aparición de las redes sociales, se está produciendo un crecimiento exponencial en el volúmen de datos administrados [8], pudiendo ser éstos estructurados, semi–estructurados o sin estructura. Ante esta situación de crecimiento en el volúmen de información, surgen como alternativa a las bases de datos relacionales, las bases de datos no relacionales o NoSQL (Not–only SQL), ya que facilitan el almacenamiento masivo de datos semi–estructurados o no estructurados.

Con la aparición de estas tecnologías, los desarrolladores de software deben analizar cuáles DBMSs son adecuados para las necesidades del problema a resolver.

1.2. Objetivos

El objetivo de esta tesina es investigar el uso de DBMSs para dispositivos móviles, y realizar un análisis comparativo que permita orientar al ingeniero de software en el proceso de selección de un DBMS a ser instalado en un dispositivo móvil.

Para ello, se propone realizar un relevamiento exhaustivo de los diferentes DBMSs, tanto relacionales (RDBMS) como no relacionales (DBMS NoSQL), utilizados en el desarrollo de software (nativo o multiplataforma) para dispositivos móviles. Se seleccionarán aquellos DBMSs más representativos de cada tipo que frecuentemente se instalan en dispositivos móviles y se realizará un análisis comparativo, considerando características específicas, ventajas y desventajas, desde el punto de vista de la experiencia del desarrollador.

Se espera que este trabajo de investigación permita al ingeniero de software mejorar el proceso de selección de un DBMS adecuado para dispositivos

móviles, de acuerdo a las necesidades del problema que se pretende resolver.

1.3. Metodología

La realización de este trabajo está dividida en dos partes.

La primera parte consiste en un relevamiento exhaustivo de RDBMSs y DBMSs NoSQL que pueden utilizarse en el desarrollo de aplicaciones móviles.

Se presenta un listado detallado de todos los DBMSs para dispositivos móviles encontrados. La búsqueda se realizó a través de los buscadores Google y Google Scholar, utilizando los términos "mobile database", "mobile dbms", "bases de datos móviles", entre otros. También se realizó una búsqueda en Google con el siguiente término: "mobile site:db-engines.com/en/system", ya que el sitio DB-Engines utiliza ese *path* para las páginas en las que se describen características de cada DBMS relevado por el sitio.

De cada DBMS para dispositivos móviles encontrado se describe información básica relacionada al desarrollo y mantenimiento, licencia, popularidad, independencia en el almacenamiento de datos, y disponibilidad en las diversas plataformas de desarrollo nativo y multiplataforma de aplicaciones móviles.

La segunda parte consiste en un análisis comparativo, desde el punto de vista de experiencia del desarrollador, de un RDBMS y un DBMS NoSQL, seleccionados considerando las siguientes características: licencia, popularidad, independencia en almacenamiento de datos, disponibilidad en Android y iOS, y disponibilidad en los principales frameworks de desarrollo multiplataforma.

En relación a la licencia de uso, en el proceso de selección de los DBMSs a analizar se considerarán aquellos que cuenten con una licencia de uso libre,

sin descartar los DBMSs que también cuenten con licencia de uso comercial (paga) y no limiten funcionalidades en la/s licencia/s de uso libre.

Respecto a la popularidad, resulta de interés analizar DBMSs que sean ampliamente usados, ya que el objetivo de este trabajo es ayudar al ingeniero de software en la elección de un DBMS para aplicaciones en dispositivos móviles, y es altamente probable que se evalúe seleccionar los de mayor uso.

La independencia en el almacenamiento de datos refiere a la existencia de DBMSs que son basados en la nube, y que la persistencia de datos en el dispositivo es sólo circunstancial, como una forma de tener disponible de forma inmediata datos frecuentemente accedidos, y mitigar problemas de conexión, recurrentes en dispositivos móviles. Se busca analizar DBMSs que, aunque cuenten con soporte de sincronización de datos con servidores centrales o servicios en la nube, puedan realizar persistencia de datos en el dispositivo móvil sin su utilización.

En la actualidad, los sistemas operativos para dispositivos móviles más utilizados son Android y iOS. La cuota de mercado de Android corresponde a un 71.9 %, y la de iOS a un 27.3 %, resultando en un total de 99.2 %. Por este motivo, resulta de gran interés la selección de DBMSs que estén disponibles para ser utilizados de forma nativa en ambas plataformas.

Considerando que la disponibilidad de una aplicación en ambas plataformas utilizando un enfoque de desarrollo nativo implica que se generen, desarrollen y mantengan dos productos de software distintos, han surgido y se han popularizado frameworks de desarrollo de aplicaciones móviles multiplataforma que, utilizando distintos enfoques, permiten generar aplicaciones móviles para Android y iOS, logrando una reusabilidad mínima de código de entre un 50 % y un 80 % [9].

0Entre los principales frameworks se encuentran React Native (enfoque interpretado), Flutter (enfoque de compilación cruzada), Cordova (enfoque híbrido), Ionic (híbrido, hace uso de plugins de Cordova), Xamarin (compilación cruzada), Unity (compilación cruzada), PhoneGap (híbrido, es una

distribución de Cordova), NativeScript (interpretado) [10]. Para el análisis de DBMSs que se realizará en este trabajo de tesina, interesa que éstos se encuentren disponibles en la mayor cantidad posible de los principales frameworks de desarrollo multiplataforma.

1.4. Organización

El resto del documento está organizado de la siguiente manera:

En el segundo capítulo se presentan trabajos de investigación relacionados al tema de esta tesina.

En el tercer capítulo se describen todos los sistemas de gestión de bases de datos para aplicaciones móviles encontrados, como resultado del relevamiento realizado. El capítulo se divide en tres secciones: la primera, detalla los RDBMSs encontrados; la segunda, los DBMSs NoSQL documentales; en la última se detallan DBMSs NoSQL correspondientes a tipos distintos al documental.

En el cuarto capítulo se experimenta en el desarrollo de una prueba de concepto que permita analizar características específicas, y ventajas y desventajas, de los DBMSs seleccionados.

En el quinto capítulo se analizan los resultados obtenidos a partir de la experimentación realizada.

En el sexto y último capítulo se presentan las conclusiones del trabajo, y se definen posibles líneas de investigación para trabajos futuros.

Capítulo 2

Trabajos relacionados

En este capítulo se realiza una recopilación de trabajos de investigación relacionados al tema de la presente tesina.

En [11] se expone un listado de características que los sistemas de gestión de bases de datos móviles y embebibles deben tener. Éstas características son:

- integrable en aplicaciones:

  Los DBMSs deberían formar parte de la aplicación o de la infraestructura de la aplicación, sin requerir administración. Debe ser distribuída como parte de la aplicación;

- bajo impacto (*small footprint*):

  considerando que el DBMS forma parte de la aplicación, el tamaño del primero afecta al tamaño final que tendrá la aplicación[1];

- ejecutable en dispositivos móviles:

[1] Nota del autor: esto influye directamente en el éxito que tendrá la aplicación, ya que el tamaño que ocupe en el dispositivo representa un requerimiento no funcional de gran relevancia [12, 13, 14]

además de manejar las limitaciones inherentes a los entornos móviles, los DBMSs móviles deben adaptarse a los sistemas operativos disponibles; asimismo, deben tener la capacidad de sincronizar datos con DBMSs de backend;

- DBMS componentizado:

  es importante que sea posible incluir sólo los componentes del DBMS que se requieren para el funcionamiento de la aplicación[2];

- DBMSs en memoria:

  los DBMSs deberían soportar el uso de almacenamiento de datos en memoria, es decir, datos que no se persisten;

- bases de datos portables:

  Las bases de datos del DBMS deberían ser altamente portables, idealmente un único archivo, que permita copiar la base de datos en una instalación o migración;

- sincronización con fuentes de datos backend:

  los DBMSs deberían soportar la sincronización de datos con servidores de bases de datos en backend.

En [15] se realiza un relevamiento de diferentes opciones de almacenamiento en dispositivos móviles, siendo éstas:

- HTML5:

  Android y iOS utilizan el framework WebKit[3], dando la posibilidad de que se utilice la API *localStorage* para almacenar datos;

- SQLite:

[2] Nota del autor: esto se debe a que requerimientos no funcionales como el tamaño resultante de la aplicación son fundamentales para su éxito

[3] Nota del autor: esta característica permite el desarrollo de aplicaciones multiplataforma híbridas

Es una de las opciones de almacenamiento de datos más populares en el mercado. Es una librería que encapsula funcionalidad SQL, pero que almacena sus datos en un archivo local en lugar de una base de datos remota;

- almacenamiento en la nube:

  Podrían utilizarse una amplia variedad de servicios de almacenamiento en la nube, como pueden ser *Apple iCloud*, *Google Drive*, *Dropbox*, *Amazon S3*;

- almacenamiento específico del dispositivo:

  las opciones anteriores se encuentran disponibles en Android y en iOS. Sin embargo, cada plataforma ofrece opciones propias.

  En Android se cuenta con *Shared Preferences*: un mecanismo para almacenar datos primitivos en pares de clave-valor.

  En iOS es posible utilizar *Core Data*, un framework de modelado de datos que usa *Cocoa Touch* que se basa en el patrón de diseño MVC. También es posible almacenar ajustes de configución y datos de usuario en archivos XML.

En [16] se argumenta que en las aplicaciones móviles se presentan varias características de la "clásica" arquitectura de aplicaciones cliente/servidor o web multicapa, pero que la diferencia más notable entre las bases de datos web y móviles está en la menor cantidad de recursos de hardware disponibles en los ambientes móviles.

En [17] se realiza una descripción de la arquitectura de la plataforma Android, que cuenta con:

- Kernel Linux: se encarga de servicios básicos del sistema, como la seguridad, administración de memoria, administración de procesos, pila de red, modelo de drivers. También actúa como una capa de abstracción entre el hardware y el resto del software.

