

Instituto Politécnico Nacional
Escuela Superior de Cómputo

Trabajo Terminal No. 2019-B052

Herramienta para transformar un modelo
entidad-relación a un modelo no relacional

Presentan:

Aparicio Quiroz Omar
Martínez Acosta Eduardo

Directoras:

M. en C. Ocotitla Rojas Nancy
D. en C. Chavarria Baez Lorena

Índice general

1. Introducción	7
1.1. Descripción del problema	8
1.2. Propuesta de solución	9
1.3. Justificación	9
1.4. Objetivos	9
1.5. Alcance y limitaciones	10
2. Estado del Arte	12
2.1. Kashliev Data Modeler	12
2.2. NoSQL Workbench for Amazon DynamoDB	13
2.3. HacKolade	13
2.4. Mortadelo	14
2.5. NoSE: Schema Design for NoSQL Applications	17
2.6. Conclusiones	18
3. Marco Teórico	19
3.1. Modelos de datos para bases de datos	19
3.1.1. Modelo entidad-relación	20
3.1.1.1. Entidades	20
3.1.1.2. Atributos	20
3.1.1.3. Relaciones	21
3.1.1.4. Resumen de la notación para los diagramas ER	23
3.1.2. Modelo relacional	24
3.1.2.1. Relaciones	25
3.1.2.2. Claves	26
3.1.2.3. Restricciones de integridad	26
3.1.2.4. Propiedades de las relaciones	27

3.1.2.5.	Structured Query Language	27
3.1.3.	Modelos NoSQL	27
3.1.3.1.	Clave-Valor	28
3.1.3.2.	Orientado a documentos	29
3.1.3.3.	Orientado a columnas	30
3.1.3.4.	Orientado a grafos	31
3.2.	Tecnologías a usar	32
3.2.1.	Hypertext Transfer Protocol	32
3.2.2.	HTML 5	33
3.2.3.	Style Sheet Language: CSS vs SASS	34
3.2.4.	JavaScript vs TypeScript	35
3.2.5.	JavaScript Web Frameworks: Vue/Nuxt vs React vs Angular	37
3.2.6.	CSS Frameworks: Vuetify vs Bootstrap	40
3.2.7.	MySQL vs MongoDB	40
3.2.8.	Bibliotecas JavaScript para diagramado: GoJS vs Fabric.js vs D3.js	42
3.2.9.	Python 3	43
3.2.10.	Django vs Flask	45
3.3.	Conclusiones	47
4.	Análisis y Diseño del Sistema	48
4.1.	Metodología	48
4.1.1.	Scrum	49
4.1.1.1.	Historias de usuario	50
4.1.1.2.	Lista de producto (<i>product backlog</i>)	56
4.2.	Análisis de factibilidad	59
4.2.1.	Factibilidad técnica	59
4.2.1.1.	Sistema operativo	59
4.2.1.2.	Lenguaje de desarrollo	59
4.2.1.3.	Sistema gestor de base de datos	60
4.2.2.	Factibilidad económica	61
4.2.2.1.	Métricas orientadas a la función	61
4.2.2.2.	Puntos de función	61
4.2.3.	Costos de desarrollo	63

5. Algoritmos	66
5.1. Validación estructural diagrama entidad-relación	66
5.2. Modelo entidad-relación a relacional	71
5.3. Obtención de esquema SQL desde modelo relacional	75
5.4. Modelo entidad-relación a Generic Data Metamodel	75
5.5. Generic Data Metamodel a modelo lógico NoSQL	75
5.6. Modelo lógico NoSQL a modelo físico en MongoDB	77
 A. Apéndice	 79
A.1. Unified Modeling Language	79