- Librerías: Android incluye un conjunto de librerías C/C++, librerías de multimedia, motor de gráficos 2D y 3D, SQLite, entre otras.
- *Android Runtime*: contiene un conjunto de librerías que provee la mayoría de las funcionalidades disponibles en las librerías base de Java, y la máquina virtual (VM) Dalvik; cada aplicación se ejecuta sobre su propia instancia de esta VM.
- Framework de aplicación: provee a las aplicaciones acceso al hardware y otros servicios del sistema operativo.
- Aplicaciones: Android cuenta con un conjunto definido de aplicaciones.

También se analiza la arquitectura de las aplicaciones, que se componen de:

- *Activities*: implementa la interfaz de usuario (UI) visible; cuando un usuario selecciona un ícono para abrir una aplicación, está iniciando una *activity*.
- Servicios: pueden utilizarse para tareas como monitorear la red o buscar actualizaciones para la aplicación.
- Proveedores de contenido: administran el acceso a la información almacenada.
- Receptores de emisiones: se utilizan para responder a determinados eventos.

Finalmente, se hace una introducción al uso de SQLite en Android.

Capítulo 3

Bases de Datos para Dispositivos Móviles

En este capítulo se presenta un detalle de DBMSs relacionales y no relacionales o NoSQL para dispositivos móviles, en el que se descibe, para cada DBMS, disponibilidad según plataformas y frameworks de desarrollo multiplataforma, licencia, información sobre el desarrollo, y otras características específicas del DBMS que se consideren relevantes.

El detalle se encuentra dividido en DBMSs relacionales y DBMSs NoSQL, y ordenado de forma ascendente según el ranking de DBMSs elaborado por el sitio web DB-Engines (db-engines.com) [3]. DB-Engines es un proyecto que busca recopilar y presentar información sobre DBMSs, creada y mantenida por solidIT, una compañía austríaca especializada en desarrollo de software, consultoría y formación para aplicaciones centradas en datos [18]. El ranking de DB-Engines está elaborado según la puntuación que obtenga cada DBMS según la popularidad que tenga, definida a partir de los siguientes parámetros:

- número de menciones en sitios web, según resultados de búsqueda realizados en los buscadores *Google* (google.com) y *Bing* (bing.com);

- interés general en el DBMS, a partir de la frecuencia de búsquedas según *Google Trends* (trends.google.com);
- frecuencia de discusiones técnicas sobre el DBMS, utilizando la cantidad de preguntas relacionadas en los sitios *Stack Overflow* (stackoverflow.com) y *DBA Stack Exchange* (dba.stackexchange.com);
- número de ofertas laborales en las que se menciona el DBMS, según los sitios *Indeed* (indeed.com) y *Simply Hired* (simplyhired.com);
- número de perfiles en redes sociales profesionales en los que se menciona el DBMS, utilizando *LinkedIn* (linkedin.com);
- relevancia en redes sociales, contando la cantidad de publicaciones en que se menciona el DBMS en *Twitter* (twitter.com) [19].

El sitio DB-Engines cuenta con un ranking general en el que se listan todos los DBMSs relevados, y permite filtrar el ranking general según los tipos de bases de datos (obteniendo de esta forma rankings según el tipo de DBMS).

A continuación se expone la posición en el ranking y el nombre de los DBMSs que se pueden instalar en dispositivos móviles:

#9. SQLite

#26. Couchbase Lite

#37. Firebase Realtime Database

#45. Realm

#51. Google Cloud Firestore

#59. Interbase

#61. SAP SQL Anywhere

#87. Oracle Berkeley DB

#88. PouchDB

#141. SQLBase

#144. LiteDB

#195. ObjectBox

#289. Sparksee

Como parte del relevamiento realizado, también se encontraron los siguientes productos [20], que se descartaron para analizarlos en esta tesina ya que fueron discontinuados (a excepción del último):

- Oracle Database Lite:

  La última versión de este producto fue publicada en 2010 [21].

- Microsoft SQL Server Compact:

  La última versión (4.0) fue publicada en 2011, con fecha de fin en 2016 [22].

- Microsoft SQL Server Express:

  No se encontró documentación oficial de este DBMS, pero sí un listado de descargas que lleva a una página inexistente [23].

- IBM DB2 Everyplace:

  La última versión (V9.1) fue publicada en 2006. [24]

- HFSQL:

  Si bien es un DBMS que se puede instalar en Android y en iOS, sólo puede ser utilizado en sistemas WinDev (producto de la misma empresa desarrolladora, PC SOFT) [25].

3.1. DBMSs Relacionales

Las bases de datos relacionales son bases de datos que se basan en el modelo relacional que fuera presentado por Edgar F. Codd por primera vez en 1970. Este modelo utiliza conceptos de la matemática como la teoría de conjuntos y la lógica de predicados, y debe su nombre al concepto matemático de relación, representado como n-tuplas de valores.

El modelo relacional organiza la información en tablas, vistas como conjuntos de n-tuplas, en la que cada tupla representa una fila de la tabla, y cada valor de la n-tupla, denominado atributo, refiere a las columnas de la tabla.

A partir de su aparición, y hasta la actualidad, el modelo relacional se convirtió en el modelo dominante de bases de datos [1, 26].

Los sistemas de gestión de bases de datos relacionales hacen uso de transacciones con el objetivo de preservar la integridad de los datos, cumpliendo así las propiedades ACID (Atomicidad, Consistencia, Aislamiento (en inglés: *Isolation*) y Durabilidad) [8].

En esta sección se detallan todos los RDBMSs para dispositivos móviles encontrados en el relevamiento realizado, ordenados según su popularidad en el ranking elaborado por DB-Engines, presentando previamente el listado de dichos sistemas junto con su posición en el ranking específico de RDBMSs [27]:

- #6. SQLite
- #34. Interbase
- #35. SAP SQL Anywhere
- #71. SQLBase

SQLite

SQLite es una librería que implementa un motor de base de datos autocontenido (embebido). Es el motor de base de datos con más despliegues (*deployments*) en el mundo, ya que se encuentra en cada dispositivo Android, cada dispositivo iOS, cada dispositivo Mac, cada dispositivo Windows 10, cada navegador Firefox, Chrome o Safari, entre otros [28].

SQLite está desarrollado en el lenguaje de programación C, tiene licencia Public Domain, y se encuentra posicionada en el ranking general de DB-Engines en el puesto 9 [3], y en el ranking específico de bases de datos relacionales en el puesto 6 [27]. La primera versión fue publicada en el año 2000, y la última versión se publicó en enero de 2021 [29].

Respecto a su uso en el desarrollo de aplicaciones móviles, SQLite puede utilizarse tanto en el desarrollo nativo de aplicaciones, ya sea en Android [30, 31, 32] o en iOS [33, 34, 35] (ya que, como se mencionó anteriormente, SQLite está instalado en ambos sistemas), como también en el desarrollo de aplicaciones multiplataforma en distintos enfoques, como el enfoque híbrido [36], el interpretado [37, 38] o el de compilación cruzada [39, 40, 41].

Interbase

Interbase es un DBMS relacional embebido, conforme al estándar SQL, con énfasis en la seguridad, ya que provee cifrado a nivel de tabla con AES 256-bit y a nivel de columna con DES, encriptación de respaldos de seguridad, generación de respaldos durante su uso y la posibilidad de realizar restauraciones rápidas [42].

Interbase está desarrollada en el lenguaje de programación C, tiene licencia comercial, y se encuentra en el puesto 59 del ranking general de DB-Engines [3], y en el puesto 34 del ranking de DBMSs relacionales [27].

En cuanto al desarrollo de aplicaciones móviles, Interbase se puede ins-

talar en Android y en iOS para el desarrollo nativo [43], sin embargo no se encuentra disponible para ninguno de los principales frameworks de desarrollo de aplicaciones multiplataforma.

SAP SQL Anywhere

SAP SQL Anywhere es un paquete de DBMSs relacionales y tecnologías de sincronización para servidores, entornos de escritorio y móviles. Se trata de un DBMS con soporte para datos espaciales (*spatial data*) [44].

Es un producto de software con licencia comercial. La primera versión fue publicada en 1992 y la más reciente en julio de 2015 [45]. Se encuentra en el puesto 61 del ranking general de DB-Engines [3], y en el puesto 35 de bases de datos relacionales [27].

Se encuentra disponible para utilizar en el desarrollo de aplicaciones móviles nativas, tanto en Android como en iOS.

SQLBase

SQLBase es un RDBMS desarrollado por OpenText. La primera versión fue publicada en 1985. Tiene licencia de uso comercial. Se encuentra disponible para el desarrollo de aplicaciones móviles nativas, en Android y iOS. En los rankings general y relacional de DB-Engines, se encuentra en los puestos 141 y 71 respectivamente [46, 47].

3.1.1. Selección de DBMS Relacional

Se selecciona para el análisis que se realiza en este trabajo de tesina un DBMS relacional. Para la selección se consideran:

- que el DBMS pueda ser utilizado sin una licencia comercial;

- la popularidad que posee cada DBMS, según el ranking elaborado por DB-Engines [3];
- la posibilidad de embeberlos en aplicaciones móviles desarrolladas de forma nativa en las principales plataformas (Android e iOS);
- la posibilidad de embeberlos en aplicaciones móviles desarrolladas con diferentes enfoques multiplatafoma en los principales frameworks de desarrollo (React Native, NativeScript, Ionic Framework, Xamarin y Flutter).

El DBMS relacional seleccionado es SQLite, ya que se trata del más utilizado y difundido, y está presente en las dos plataformas y los principales frameworks de desarrollo multiplataforma.

3.2. DBMSs NoSQL

A partir de la década de 2010, con la aparición y popularización de los *smartphones* y las redes sociales, se está produciendo un crecimiento exponencial en el volúmen de datos administrados, comprendiendo datos estructurados, semi-estructurados, y sin estructura. Ante esta situación de crecimiento en el volúmen de datos, surgen las bases de datos NoSQL, no como reemplazo de las bases de datos relacionales, sino como alternativa a éstas, ya que facilitan el almacenamiento masivo de datos semi-estructurados o no estructurados.

Las bases de datos NoSQL no corresponden a un tipo de base de datos, sino que el término NoSQL se utiliza para englobar a los diferentes tipos de bases de datos que no se basan en el modelo relacional.

En relación a las propiedades ACID, las bases de datos NoSQL no aseguran que éstas se cumplan. Proponen en reemplazo a este conjunto de propiedades, las denominadas propiedades BASE:

- Básicamente Disponible (*Basically Available*):

  implica que el sistema se encontrará disponible la mayoría del tiempo;

- Estado suave/débil (*Soft state*):

  el sistema presenta flexibilidad en cuanto a la consistencia de los datos almacenados;

- consistencia Eventual (*Eventual consistency*):

  el sistema garantiza que eventualmente se alcance un estado consistente [8].

3.2.1. Documentales

Las bases de datos documentales (*document store*) representan al tipo de DBMS NoSQL más utilizado en la actualidad [3]. El concepto central de este tipo de almacenamiento de datos es el documento. Los DBMSs documentales almacenan, recuperan y gestionan datos de documentos, bajo algún formato estándar, como pueden ser XML, JSON, BSON, YAML, entre otros [8].

Se presenta en esta sección el detalle de todos los DBMSs documentales para dispositivos móviles encontrados, listando previamente el ranking correspondiente a este tipo de DBMS elaborado por DB-Engines, en el que se muestra el puesto en este ranking junto al nombre del sistema [48]:

#4. Couchbase Lite

#5. Firebase Realtime Database

#8. Realm

#9. Google Cloud Firestore

#16. PouchDB

#23. LiteDB

Couchbase Lite

Couchbase Lite es una base de datos NoSQL documental embebida. Utiliza JSON como formato de los documentos. Al formar parte del paquete de programas de Couchbase, es posible utilizar Sync Gateway para sincronizar los datos almacenados en el dispositivo en la base de datos Couchbase Lite con una base de datos Couchbase Server alojada en un servidor central. También es posible sincronizar datos entre bases de datos Couchbase Lite sin la necesidad de un servidor central utilizando sincronización peer-to-peer [49]. Además, al tratarse de un proyecto derivado de Apache CouchDB, utiliza el mismo protocolo de sincronización que ésta, por lo que también es posible sincronizar los datos con una base de datos CouchDB central. Asimismo, es posible realizar sincronización entre pares con otras bases de datos Couchbase Lite, o con bases de datos PouchDB, un DBMS desarrollado en javascript inspirado en CouchDB [50, 51].

Couchbase Lite está desarrollada en los lenguajes de programación C, C++, Go y Erlang, tiene licencia Apache 2.0, aunque también cuenta con licencias comerciales. Su primera versión fue publicada en 2011, y su versión más reciente en enero de 2020. Ocupa el puesto 26 del ranking general de DB-Engines, y el puesto 4 en el ranking específico de DBMS NoSQL documentales [52, 48].

Las consultas al DBMS se realizan utilizando un lenguaje basado en SQL denominado N1QL (pronunciado *nickel*) que adapta SQL a las posibilidades que ofrecen los documentos JSON, como campos de tipo array y documentos embebidos [53].

Es posible utilizar Couchbase Lite en el desarrollo de aplicaciones móviles de desarrollo nativo, tanto para Android como para iOS, como así también para distintos enfoques multiplataforma, como híbrido, interpretado, o de compilación cruzada [54, 55, 56].

Firebase Realtime Database

Firebase Realtime Database es una base de datos NoSQL documental alojada en la nube. Los datos se almacenan en un único documento JSON y se sincronizan en tiempo real con todos los clientes conectados, manteniéndose disponibles aún sin conexión [57].

Firebase Realtime Database tiene licencia comercial, sin embargo es posible utilizarla de forma gratuita con límites de 100 conexiones simultáneas, una única base de datos, 10 GB de almacenamiento y 10 GB/mes descargados [58]. En el ranking general de DB-Engines se encuentra en el puesto 37 [3], y en el ranking de DBMSs documentales, en el puesto 5 [48].

Las consultas al DBMS se realizan a través de una API que permite obtener datos que se actualicen a medida que se modifiquen (por el mismo cliente o por otros clientes) u obtenerlos una única vez; es decir, que en caso de que se modifiquen, para obtener los datos actualizados, debería realizarse una nueva consulta. El SDK de Firebase Realtime Database se encuentra disponible para instalar en Android, iOS (lo que permite su uso en el desarrollo nativo de aplicaciones móviles) y Javascript, existiendo librerías para utilizar el DBMS en el desarrollo de aplicaciones móviles multiplataforma en enfoques híbridos [59], interpretados [60, 61] y de compilación cruzada [62, 63].

Realm

Realm es una base de datos NoSQL documental embebida, que utiliza un modelo de datos orientado a objetos. Es posible utilizarla de forma autónoma, o de forma sincronizada con una base de datos MongoDB central, ofreciendo la posibilidad de trabajar con "objetos vivos" (*live objects*), que se actualizan de forma automática ante cualquier cambio realizado en el servidor o en cualquier cliente [64].

Está desarrollada en C++ [65]. La primera versión fue liberada en 2014

bajo licencia Apache 2.0 por la empresa Realm, que fue adquirida por MongoDB en 2019 [66].

En el ranking general del ranking DB-Engines se encuentra en el puesto 45, y en el puesto 8 del ranking de DBMSs NoSQL documental [66].

En cuanto a su uso en el desarrollo de aplicaciones móviles, el DBMS cuenta con SDKs para Android, iOS, Xamarin, y TypeScript y Javascript [67]. Esto implica que es posible utilizarlo en el desarrollo de aplicaciones de forma nativa, y en los enfoques multiplataforma interpretado y de compilación cruzada; respecto al enfoque híbrido, se encuentran en desarrollo librerías para Cordova [68] y Capacitor [69].

Google Cloud Firestore

Integrado al ecosistema Firebase de Google Cloud, Google Cloud Firestore en un DBMS NoSQL documental alojado en la nube que, al igual que Firebase Realtime Database, permite acceder sin conexión a datos que fueron accedidos previamente, y mantiene a los datos de los clientes y el servidor sincronizados de forma automática. A diferencia de Firebase Realtime Database, DBMS en el que una base de datos consiste en un único documento JSON, Google Cloud Firestore permite crear múltiples documentos JSON y organizarlos en distintas colecciones. Además presenta mejoras en cuanto a la escabilidad, la velocidad, y el manejo de estructuras y consultas más complejas [70, 71].

La primera versión del DBMS fue liberada en 2017. Al igual que Firebase Realtime Database, Google Cloud Firestore cuenta con una licencia comercial y un plan de uso gratuito. El mismo incluye la posibilidad de almacenar 1 GiB de datos, realizar 50000 operaciones de lectura, 20000 de modificación y 20000 de eliminación por día, y documentos de hasta 1 MiB, entre otras restricciones [72, 73].

En el ranking DB-Engines general se encuentra en el lugar 51 [3], y ocupa

el lugar 9 del ranking de DBMSs documentales [48].

Cuenta con SDKs para Android y iOS, por ende es posible utilizar el DBMS para el desarrollo de aplicaciones móviles nativas [70], y existen librerías para utilizarlo en el desarrollo de aplicaciones móviles multiplataforma con enfoque híbrido [74], interpretado [61, 75] y de compilación cruzada [76, 77].

PouchDB

PouchDB es un DBMS NoSQL documental desarrollado en JavaScript inspirado en CouchDB. Utiliza documentos JSON para almacenar la información. Tiene como objetivo emular la API y utilizar el protocolo de sincronización de CouchDB, motivo por el cual es posible utilizar PouchDB en sincronización con una base de datos CouchDB, Couchbase o IBM Cloudant. Utiliza una API JavaScript y una API HTTP RESTful para manipular datos [51, 78].

Al estar desarrollada en JavaScript, es posible utilizar PouchDB en el navegador o en un entorno Node.js [78]. Es desarrollado por la Apache Software Foundation, su primera versión fue publicada en 2012, y la más reciente en junio de 2019. Es de código abierto, tiene licencia Apache 2.0 [79]. Se encuentra en el puesto 88 del ranking general de DB-Engines [3], y en el puesto 16 de DBMSs documentales [48].

No es posible utilizar PouchDB en el desarrollo nativo de aplicaciones. Sin embargo, es posible utilizarlo en el desarrollo multiplataforma con enfoque híbrido [80] e interpretado [81].

LiteDB

LiteDB es un DBMS NoSQL documental embebido similar a MongoDB. Utiliza documentos BSON para almacenar la información. Está desarrollado en .NET C# y distribuido como un único archivo DLL. Utiliza una sintaxis

similar a SQL para la manipulación de datos, agregando soporte para expresiones LINQ (funciones lambda). Es de código abierto, con licencia MIT [82].

En los rankings general y NoSQL documental elaborados por DB-Engine, LiteDB se encuentra en los puestos 144 y 23 respectivamente [3, 48].

En relación a la utilización del DBMS en el desarrollo de aplicaciones móviles, LiteDB sólo se encuentra disponible en Xamarin (enfoque de compilación cruzada para el desarrollo de aplicaciones multiplataforma) [82].

3.2.2. Otros tipos

Se presentan finalmente DBMSs NoSQL correspondientes a tipos distintos al documental.

En primer lugar, se encuentra un DBMS de clave-valor. Este tipo de DBMS almacena la información como conjunto de pares clave-valor, en los que la clave representa un identificador que referencia a un objeto complejo y arbitrario de información, denominado valor [8].

En segundo lugar, se encuentra un DBMS orientado a objetos. Este tipo de DBMS sigue el modelo de datos de la programación orientada a objetos, por lo que se encargan de almacenar, recuperar y gestionar objetos [83].

Por último, se encuentra un DBMS de grafos. En este tipo de DBMS se representa a la base de datos bajo el concepto de un grafo. Permite almacenar la información como nodos de un grafo, y como aristas las relaciones entre nodos, y aplican la teoría de grafos para obtener información [8].

Oracle Berkeley DB

Berkeley DB es una familia de bases de datos NoSQL de clave-valor embebidas. Los DBMS que integran Berkeley DB utilizan APIs de llamadas

a función para manipular los datos; son tres productos:

1. Berkeley DB: librería desarrollada en C, que cuenta con una API clave-valor y una API SQL incorporando SQLite para manipular los datos.
2. Berkeley DB Java Edition: librería (JAR) desarrollada en Java, cuenta con APIs clave-valor, Java Direct Persistence Layer, y Java Collections.
3. Berkeley DB XML: librería desarrollada en C++ que soporta el almacenamiento de documentos XML [84].