Índice de figuras

2.1. Mortadelo	14
2.2. Generic Data Metamodel	15
2.3. Notación textual del GDM	16
2.4. Modelo lógico orientado a documentos de Mortadelo	17
3.1. Tabla en el modelo relacional	25
3.2. <i>Bucket</i> en el almacenamiento clave-valor	28
3.3. Colección en el almacenamiento de documentos	29
3.4. Familia de columnas	31
3.5. modelo conceptual orientado a grafos	31
5.1. Taxonomía de relaciones recursivas	67
5.2. Access Tree - Modelo lógico orientado a documentos	76
5.3. Access query Q4	76
A.1. Asociación	80
A.2. Agregación	81
A.3. Asociación rombo sin rellenar y composición rombo negro	81

Índice de tablas

2.1. Tabla comparativa de las herramientas estudiadas y la propuesta de solución.	18
3.1. Notación del modelo entidad-relación	24
4.1. Lista de producto	58
4.2. Equipo de cómputo	60
4.3. Cálculo de las métricas por puntos de función	61
4.4. Factores de ajuste	62
4.5. Costos del personal	63
4.6. Costos por licencias de software	64
4.7. Costos por servicios	64
5.1. Tipos de relación recursiva válidos según las restricciones de cardinalidad	68
5.2. Resumen de reglas de validez para relaciones recursivas con ejemplos	69
5.3. Resumen de reglas de validez para relaciones binarias con ejemplos .	70
5.4. Correspondencia entre los modelos ER y relacional.	74

Resumen

Capítulo 1

Introducción

En los últimos años, los sistemas *Not only SQL* o NoSQL han surgido como alternativa a los sistemas de bases de datos relacionales y se enfocan, principalmente, en buscar resolver problemáticas inherentes al usarlas en algunos escenarios.

Los sistemas NoSQL admiten diferentes modelos de datos como los de clave-valor, documentos, columnas y grafos; con ello se enfocan en guardar datos de una manera apropiada de acuerdo a los requisitos actuales para la gestión de datos en la web o en la nube, enfatizando la escalabilidad, la tolerancia a fallos y la disponibilidad a costa de la consistencia.

De acuerdo a Google Trends[1], en 2009 aumentó el interés en los sistemas NoSQL y desde entonces, también empezaron a resaltar algunas problemáticas propias de estos sistemas, porque si bien resuelven algunos dilemas de las bases de datos relacionales, por enfocarse en la ya mencionada escalabilidad, la tolerancia a fallos o la flexibilidad para realizar cambios en el esquema de la base de datos, la heterogeneidad de los sistemas NoSQL ha llevado a una amplia diversificación de las interfaces de almacenamiento de datos, provocando la pérdida de un paradigma de modelado común o de un lenguaje de consultas estándar como SQL.

Respecto al modelado de datos, en el diseño de las bases de datos tradicionales, el modelo entidad-relación[2] es el modelo conceptual más usado y tiene procedimientos bien establecidos como resultado de décadas de investigación; sin embargo, para los sistemas NoSQL los enfoques tradicionales de diseño de bases de datos no proporcionan un soporte adecuado para satisfacer el modelado de sus diferentes modelos de datos y para abordar esta problemática se han creado varias metodologías de diseño para sistemas NoSQL en los últimos años, porque los diseñadores de bases de datos deben tener en cuenta no solo qué datos se almacenarán en la base de datos, sino también cómo se accederán a ellos para modelar estos sistemas [3]-[5].

Para tener en cuenta cómo se accederán a los datos se debe conocer cómo se realizarán las consultas de los mismos y de acuerdo al trabajo de Mosquera[6], que es una investigación de 1376 *papers* sobre el modelado de sistemas NoSQL, se muestra que de las metodologías propuestas la mayoría usa el lenguaje de modelado UML (*Unified Modeling Language*), mientras que algunos proponen su propio modelo conceptual y en cada propuesta presentan una manera de representar las consultas de los datos; en resumen, en la literatura sobre el tema solo existen cinco herramientas

propuestas para el modelado conceptual y, en general, no hay una tendencia en el modelado de datos NoSQL.

Lo que resta del capítulo está organizado de la siguiente manera: primero se muestra la problemática a resolver, después la propuesta de solución, la justificación, los objetivos del proyecto y por último se menciona el alcance con las limitaciones del mismo.

1.1. Descripción del problema

Como se ha visto en la introducción, son pocas las herramientas propuestas en la literatura para el modelado de sistemas NoSQL en sus tres niveles de abstracción (nivel conceptual, lógico y físico).

Solo en el modelado conceptual, el modelo entidad-relación puede considerarse como una tendencia, sin embargo, el modelo entidad-relación por sí mismo no es suficiente para representar cómo se consultarán los datos ni tampoco con qué frecuencia se accederán a ellos; por eso, para modelar sistemas NoSQL es necesario conocer enfoques de desarrollo como el *query-driven design*, el *domain-driven design*, el *data-driven design* o el *workload-driven design*.

En general, para los modelos de datos NoSQL no hay un modelo estándar ni tampoco existe un acuerdo sobre la mejor definición de reglas de transformación entre sus tres niveles de abstracción; por ejemplo, en la literatura sobre el tema hay unos 36 estudios que proponen diferentes enfoques para las transformaciones.

Por los pocos años que han pasado desde que el interés sobre el tema aumentó, que son usados estos sistemas de bases de datos y los diferentes enfoques de transformación entre modelos, aún no se ven reflejados en los programas de estudio o solo se da una introducción del tema a los estudiantes de licenciatura en ingeniería en sistemas o carreras afines.

Asimismo, el estudiante tiene problemas para diseñar una base de datos relacional, porque desde el modelado conceptual tiene que confiar en sus conocimientos recientemente adquiridos para verificar la validez de los diagramas entidad-relación que desarrolla o consultarlo con el maestro de turno.

En consecuencia, para el estudiante es un problema tener que enfrentarse a diseñar un sistema NoSQL, porque no tiene la certeza de desarrollar diagramas entidad-relación válidos, no se le enseña ninguna metodología de desarrollo para sistemas NoSQL, tampoco conoce los distintos modelos de datos NoSQL que hay, mucho menos sus ventajas o desventajas para poder elegir el modelo más apropiado para su aplicación y, como resultado, no visualiza el diseño de una base de datos no relacional a partir de los conocimientos adquiridos en la asignatura de Bases de Datos.

Además, la escasez de herramientas CASE para el diagramado de esquemas no relacionales o herramientas que apoyen la migración de un modelo entidad-relación o relacional a uno no relacional aumenta la complejidad para que el estudiante haga uso de una base de datos no relacional.

1.2. Propuesta de solución

Desarrollar una aplicación web que permita:

- La edición de un diagrama entidad-relación.
- Realizar la validación estructural del diagrama entidad-relación.
- Transformar el diagrama entidad-relación a un diagrama del modelo relacional.
- Obtener el esquema de la base de datos en sentencias SQL del diagrama relacional.
- Obtener el modelo conceptual de un modelo de datos NoSQL.
- Obtener el esquema de datos NoSQL en un NoSQL DBMS (*NoSQL Database Management System*).

1.3. Justificación

Las pocas herramientas para modelado conceptual NoSQL no ofrecen validaciones para sus diagramas ni dan la posibilidad de obtener el esquema SQL, ya que no están enfocados en sistemas relacionales y se enfocan en un modelo lógico y físico específico para cada tipo sistemas NoSQL (clave-valor, orientado a documentos, columnas o grafos).

Asimismo, como se mencionó en la introducción, por los pocos años que han pasado desde que el interés y el uso de los sistemas NoSQL aumentó, estos sistemas todavía no están reflejados en los programas de estudio.