Berkeley DB tiene licencia open source, su primera versión fue publicada en 1994, y la última versión en junio de 2018. En el ranking general de DBEngines ocupa el puesto 87, y en el ranking específico de DBMSs clave-valor, el puesto 14 [85].

Berkeley DB se encuentra disponible para su uso en el desarrollo de aplicaciones móviles nativas en Android y en iOS [85].

ObjectBox

ObjectBox es un DBMS NoSQL orientado a objetos, de alta velocidad, embebido, para dispositivos móviles e IoT. Está desarrollado en C y C++, tiene licencia Apache 2.0. En el ranking DB-Engines general se encuentra en el puesto 195, mientras que en el ranking DB-Engines específico de DBMSs orientados a objetos se encuentra en el puesto 9 [86].

Cuenta con un servicio de almacenamiento en la nube y un servicio que permite sincronizar los datos en la nube con los almacenados en el dispositivo.

ObjectBox se encuentra disponible para ser utilizado en el desarrollo de aplicaciones móviles nativas en Android y en iOS. También se encuentra disponible de ser utilizado en el desarrollo de aplicaciones móviles multiplataforma en Flutter [87].

Sparksee

Sparksee es un DBMS NoSQL de grafos. Está desarrollado en C++ y tiene licencia comercial, pero cuenta con licencias gratuitas para desarrollo e investigación [88]. Su primera versión fue publicada en 2008, y la última versión en 2015 [89].

Se encuentra disponible para su uso en el desarrollo de aplicaciones móviles nativas en Android y en iOS [88].

En el ranking general de DB-Engines, se encuentra en el puesto 289. En el ranking específico de DBMSs de grafos ocupa el puesto 25 [89].

3.2.3. Selección de DBMS NoSQL

Se selecciona para el análisis que se realiza en este trabajo de tesina un DBMS NoSQL. Para la selección se consideran:

- que el DBMS pueda ser utilizado sin una licencia comercial;
- la popularidad que posee cada DBMS, según el ranking elaborado por DB-Engines [3];
- la posibilidad de embeberlos en aplicaciones móviles desarrolladas de forma nativa en las principales plataformas (Android e iOS);
- la posibilidad de embeberlos en aplicaciones móviles desarrolladas con diferentes enfoques multiplatafoma en los principales frameworks de desarrollo (React Native, NativeScript, Ionic Framework, Xamarin y Flutter).

El DBMS NoSQL seleccionado es Couchbase Lite, ya que es del tipo de base de datos NoSQL más utilizado (documental) según el ranking DB-Engines [3], y puede ser utilizado tanto en el desarrollo nativo en Android y en iOS, como en el desarrollo multiplataforma en los frameworks mencionados.

Capítulo 4

Experimentación con los DBMSs seleccionados

En este capítulo se presenta un análisis del uso en el desarrollo de aplicaciones para dispositivos móviles de un DBMS relacional, SQLite, y un DBMS NoSQL, Couchbase Lite.

Para llevar a cabo el análisis se implementan parcialmente dos aplicaciones móviles a modo de prueba de concepto (PoC) en la plataforma Android, cada una utilizando los DBMSs seleccionados. La aplicación consiste en una agenda de contactos que cumple con los siguientes requerimientos:

Requerimientos funcionales

1. La aplicación debe permitir crear un nuevo contacto, con los siguientes datos:
   - Apellido (requerido)
   - Nombre (requerido)
   - Fecha de nacimiento (opcional)
   - Apodo (opcional)

- Empresa (opcional)
- Teléfonos (cero o más) Para cada Teléfono se almacena:
  - Número (requerido)
  - Tipo (requerido, se debe seleccionar entre las siguientes opciones: Móvil, Casa, Trabajo, Otro)

2. La aplicación debe listar los contactos almacenados ordenados por apellido y nombre de forma ascendente.

3. La aplicación debe permitir filtrar los contactos almacenados, a partir de un único campo de búsqueda, listando los contactos que coincidan parcial o totalmente con el término de búsqueda en algunos de sus campos. Los contactos filtrados se mostrarán ordenados por Apellido y Nombre ascendentemente.

Requerimientos no funcionales

4. Toda la información debe almacenarse de forma local en la base de datos.

Para el desarrollo de la aplicación se define el diagrama correspondiente al modelo conceptual de base de datos, utilizando el Modelo Entidad-Relación, representado en la Figura 4.1, que luego se deriva a los modelos lógicos/físicos correspondientes según cada tipo de base de datos y DBMS.

El análisis comparativo se realiza desde el punto de vista de la experiencia del ingeniero de software en el desarrollo de cada aplicación, considerando facilidad de instalación/configuración, utilización de estructuras de datos, facilidad en la realización de consultas, entre otras.

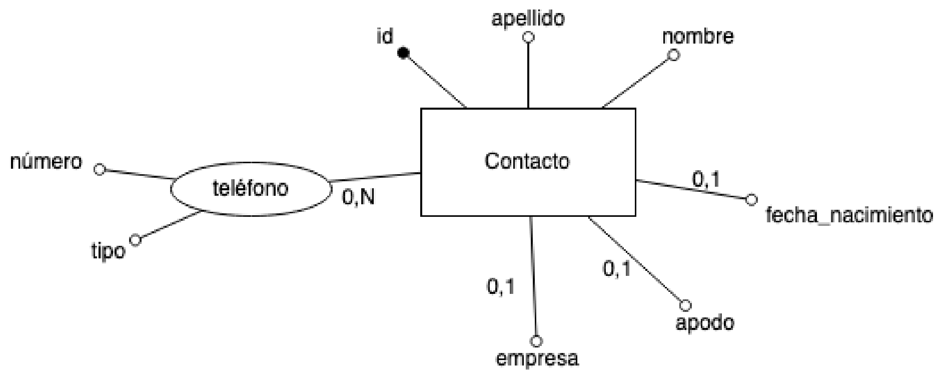

Figura 4.1: Diagrama conceptual de la PoC

4.1. SQLite

Como primer paso de la implementación de la aplicación utilizando SQLite como DBMS, se deriva el diagrama correspondiente al modelo conceptual (Figura 4.1) a los modelos lógico (Figura 4.2) y físico (Figura 4.3).

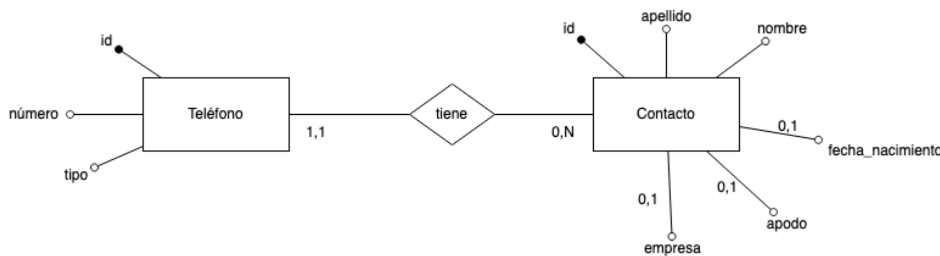

Figura 4.2: Diagrama correspondiente al modelo lógico para RDBMS

Contacto (id, apellido, nombre, fecha_nacimiento?, apodo?, empresa?)
Telefono (id, numero, tipo, contacto_id (FK))

Figura 4.3: Diagrama correspondiente al modelo físico para SQLite

Para la utilización de SQLite en el desarrollo de aplicaciones móviles, Android provee dos formas de gestionar bases de datos en dicho DBMS. La primera de ellas (recomendada por la documentación oficial) es utilizando

Room, una biblioteca de persistencia que funciona a modo de capa de abstracción de SQLite [31]. La segunda forma es utilizando la API de SQLite directamente.

Independientemente de si se utiliza Room o SQLite directamente, no es necesario que la aplicación obtenga permisos adicionales. Los datos almacenados en la base de datos no pueden ser accedidos por otras aplicaciones, y se borrarán al eliminarse la aplicación del dispositivo [30]. Además, los datos almacenados pueden eliminarse en su totalidad desde la configuración de la aplicación.

Instalación/Configuración

Si bien SQLite es una librería que forma parte del sistema operativo Android, y por lo tanto se encuentra disponible en la plataforma, es necesario agregar dependencias en el archivo *build.gradle* del proyecto de Android Studio e instalarlas. Para ello, deben agregarse las siguientes líneas en el bloque *dependencies*:

```
def room_version = "2.2.6"
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"
```

La primera línea define un valor que se utiliza en las dos líneas restantes, aunque bien podría eliminarse esa línea y copiar en las últimas el número de versión.

Definición del modelo

El uso de Room se lleva a cabo definiendo clases e interfaces de objetos a los que se les deben definir determinadas anotaciones.

La biblioteca cuenta con tres componentes principales: *entidades*, que son clases que representan a las entidades del modelo, y que representan las

tablas del modelo relacional. Estas clases deben anotarse con *@Entity*.

El segundo componente principal de Room son los *DAOs*, definidos a partir de interfaces con la anotación *@Dao*, que son utilizados para realizar consultas sobre la tabla que representa cada entidad, permitiendo obtener entidades, modificarlas, crearlas y eliminarlas.

El tercer componente es la *base de datos*, que sirve como punto de acceso principal para la conexión subyacente a los datos persistentes y relacionales de la aplicación. Se debe definir una clase abstracta que extienda de la clase *RoomDatabase* y que esté anotada con *@Database*. La anotación deberá contener el listado de Entidades que se definan para la aplicación. Esta clase también debe implementar, por cada DAO de la aplicación, un método público abstracto cuyo valor de retorno sea el de la interfaz del DAO.

Para el desarrollo de la prueba de concepto se definen las entidades *Contacto* y *Telefono*. La clase *Contacto* se define con las propiedades públicas[1] id (clave primaria), apellido, nombre, fechaNacimiento, apodo, empresa; todas las propiedades son de tipo *String*, a excepción de id (*long*) y fechaNacimiento (*Date*). La clase *Telefono* tiene las propiedades públicas[2] id (clave primaria, de tipo *long*), numero (de tipo *String*), tipo (cuyo tipo se define como TipoTelefono, un enumerativo que contiene los valores *MOVIL*, *CASA*, *TRABAJO* y *OTRO*) y contactoId (de tipo *long*), cuyo valor se corresponderá con la propiedad id de un objeto de la clase *Contacto*.

Para implementar la relación entre las entidades *Contacto* y *Telefono* se define una clase *ContactoConTelefonos*, debido a que Room no permite referencias entre objetos correspondientes a clases de entidades, ya que las mismas degradan el rendimiento de la aplicación y requieren una cantidad mayor de memoria [90]. Esta clase se define con dos propiedades públicas: *contacto*, de tipo *Contacto* y anotada con *@Embedded*; *telefonos*, de tipo *List* *<Telefono>* y anotada con *@Relation*. Esta última anotación debe definir

[1] Con el único objetivo de reducir la cantidad de código, no por restricción del DBMS o de la librería

[2] Ídem nota al pie 1

las propiedades *parentColumn*, cuyo valor refiere a la clave primaria de la entidad cuya cardinalidad máxima en la relación es N (en este caso, el valor debe ser “id”, referenciando a la propiedad *id* de la clase *Contacto*), y *entityColumn*, cuyo valor refiere a la clave primaria de la entidad cuya cardinalidad máxima en la relación es 1 (en este caso, el valor debe ser “contactoId”, referenciando a la propiedad *contactoId* de la clase *Telefono*) [91].

Debido a la imposibilidad de referenciar objetos en Room, las propiedades *fechaNacimiento* de la entidad *Contacto*, y *tipoTelefono* de la entidad *Telefono* deben ser convertidas para poder ser almacenadas en la base de datos. Para ello debe definirse una o más clases con métodos anotados con *@TypeConverter* de forma tal que un método reciba un parámetro del tipo correspondiente al definido en la entidad y retorne el tipo con el que se almacenará, y otro método reciba un parámetro del tipo correspondiente al tipo almacenado y retorne el tipo correspondiente a la entidad.

La o las clases que definan métodos para realizar conversiones deben listarse en la anotación de clase *@TypeConverters* que debe agregarse a la clase correspondiente a la base de datos.

Creación de contactos

Para satisfacer el requerimiento funcional #1 se genera una *Activity* que determina de forma aleatoria cuáles son los campos opcionales que se completan, y selecciona, también de forma aleatoria, el contenido de los campos, a partir de un conjunto de datos arbitrario, simulando de esta manera la carga de datos realizada por el usuario de la aplicación a partir de un formulario.

Una vez obtenidos los datos, se requiere insertar en la base de datos una tupla correspondiente al contacto que se desea crear, junto con las tuplas correspondientes a los teléfonos (cuya cantidad también se define de forma aleatoria, pudiendo ser ésta entre cero y cuatro).

La inserción de tuplas se realiza definiendo métodos con la anotación *@Insert* en la interfaz DAO correspondiente a la entidad.

Los métodos se pueden definir de forma tal que reciban como parámetro una único objeto (instancia de la clase) de la entidad o un array de objetos de la entidad (o múltiples objetos, ya que es posible definir el método con la notación de cantidad variable de argumentos —por ejemplo, *Contacto ... contactos*)).

Es posible definir el valor de retorno de estos métodos como *long* o array de *long* respectivamente, que serán los valores de las claves primarias de las tuplas insertadas. Es por este motivo que debe realizarse primero la inserción de la tupla de la entidad *Contacto*, y obtener el valor de la clave primaria para luego asignarlo a los objetos de las entidades *Telefono* como valor de la propiedad *contactoId*.

Este flujo de trabajo debe definirse de esta manera cuando se decide que la clave primaria de la entidad sea un número autoincremental gestionado por el DBMS, ya que también es posible definir alguna/s propiedad/es con semántica definida en el modelo como clave primaria (por ejemplo, podría definirse como clave primaria de *Contacto* las propiedades *apellido* y *nombre* —clave compuesta—, y como clave primaria —también compuesta— de *Telefono*, las propiedades *numero* y *contactoId*).

Listado de contactos

La aplicación debe listar todos los contactos almacenados para satisfacer el requerimiento funcional #2. Para ello, se genera una *Actity* que realice la invocación a un método definido en el DAO de *Contacto* y muestre los resultados obtenidos.

La definición de dicho método en la interfaz correspondiente al DAO debe anotarse con *@Query*, y debe definirse como valor de esta anotación la consulta a ejecutar:

```sql
SELECT *
FROM contacto
ORDER BY apellido, nombre
```

Filtrado de contactos

La funcionalidad correspondiente al filtrado de contactos, que debe satisfacer el requerimiento funcional #3, es similar a la funcionalidad correspondiente al listado de contactos, ya que también consiste en realizar una consulta a la base de datos para obtener contactos y mostrar los resultados obtenidos.

En este caso, a diferencia del requerimiento #2, no deben mostrarse todos los contactos, sino sólo aquellos que coincidan parcial o totalmente con un término de búsqueda ingresado por el usuario.

De forma análoga a la implementación del requerimiento #1, en lugar de presentar un formulario al usuario de la aplicación para que ingrese un término de búsqueda, se simula la búsqueda seleccionando aletaoriamente un término a partir de un conjunto arbitrario de cadenas de caracteres.

La ejecución de la consulta debe realizarse definiendo un método en el DAO de la entidad *Contacto*, pero a diferencia del método correspondiente al listado de contactos, éste debe definirse con un parámetro de tipo *String* para recibir el término de búsqueda.

La definición de la consulta a ejecutar se realiza de la misma forma que en el listado, determinándola como valor de la anotación *@Query* del método de la interfaz, utilizando una notación especial para referenciar el valor del parámetro ingresado en la consulta. La notación consiste en anteponer dos puntos (:) al nombre del parámetro según se haya definido en el método. Es decir que la consulta, considerando que el nombre del parámetro del método se establece como "termino", queda de la siguiente forma:

```sql
SELECT DISTINCT contacto.*
```

```sql
FROM contacto
  LEFT JOIN telefono ON telefono.contactoId = contacto.id
WHERE apellido LIKE '%' || :termino || '%'
  OR nombre LIKE '%' || :termino || '%'
  OR fechaNacimiento LIKE '%' || :termino || '%'
  OR apodo LIKE '%' || :termino || '%'
  OR empresa LIKE '%' || :termino || '%'
  OR numero LIKE '%' || :termino || '%'
ORDER BY apellido, nombre
```

Nótese que para realizar búsquedas de coincidencia parcial en cadenas de caracteres con el operador *LIKE*, debe concatenarse (con el operador "||") al parámetro (denotado con *:termino*) los caracteres "%".

Código fuente

El código fuente de la prueba de concepto se encuentra en el siguiente repositorio de GitHub, en la versión etiquetada como *pre-release* v0.1: `https://github.com/ftesone/tesina-room/tree/v0.1`

4.2. Couchbase Lite

La primera tarea a resolver para implementar la aplicación utilizando Couchbase Lite como DBMS es generar el diagrama físico de base de datos a partir del diagrama conceptual (Figura 4.1). Para ello se utilizará la notación *crow's foot*, que permite modelar características propias de las bases de datos NoSQL documentales, como los arrays y los documentos embebidos. En la Figura 4.4 se presenta el diagrama físico.

Una característica particular de las bases de datos NoSQL es que ofrecen la posibilidad de almacenar datos sin estructura o semiestructurados. Asimismo, es posible definir parcial o totalmente un esquema para los do-

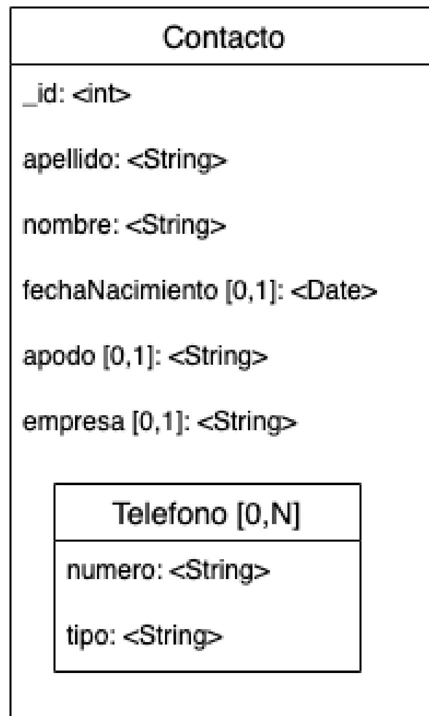

Figura 4.4: Diagrama correspondiente al modelo físico para Couchbase Lite

cumentos, aunque no es algo requerido.

En lo que respecta a la prueba de concepto, no se definirá un esquema, por lo que el diagrama físico definido en la Figura 4.4 sólo sirve de referencia sobre cómo estructurar los documentos (es decir, los documentos se estructurarán siguiendo el modelo definido aunque no exista un esquema que valide dicha estructura).

Respecto a los datos que se almacenan en la/s base/s de datos Couchbase Lite, cabe mencionar que éstos son eliminados si la aplicación se elimina del sistema. Además, es posible eliminar los datos por completo desde la configuración de la aplicación.

No es necesario que la aplicación obtenga permisos adicionales para utilizar este almacenamiento de datos. Los datos almacenados no pueden ser

accedidos por otras aplicaciones.

Instalación/Configuración

La instalación de Couchbase Lite se realiza agregando las dependencias al archivo *build.gradle* del proyecto, y posteriormente actualizándolas. Deben agregarse en el bloque *android* las siguientes líneas:

```
compileOptions {
    targetCompatibility 1.8
    sourceCompatibility 1.8
}
```

Estas líneas aseguran una versión mínima de la máquina virtual de java, ya que Couchbase Lite hace uso de características presentes en versiones recientes del lenguaje, como las expresiones lambda.

Se requiere también que se agregue la siguiente línea en el bloque *dependencies*:

```
implementation "com.couchbase.lite:couchbase-lite-android:2.8.1"
```

Definición del modelo

Si bien no se define un esquema de documentos en Couchbase Lite, se pueden definir clases de objetos para representar las entidades (entendiendo el término "entidades" como clase para representar al modelo datos de la aplicación, sin relación con la persistencia de datos).