En consecuencia, para abordar esta problemática en la que el estudiante diseñe un sistema NoSQL sin haber aprendido ninguna metodología de desarrollo para dichos sistemas y sin que logre visualizar en primera instancia el diseño de una base de datos no relacional a partir de lo que ve en la asignatura de Bases de Datos, se propone una herramienta CASE que apoye al estudiante a diseñar una base de datos NoSQL a partir de los conocimientos adquiridos del estudiante.

Se usará como modelo conceptual el modelo entidad-relación por ser un modelo bien conocido para el estudiante de licenciatura en sistemas, que está probado en el área de las bases de datos y de acuerdo con Mosquera[6], es también el más usado en las investigaciones sobre el modelado conceptual de sistemas NoSQL.

1.4. Objetivos

Objetivo general

Desarrollar una aplicación web que permita la edición de un diagrama de bases de datos bajo el modelo entidad-relación y realice la validación del mismo; el

diagrama podrá ser transformado al modelo relacional con la posibilidad de obtener el esquema de la base de datos en sentencias SQL, o bien, obtendrá el modelo conceptual en un modelo de datos no relacional y tendrá la posibilidad de implementar el modelo no relacional en un NoSQL DBMS (*NoSQL Database Management System*).

Objetivos específicos

La edición del diagrama entidad-relación (o diagrama ER) se implementará con alguna biblioteca para diagramado, en donde al usuario se le permitirá arrastrar los entidades, atributos y relaciones del diagrama ER desde una “paleta” a la zona para diagramar, conocida también como “canvas”.

La validación del diagrama ER se realizará con eventos de escucha en el “canvas” o zona de diagramado y será de acuerdo a reglas de validación que se expondrán más adelante.

La transformación del diagrama ER al diagrama del esquema relacional se realizará con alguno de los algoritmos bien conocidos en la literatura del tema y se generará el esquema de sentencias SQL desde el diagrama del esquema relacional.

Se usará algún modelo conceptual NoSQL que ya exista en la literatura del tema.

Finalmente, se generará el esquema de una base de datos NoSQL desde el modelo conceptual elegido y se probará en un único NoSQL DBMS.

1.5. Alcance y limitaciones

De acuerdo con Dullea[7], un diagrama ER es válido solo si es estructural y semánticamente válido; sin embargo, hasta donde se sabe, en los últimos 17 años no hay estudios sobre la validez semántica de un diagrama ER, porque como expresa Dullea en su trabajo, la validez semántica depende del minimundo que se quiere representar en el diagrama y es imposible definir una métrica generalizada; por ello la validez de un diagrama ER será solo estructuralmente.

Se usará la notación oficial en los modelos de datos conceptuales entidad-relación y relacional, es decir, la notación de Chen y la notación de Codd, respectivamente.

Para la generación de *scripts* del esquema de datos relacional se usará el lenguaje de consultas SQL.

Asimismo, los *scripts* generados se probarán con MySQL porque es el DBMS que se usa en la asignatura de Base de Datos y es con el que están más familiarizados los estudiantes.

Para la transformación del modelo ER al esquema del modelo relacional no se realizará ninguna normalización, porque está fuera del alcance para el equipo programar la lógica necesaria y se necesita un diagrama no normalizado para la transformación al modelo conceptual NoSQL.

Dado que hay varias propuestas en la literatura sobre modelos conceptuales NoSQL, se optará por usar la propuesta de Alfonso de la Vega[8], ya que el modelo que propone, conocido como Generic Data Metamodel, es un metamodelo conceptual que describe un modelo de datos NoSQL independientemente de si es de clave-valor, orientado a documentos, a columnas o grafos.

Los *scripts* NoSQL generados por la aplicación solo serán probados y ejecutados en un único NoSQL DBMS.

Se elegirá el modelo de datos orientado a documentos y se usará MongoDB para probar los *scripts* NoSQL generados por la aplicación, porque de acuerdo a Mosquera[6], las de bases de datos orientadas a documentos son las más estudiadas y MongoDB es el NoSQL DBMS más probado para modelado físico en la literatura del tema.

Respecto al registro de los usuarios, se podrá acceder a la aplicación con un correo válido o uno que no lo sea, teniendo en cuenta de que si no crea su cuenta con un correo válido, no le llegará notificación de correo que le indica que se ha registrado exitosamente junto con su contraseña.

El diagramador ER solo podrá editar elementos del modelo entidad-relación, no se podrá diagramar elementos del modelo entidad-relación extendido ni de ningún otro modelo.

No se podrá editar el diagrama generado para el esquema del modelo relacional; tampoco se podrá editar el diagrama conceptual del modelo NoSQL.

Capítulo 2

Estado del Arte

De acuerdo con varios autores [4], [9], [10], un modelo conceptual que especifique solo qué entidades conforman el sistema no es suficiente, porque para los sistemas NoSQL es clave conocer cómo las entidades del modelo conceptual serán consultadas.

Debido a lo anterior, estos autores complementan los modelos conceptuales tradicionales como el modelo entidad-relación con propuestas sobre cómo conocer los patrones de acceso; sin embargo, ¿qué herramientas usan estas propuestas?

Para responder esta pregunta en este capítulo se presenta una investigación de herramientas similares, empezando por dos inspiradas en el modelo conceptual propuesto por Chebotko en 2015.

Además, se muestran otras tres herramientas, todas basadas en diferentes propuestas sobre modelado conceptual, explicando a detalle Mortadelo, la herramienta que hace uso del modelo conceptual elegido en el proyecto para representar una base de datos NoSQL.

Lo que resta de este capítulo está organizado de la siguiente manera: se presenta la descripción de cada oferta y se finaliza en las conclusiones con una tabla comparativa.

2.1. Kashliev Data Modeler

De acuerdo al sitio web de KDM[11], Kashliev Data Modeler es una herramienta de modelado desarrollada el profesor asistente del Departamento de informática de la Universidad del Oeste Michigan, Andrii Kashliev, que automatiza el diseño de esquemas para una base de datos NoSQL orientada a columnas en Apache Cassandra.

Por medio de una aplicación web KDM ayuda al usuario comenzando con un modelo conceptual ER más consultas asociadas para generar un modelo lógico y físico de datos o *scripts* CQL (Cassandra Query Language).

KDM automatiza:

1. El mapeo conceptual a lógico.

-
2. El mapeo lógico a físico.
 3. La generación de *scripts* CQL.

El modelo conceptual usado en KDM es el propuesto por Chebotko[4] y automatiza el proceso de transformación entre los tres niveles de modelos (conceptual, lógico y físico).

Además, cuenta con una versión de prueba de tiempo indefinido con características limitadas que permite la generación de un modelo lógico y guardar los proyectos.

2.2. NoSQL Workbench for Amazon DynamoDB

De acuerdo al sitio web de Amazon[12], NoSQL Workbench for Amazon DynamoDB es una herramienta desarrollada por Amazon que proporciona funciones de desarrollo de consultas, modelado y visualización de datos para diseñar, crear, consultar y administrar bases de datos NoSQL orientadas a columnas.

NoSQL Workbench for Amazon DynamoDB usa el modelo conceptual propuesto por Chebotko[4].

El visualizador de modelo de datos proporciona un lienzo donde se asignan consultas y visualizan las facetas (parte de la base de datos) de la aplicación sin tener que escribir código.

Cada faceta corresponde a un patrón de acceso diferente en DynamoDB, donde cada patrón se agrega manualmente; cuenta con un generador de operaciones para ver, explorar y consultar conjuntos de datos.

Por último, admite la proyección, la declaración de expresiones condicionales y permite generar código de muestra en varios idiomas.

2.3. HacKolade

De acuerdo al sitio web de la herramienta[13], Hackolade es un *software* con capacidad de representar objetos JSON anidados; la aplicación combina la representación gráfica de colecciones (término usado en las bases de datos orientados a documentos) y vistas en un diagrama.