Se definen entonces para la aplicación dos clases: *Contacto*, con propiedades públicas[3] *apellido*, *nombre*, *apodo*, *fechaNacimiento*, *empresa* y *telefonos*, todas de tipo *String* a excepción de *fechaNacimiento* (de tipo *Date*) y *telefonos* (de tipo *List<Telefono>*); *Telefono*, con propiedades *numero* de

[3] Con el único objetivo de reducir la cantidad de código

tipo *String* y *tipo* de tipo *TipoTelefono* (un tipo enumerativo con los valores *MOVIL*, *CASA*, *TRABAJO* y *OTRO*).

Estas clases, que no tendrán utilidad en la transferencia de datos desde/hacia la base de datos, se utilizarán para mostrar la información al usuario.

Creación de contactos

De forma análoga a la prueba de concepto correspondiente a la utilización de SQLite con Room, los datos utilizados para crear contactos son generados de forma aleatoria a partir de un conjunto arbitrario de valores (se utilizan los mismos datos y la misma forma de seleccionarlos que en la aplicación que utiliza Room), en lugar de proveer un formulario al usuario de la aplicación.

La funcionalidad encargada de satisfacer el requerimiento funcional #1 se define por completo en una *Activity*. En ésta se generan los datos y se crean dos objetos: el primero, una instancia de la clase *Contacto*, que se utiliza para mostrar los datos del contacto a crear al usuario; el segundo, una instancia de *MutableDocument*, una clase que provee Couchbase Lite para representar a documentos que se pueden modificar (en contraposición con *Document*, que representa documentos inmutables).

Los datos seleccionados de forma aleatoria se asignan a las propiedades del objeto *Contacto* y se agregan al objeto *MutableDocument* utilizando los métodos *setString(clave, valor)* (para las propiedades *apellido*, *nombre*, *apodo* y *empresa*), *setDate(clave, valor)* (para la propiedad *fechaNacimiento*).

La forma de asignar los valores correspondientes a teléfonos difiere del resto de los campos de *Contacto*, ya que, siguiendo el diagrama físico definido, se deberían representar como un array de documentos embebidos en *Contacto*. Para ello se debe crear una instancia de la clase *MutableArray* proveída por el SDK de Couchbase Lite, a la cual se agregan instancias de *MutableDictionary* (también definida por el SDK) a las cuales se les define

los campos y valores de la misma forma que un *MutableDocument*.

Una vez definidos los valores del objeto *MutableArray*, se asigna al *MutableDocument* que contiene los datos de contacto con el método *setArray(clave, valor)*.

Finalmente, debe obtenerse una instancia de la base de datos (un objeto de la clase *Database*) y se invoca al método *save*, con la instancia de *MutableDocument* como parámetro para persistir el documento.

Listado de contactos

La implementación del requerimiento funcional #2 se lleva a cabo definiendo una *Activity* en la que se obtiene una instancia de la base de datos y se realiza una consulta para obtener todos los contactos.

En Couchbase Lite en Android no es posible utilizar N1QL para realizar la consulta; debe utilizarse un *QueryBuilder*, una clase que provee el SDK del DBMS para construirse la consulta a partir de distintos mensajes.

En primer lugar debe definirse la proyección de las propiedades del documento, para lo cual es posible listar una a una las propiedades que se desean obtener, o utilizar un selector que referencia a todas las propiedades. La diferencia en el uso de una u otra forma, es que la primera obtiene el documento tal cual se ha almacenado, mientras que la segunda obtiene un documento con una única propiedad, cuyo valor es el documento tal cual se ha almacenado; esto hace que cambie la forma en que se obtienen los resultados de la consulta.

Luego de completar la consulta con el ordenamiento, ésta se ejecuta, lo que devuelve una lista de resultados (*List<Result>*), que debe iterarse para obtener los datos almacenados e hidratar un objeto *Contacto* asignando los valores del documento a las propiedades del objeto una por una.

Para hidratar la propiedad *telefonos* de *Contacto*, se debe realizar el

proceso inverso al realizado en la creación del documento: se debe obtener el *array* correspondiente al valor de la propiedad del documento, y luego obtener un diccionario que representa a cada teléfono, y que está compuesto por las propiedades *numero* y *tipo*.

Filtrado de contactos

La implementación de la funcionalidad necesaria para satisfacer el requerimiento funcional #3 es similar a la funcionalidad de listar todos los contactos. La diferencia radica en que deben agregarse las condiciones necesarias en la consulta para que sólo se retornen contactos que coincidan parcial o totalmente con el término de búsqueda en alguna de sus propiedades.

Al igual que en la aplicación que utiliza SQLite, el término de búsqueda se selecciona aleatoriamente a partir de un conjunto arbitrario de *strings* en lugar de proveer al usuario un formulario que le permita ingresar el término.

Para definir las condiciones de búsqueda en la consulta, el SDK de Couchbase Lite provee *expresiones* (clase *Expression*), que permiten referenciar a propiedades del documento y realizar comparaciones con otras expresiones de tipo *string*.

Las comparaciones entre *strings* es sensible a mayúsculas y minúsculas, por lo que si se desea que la búsqueda sea insensible a mayúsculas y minúsculas se deben pasar tanto el término de búsqueda como los valores de las propiedades del documento a minúsculas (o mayúsculas, ya que a fines prácticos es equivalente) previo a la comparación.

Para buscar coincidencias entre el término de búsqueda y números de telefono de cada contacto, deben utilizarse objetos de la clase *ArrayExpression*, que permiten aplicar condiciones de selección a documentos embebidos.

Código fuente

El código fuente de la prueba de concepto se encuentra en el siguiente repositorio de GitHub, en la versión etiquetada como *pre-release* v0.1: `https://github.com/ftesone/tesina-couchbase/tree/v0.1`

Capítulo 5

Análisis de Resultados

En este capítulo se presentan los resultados obtenidos a partir del análisis realizado en cada DBMS seleccionado. Se consideran ventajas y desventajas en la forma de trabajo llevada a cabo en el desarrollo de la prueba de concepto realizada. Se analizan variantes y/o mejoras en la implementación de la solución al problema presentado.

5.1. SQLite

En la prueba de concepto desarrollada utilizando SQLite como DBMS para persistir la información se decidió utilizar la biblioteca Room, que representa una capa de abstracción sobre SQLite. En la implementación realizada se destacan como ventajas:

1. la estructura de clases que la biblioteca requiere definir, ya que lleva a trabajar de una forma ordenada y sistemática en relación a la gestión de la persistencia de datos;
2. la definición del esquema de la base de datos a partir de la definición de las clases que conforman las entidades del modelo. Es decir, el hecho

de que no sea necesario definir las sentencias que generan el esquema de la base de datos;

3. la facilidad y flexibilidad que se provee para definir consultas para obtener información, ya que sólo se debe agregar una anotación a un método de una interfaz (es decir, no se requiere implementar dicho método) con la consulta como valor de la anotación; se suma a esto el hecho de que los parámetros que se deseen pasar a la consulta se definen como parámetros del método, y el nombre de los mismos se puede utilizar en la consulta sin necesidad de realizar otro tipo de implementaciones;

4. la facilidad con la que se definen conversiones de tipos que no se pueden almacenar directamente en la base de datos, ya que basta con definir una clase con una anotación, y con dos métodos para realizar las conversiones, y registrar la clase mediante otra anotación;

5. la facilidad con la que se insertan, modifican y eliminan tuplas, ya que sólo deben definirse métodos con las anotaciones *@Insert*, *@Update* y *@Delete* respectivamente.

Asimismo, se enumeran algunas desventajas en la utilización de Room:

1. la necesidad de utilizar estructuras auxiliares para expresar relaciones entre entidades;

2. la necesidad de definir un conversor para un tipo de dato ampliamente utilizado como es *Date*.

Por otra parte, con respecto a la utilización de SQLite como DBMS, se analizó la utilización de la API de SQLite en lugar de Room. A pesar de que no se implementó su uso en el desarrollo de este trabajo de tesina, se reconocen las siguientes desventajas:

1. se deben definir y ejecutar todas las sentencias relacionadas a la definición del esquema de la base de datos (creación de tablas, definición

de índices, etc.);

2. se deben implementar las consultas que modifican el contenido de tablas de la base de datos (alta, baja y modificación de tuplas);

3. se debe implementar la hidratación de objetos a partir de los datos recuperados de la base de datos si se desea utilizar el paradigma de programación orientada a objetos.

5.2. Couchbase Lite

En la utilización de Couchbase Lite como DBMS en la segunda prueba de concepto desarrollada, pueden destacarse como ventajas:

1. sólo se almacenan los datos que están definidos, ya que los posibles valores nulos no se almacenan;

2. no es necesario realizar cruces de datos, ya que las posibles relaciones de datos están dentro de un mismo documento.

Cabe mencionar que estas ventajas no se deben a características propias de Couchbase Lite, sino a características inherentes de las bases de datos NoSQL documentales.

Como desventajas del uso de Couchbase Lite, pueden mencionarse:

1. la utilización de estructuras de datos diferentes para interactuar con la base de datos y la interfaz gráfica;

2. la complejidad que se presenta al realizar consultas seleccionando datos, sobre todo al aplicar condiciones sobre documentos embebidos;

3. la falta de una organización clara de trabajo, que puede llevar a programadores inexperimentados a generar código difícil de comprender y mantener.

A raíz de los problemas mencionados, se analizó realizar un refactoring para reorganizar el código, intentando replicar la estructura de clases utilizada por Room. Para ello se definió la clase *ContactoDao*, implementando el patrón *singleton*, con métodos para insertar un contacto, obtener todos los contactos, y obtener todos los contactos que coincidan parcial o totalmente en alguna propiedad con un término de búsqueda recibido por parámetro. De esta forma se logró encapsular la utilización de objetos usados para representar documentos (*MutableDocument*, *MutableDictionary*, entre otros), así como también la hidratación de objetos de clase *Contacto* y *Telefono*. Esto llevó a que el código definido en las *activities* que manipulan datos trabaje sólo con las clases *Contacto* y *Telefono* y se reduzca considerablemente la cantidad de líneas de código. A partir de estos cambios, se generó una nueva versión etiquetadda como *pre-release* v0.2 en el repositorio: `https://github.com/ftesone/tesina-couchbase/tree/v0.2`

Capítulo 6

Conclusiones y Trabajo Futuro

Este trabajo intenta abordar la problemática que representa la elección de un DBMS adecuado para su utilización en el desarrollo de aplicaciones móviles, acorde al problema que se intenta resolver.

Se introduce el tema analizando el aumento exponencial en el volúmen de datos administrados, incluyendo datos estructurados, semi-estructurados, y no estructurados, que se produjo a partir del crecimiento en el alcance y el uso de internet, los dispositivos móviles, y las redes sociales, que llevaron al surgimiento de nuevos tipos de bases de datos, las bases de datos NoSQL, que facilitan el almacenamiento masivo de datos semi-estructurados y no estructurados.

Se plantea como objetivo realizar un relevamiento exhaustivo de DBMSs que se pueden instalar en dispositivos móviles, ya que las mejoras en el hardware de éstos llevan a que se utilicen para administrar cada vez más información, así como también un análisis detallado de todos los DBMSs encontrados, y una experimentación sobre el uso de DBMSs seleccionados que permita elaborar un análisis sobre características, ventajas y desventajas,

desde el punto de vista del ingeniero de software.

A continuación se presentan trabajos de investigación que se relacionan con el tema analizado de la presente tesina, en los que se analizan características de los sistemas de gestión de bases de datos para dispositivos móviles, la arquitectura del sistema operativo Android, la arquitectura de las aplicaciones para Android, las diferentes formas de persistencia de datos presentes en Android y iOS, y el uso de SQLite en dispositivos móviles.

Posteriormente, se presenta un detalle de todos los DBMSs para dispositivos móviles que se encontraron a raíz del relevamiento llevado a cabo. El detalle de los DBMSs se presenta organizado por tipo de DBMS, y ordenado acorde al ranking de popularidad que realiza el sitio DB-Engines; se detalla para cada DBMS información relacionada al desarrollo y distribución del producto de software, así como también presencia en las distintas plataformas de desarrollo de aplicaciones móviles nativo y multiplataforma. Finalmente se selecciona un DBMS relacional y uno NoSQL que resulten representativos de su tipo.

Luego, se lleva a cabo una experimentación utilizando los DBMSs seleccionados, en la que se presenta un conjunto de requerimientos funcionales y no funcionales de una prueba de concepto, y se desarrollan dos aplicaciones móviles nativas para Android, utilizando en una de las aplicaciones el DBMS relacional seleccionado —SQLite—, y en la otra, el DBMS NoSQL seleccionado —Couchbase Lite.

Finalmente, se analizan los resultados de la experimentación realizada, definiendo ventajas y desventajas en el uso de cada DBMS, y además se realiza un refactoring sobre una de las aplicaciones. Fruto del refactoring y de un nuevo análisis, se considera que la utilización de Room lleva a que se genere código más organizado que utilizando SQLite sin Room o Couchbase Lite; sin embargo, es posible estructurar el código de forma similar a la utilizada por Room con SQLite sin Room y con Couchbase Lite, pero requiere de un mayor esfuerzo por parte del programador, ya que, además de la definición de las clases de objetos, deben codificarse las consultas a

ejecutar y la hidratación de objetos.

Por otro lado, se considera que Couchbase Lite se adapta mejor a modelos de datos en los que las estructuras de éstos no están claramente definidas o muchas propiedades son opcionales; no obstante, esto se debe a que se trata de ventajas inherentes a un DBMS NoSQL documental y no a características exclusivas de Couchbase Lite.

Para finalizar, a partir de la experimentación realizada, es conveniente destacar que el modelo de datos del problema a resolver, es determinante para elegir el DBMS más adecuado entre los seleccionados (SQLite o Couchbase Lite) para desarrollo móvil, de acuerdo a las características oportunamente descriptas.

Se proponen varias líneas de investigación como posibles trabajos futuros:

- extender la experimentación realizada agregando el requerimiento no funcional de tener que sincronizar los datos almacenados en la base de datos del dispositivo móvil con una base de datos en un servidor central.

  Se establece como hipótesis, en base a las características relevadas en este trabajo, que Couchbase Lite requerirá un menor esfuerzo en la implementación si se utiliza como base de datos en el servidor Couchbase Server o CouchDB, ya que dentro del paquete de productos de Couchbase se encuentra Sync Gateway, una herramienta que permite automatizar la sincronización de datos entre una base de datos Couchbase Lite y (1) una base de datos Couchbase Server, (2) una base de datos CouchDB, (3) otra base de datos Couchbase Lite, o (4) una base de datos PouchDB, estas últimas dos utilizando sincronización entre pares, ya que se tratan de DBMSs que funcionan como clientes de un servidor central de base de datos;

- analizar el impacto que produce la utilización de SQLite o Couchbase Lite en requerimientos no funcionales que resultan determinantes

en el éxito de la aplicación, como lo son el espacio de almacenamiento utilizado, uso de memoria, rendimiento en tiempo de ejecución de consultas, y consumo de energía;

- extender el análisis realizado utilizando para la experimentación otros DBMSs, resultando interesante sobre todo DBMSs que se encuentren en ambas plataformas y en los principales frameworks de desarrollo multiplataforma de aplicaciones móviles.

Bibliografía

[1] K. L. Berg, T. Seymour, and R. Goel, "History of databases," *International Journal of Management & Information Systems (IJMIS)*, vol. 17, no. 1, pp. 29–36, 2013.

[2] B. Grad and T. J. Bergin, "Guest editors' introduction: History of database management systems," *IEEE Annals of the History of Computing*, vol. 31, no. 4, pp. 3–5, 2009.

[3] "Db-engines ranking - populariry ranking of database management systems." `https://db-engines.com/en/ranking`. Accedido por última vez: 17/02/2021.

[4] M. Campbell-Kelly and D. D. Garcia-Swartz, *From mainframes to smartphones: a history of the international computer industry*, vol. 1. Harvard University Press, 2015.

[5] "List of countries by smartphone penetration - wikipedia." `https://en.wikipedia.org/wiki/List_of_countries_by_smartphone_penetration#2013_rankings`. Accedido por última vez: 17/02/2021.

[6] "• cell phone sales worldwide 2007-2020 | statista." `https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/`. Accedido por última vez: 17/02/2021.

[7] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*, pp. 213–220, 2013.

[8] L. Marrero, V. Olsowy, P. J. Thomas, L. N. Delía, F. Tesone, J. Fernández Sosa, and P. M. Pesado, "Un estudio comparativo de bases de datos relacionales y bases de datos nosql," in *XXV Congreso Argentino de Ciencias de la Computación (CACIC)(Universidad Nacional de Río Cuarto, Córdoba, 14 al 18 de octubre de 2019)*, 2019.

[9] "One code, many apps - cross-platform mobile app development | toobler." `https://www.toobler.com/one-code-many-apps-cross-platform-mobile-app-development/`. Accedido por última vez: 04/03/2021.

[10] "· cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020 | statista." `https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/`. Accedido por última vez: 04/03/2021.

[11] A. Nori, "Mobile and embedded databases," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data*, pp. 1175–1177, 2007.

[12] J. Fernández Sosa, P. J. Thomas, L. N. Delía, G. Cáseres, L. C. Corbalán, F. Tesone, A. Cuitiño, and P. M. Pesado, "Mobile application development approaches: a comparative analysis on the use of storage space," in *XXIV Congreso Argentino de Ciencias de la Computación (La Plata, 2018).*, 2018.

[13] L. Corbalán, P. Thomas, L. Delía, G. Cáseres, J. F. Sosa, F. Tesone, and P. Pesado, "A study of non-functional requirements in apps for mobile devices," in *Conference on Cloud Computing and Big Data*, pp. 125–136, Springer, 2019.

[14] G. Cáseres, J. F. Sosa, F. Tesone, and P. Pesado, "A study of non-functional requirements in apps for mobile devices," in *Cloud Computing and Big Data: 7th Conference, JCC&BD 2019, La Plata, Buenos Aires, Argentina, June 24–28, 2019, Revised Selected Papers*, vol. 1050, p. 125, Springer, 2019.

[15] Q. H. Mahmoud, S. Zanin, and T. Ngo, "Integrating mobile storage into database systems courses," in *Proceedings of the 13th annual conference on Information technology education*, pp. 165–170, 2012.

[16] M. Fotache and D. Cogean, "Nosql and sql databases for mobile applications. case study: Mongodb versus postgresql.," *Informatica Economica*, vol. 17, no. 2, 2013.

[17] S. Lee, "Creating and using databases for android applications," *International Journal of Database Theory and Application*, vol. 5, no. 2, 2012.

[18] "Db-engines - about us." `https://db-engines.com/en/about`. Accedido por última vez: 17/02/2021.

[19] "Db-engines ranking - method." `https://db-engines.com/en/ranking_definition`. Accedido por última vez: 17/02/2021.

[20] "Mobile databases - wikipedia." `https://en.wikipedia.org/wiki/Mobile_database`. Accedido por última vez: 04/03/2021.

[21] "Oracle database lite release notes." `https://docs.oracle.com/cd/E12095_01/doc.10303/e12094/toc.htm`. Accedido por última vez: 04/03/2021.

[22] "Microsoft sql server compact 4.0 - microsoft lifecycle | microsoft docs." `https://docs.microsoft.com/en-us/lifecycle/products/microsoft-sql-server-compact-40`. Accedido por última vez: 04/03/2021.

[23] "Site search results for downloads." `https://www.microsoft.com/en-us/search/DownloadsDrillInResults.aspx?q=sql+server+`

express&cateorder=1_5_2_3_11&site=. Accedido por última vez: 04/03/2021.

[24] "Ibm db2 everyplace v9.1 offers a small footprint relational data store and enterprise." https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=ca&infotype=an&appname=iSource&supplier=897&letternum=ENUS206-182. Accedido por última vez: 04/03/2021.

[25] "Replicación de datos móviles (android o ios) - pc soft - documentación en línea." https://ayuda.windev.es/?1000021071. Accedido por última vez: 04/03/2021.

[26] A. Silberschatz, H. F. Korth, S. Sudarshan, *et al.*, *Database system concepts*, vol. 4. Mcgraw-hill New York, 1997.

[27] "Db-engines ranking - popularity ranking of relational dbms." https://db-engines.com/en/ranking/relational+dbms. Accedido por última vez: 21/02/2021.

[28] "About sqlite." https://www.sqlite.org/about.html. Accedido por última vez: 21/02/2021.

[29] "Sqlite system properties." https://db-engines.com/en/system/SQLite. Accedido por última vez: 21/02/2021.

[30] "Descripción general del almacenamiento de archivos y datos." https://developer.android.com/training/data-storage. Accedido por última vez: 21/02/2021.

[31] "Cómo guardar contenido en una base de datos local con room." https://developer.android.com/training/data-storage/room. Accedido por última vez: 21/02/2021.

[32] "Cómo guardar datos con sqlite | desarrolladores de android." https://developer.android.com/training/data-storage/sqlite. Accedido por última vez: 21/02/2021.

[33] "Data persistence in ios. ways to store data in ios world | by imran jutt | medium." https://medium.com/@imranjutt/data-persistence-in-ios-2804d04bde62. Accedido por última vez: 21/02/2021.

[34] "Data persistence in ios apps with swift - overview - swift tutorial." https://www.iosapptemplates.com/blog/ios-development/data-persistence-ios-swift. Accedido por última vez: 21/02/2021.

[35] "Core data or sqlite or realm? | apple developer forums." https://developer.apple.com/forums/thread/649649. Accedido por última vez: 21/02/2021.

[36] "Sqlite | ionic documentation." https://ionicframework.com/docs/native/sqlite. Accedido por última vez: 21/02/2021.

[37] "Getting started with sqlite in react-native | by sambhav jain | codeburst." https://codeburst.io/getting-started-with-sqlite-in-react-native-e25e361b8205. Accedido por última vez: 21/02/2021.

[38] "React native sqlite storage. examples of query uses in sqlite | by infinitbility | infinitbility | medium." https://medium.com/infinitbility/react-native-sqlite-storage-422503634dd2. Accedido por última vez: 21/02/2021.

[39] "Store data in a local sqlite.net database - xamarin | microsoft docs." https://docs.microsoft.com/en-us/xamarin/get-started/quickstarts/database?pivots=macos. Accedido por última vez: 21/02/2021.

[40] "Persistencia de datos en xamarin usando sqlite." https://somostechies.com/persistencia-de-datos-en-xamarin-usando-sqlite/. Accedido por última vez: 21/02/2021.

[41] "Persist data with sqlite - flutter." https://flutter.dev/docs/cookbook/persistence/sqlite. Accedido por última vez: 22/02/2021.

[42] "Características - embarcadero website." https://www.embarcadero.com/es/products/interbase/features/sql-database. Accedido por última vez: 03/03/2021.

[43] "Ediciones de productos - embarcadero website." https://www.embarcadero.com/es/products/interbase/product-editions. Accedido por última vez: 03/03/2021.

[44] "Sap sql anywhere | rdbms for iot & data-intensive apps | technical information." https://www.sap.com/products/sql-anywhere/technical-information.html. Accedido por última vez: 03/03/2021.

[45] "Sap sql anywhere system properties." https://db-engines.com/en/system/SAP+SQL+Anywhere. Accedido por última vez: 03/03/2021.

[46] "Opentext gupta sqlbase." https://www.opentext.com/products-and-solutions/products/specialty-technologies/opentext-gupta-development-tools-databases/opentext-gupta-sqlbase. Accedido por última vez: 03/03/2021.

[47] "Sqlbase system properties." https://db-engines.com/en/system/SQLBase. Accedido por última vez: 03/03/2021.

[48] "Db-engines ranking - popularity ranking of document stores." https://db-engines.com/en/ranking/document+store. Accedido por última vez: 21/02/2021.

[49] "Lite | couchbase." https://www.couchbase.com/products/lite. Accedido por última vez: 21/02/2021.

[50] "Apache couchdb." https://couchdb.apache.org/. Accedido por última vez: 21/02/2021.

[51] “Pouchdb, the javascript database that syncs!.” https://pouchdb.com/. Accedido por última vez: 21/02/2021.

[52] “Couchbase system properties.” https://db-engines.com/en/system/Couchbase. Accedido por última vez: 21/02/2021.

[53] “Database query language n1ql: Sql for json | couchbase.” https://www.couchbase.com/products/n1ql. Accedido por última vez: 21/02/2021.

[54] “Introduction | couchbase docs.” https://docs.couchbase.com/couchbase-lite/current/introduction.html. Accedido por última vez: 21/02/2021.

[55] “nativescript-couchbase | nativescript marketplace.” https://market.nativescript.org/plugins/nativescript-couchbase/. Accedido por última vez: 22/02/2021.

[56] “couchbase_lite | flutter package.” https://pub.dev/packages/couchbase_lite. Accedido por última vez: 22/02/2021.

[57] “Firebase realtime database | firebase realtime database.” https://firebase.google.com/docs/database. Accedido por última vez: 21/02/2021.

[58] “Firebase pricing.” https://firebase.google.com/pricing. Accedido por última vez: 22/02/2021.

[59] “Integrate firebase with ionic - ionic.” https://ionicframework.com/integrations/firebase. Accedido por última vez: 22/02/2021.

[60] “React native firebase | react native firebase.” https://rnfirebase.io/. Accedido por última vez: 22/02/2021.

[61] “nativescript-plugin-firebase | nativescript marketplace.” https://market.nativescript.org/plugins/nativescript-plugin-firebase/. Accedido por última vez: 22/02/2021.

[62] "Nuget gallery | xamarin.firebase.database 119.6.0." https://www.nuget.org/packages/Xamarin.Firebase.Database/. Accedido por última vez: 22/02/2021.

[63] "Agrega firebase a tu app de flutter." https://firebase.google.com/docs/flutter/setup?hl=es. Accedido por última vez: 22/02/2021.

[64] "Home | realm.io." https://realm.io/. Accedido por última vez: 22/02/2021.

[65] "Realm · github." https://github.com/realm. Accedido por última vez: 22/02/2021.

[66] "Realm system properties." https://db-engines.com/en/system/Realm. Accedido por última vez: 22/02/2021.

[67] "Introduction to mongodb realm for mobile developers — mongodb realm." https://docs.mongodb.com/realm/get-started/introduction-mobile/. Accedido por última vez: 22/02/2021.

[68] "Cordova / phonegap / ionic support · issue #261 · realm/realm-js · github." https://github.com/realm/realm-js/issues/261. Accedido por última vez: 22/02/2021.

[69] "Github - capacitor-community/realm." https://github.com/capacitor-community/realm. Accedido por última vez: 22/02/2021.

[70] "Cloud firestore | firebase." https://firebase.google.com/docs/firestore/. Accedido por última vez: 22/02/2021.

[71] "Elige una base de datos: Cloud firestore o realtime database." https://firebase.google.com/docs/firestore/rtdb-vs-firestore. Accedido por última vez: 22/02/2021.

[72] "Google cloud firestore system properties." https://db-engines.com/en/system/Google+Cloud+Firestore. Accedido por última vez: 22/02/2021.

[73] "Uso y límites | firebase." https://firebase.google.com/docs/firestore/quotas. Accedido por última vez: 22/02/2021.

[74] "Building ionic apps with firestore - ionic blog." https://ionicframework.com/blog/building-ionic-apps-with-firestore/. Accedido por última vez: 22/02/2021.

[75] "Cloud firestore | react native firebase." https://rnfirebase.io/firestore/usage. Accedido por última vez: 22/02/2021.

[76] "Nuget gallery | xamarin.firebase.firestore 122.1.0." https://www.nuget.org/packages/Xamarin.Firebase.Firestore/. Accedido por última vez: 22/02/2021.

[77] "Cloud firestore | flutterfire." https://firebase.flutter.dev/docs/firestore/usage/. Accedido por última vez: 22/02/2021.

[78] "Introduction to pouchdb." https://pouchdb.com/guides/. Accedido por última vez: 03/03/2021.

[79] "Pouchdb system properties." https://db-engines.com/en/system/PouchDB. Accedido por última vez: 03/03/2021.

[80] "Faq." https://pouchdb.com/faq.html. Accedido por última vez: 03/03/2021.

[81] "pouchdb-react-native - npm." https://www.npmjs.com/package/pouchdb-react-native. Accedido por última vez: 03/03/2021.

[82] "Litedb :: A .net embedded nosql database." http://www.litedb.org/. Accedido por última vez: 03/03/2021.

[83] "Object oriented dbms - db-engines encyclopedia." https://db-engines.com/en/article/Object+oriented+DBMS. Accedido por última vez: 04/03/2021.

[84] “Oracle berkeley db.” https://www.oracle.com/database/technologies/related/berkeleydb.html. Accedido por última vez: 03/03/2021.

[85] “Oracle berkeley db system properties.” https://db-engines.com/en/system/Oracle+Berkeley+DB. Accedido por última vez: 03/03/2021.

[86] “Objectbox system properties.” https://db-engines.com/en/system/ObjectBox. Accedido por última vez: 03/03/2021.

[87] “Mobile database | android database | ios database | flutter database.” https://objectbox.io/mobile-database/. Accedido por última vez: 03/03/2021.

[88] “Sparsity-technologies: Sparksee high-performance graph database.” http://sparsity-technologies.com/#sparksee. Accedido por última vez: 03/03/2021.

[89] “Sparksee system properties.” https://db-engines.com/en/system/Sparksee. Accedido por última vez: 03/03/2021.

[90] “Cómo hacer referencia a datos complejos con room.” https://developer.android.com/training/data-storage/room/referencing-data. Accedido por última vez: 27/02/2021.

[91] “Cómo definir relaciones entre objetos | desarrolladores de android.” https://developer.android.com/training/data-storage/room/relationships. Accedido por última vez: 27/02/2021.