La aplicación está basada en la denormalización, el polimorfismo y las matrices anidadas JSON; la representación gráfica de la definición del esquema JSON de cada colección está en una vista de árbol jerárquica.

Lamentablemente, en el sitio web de HacKolade no hay información sobre el modelo conceptual en el que está basado su aplicación.

Es una herramienta de modelado de datos para MongoDB, Neo4j, Cassandra, Couchbase, Cosmos DB, DynamoDB, Elasticsearch, HBase, Hive, Google BigQuery, Firebase/Firestore, MarkLogic, entre otros.

2.4. Mortadelo

De acuerdo con de la Vega[8], Mortadelo está basado en el *model driven*; es decir que para diseñar bases de datos NoSQL necesita de modelos conceptuales bien definidos.

Lo anterior implica usar dos modelos diferentes pero interrelacionados que añaden complejidad al modelado de la base datos; por esta razón de la Vega propone el Generic Data Metamodel (GDM), que es un modelo conceptual NoSQL donde la estructura (entidades, atributos, relaciones entre entidades) y patrones de acceso (cómo se consultarán los datos) están integradas en un mismo modelo conceptual.

Cabe destacar que el GDM es un modelo conceptual independiente del paradigma NoSQL, por lo que puede representar una base de datos NoSQL de clave-valor, orientado a documentos, orientado a columnas u orientado a grafos.

La figura 2.1 muestra los tres pasos principales de Mortadelo, empezando desde la izquierda con su propuesta de modelo conceptual NoSQL, el Generic Data Metamodel, siguiendo con una representación de los modelos lógicos orientado a columnas o documentos en los que realiza “modelo a modelo” (M2M) para generar el modelo lógico, para finalizar una transformación “modelo a texto” (M2T) con el modelo físico de cada modelo lógico, respectivamente.

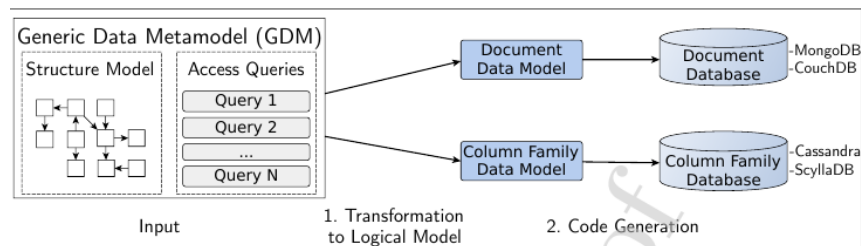


Figura 2.1: Mortadelo

Mortadelo: modelo conceptual (Generic Data Metamodel)

La figura 2.2 muestra el Generic Data Metamodel, que está compuesto por clases interrelacionadas entre sí con notación UML¹ y consta de dos secciones principales: los elementos de la estructura (*structure model elements*) y el cómo se realizarán las consultas (*access queries elements*).

¹La simbología usada en los diagramas de clase UML está en el apéndice



Figura 2.2: Generic Data Metamodel

A continuación se da una explicación de cada elemento de la figura 2.2 donde la clase *model* es para indicar que un modelo GDM tiene n entidades y n consultas donde $n = 0, 1, \dots, n$.

Structure model elements

- Clase *entity*: contiene *features* y solo es referenciada directamente desde las clases *from* y *reference*.
- Clase *feature*: es una clase abstracta, una *reference*, un *attribute*, o hereda de la clase *annotatable element* para que la instancia de la clase sea comentada con indicadores de texto que proveen información extra para generar el modelo lógico.
- Clase *reference*: empieza con la palabra clave “ref”, un nombre de tipo de entidad, una cardinalidad y un nombre de referencia. Por ejemplo: “ref Category[*] categories” define una referencia llamada *categories*, del tipo de entidad *category* con una cardinalidad de cero a varios.
- Clase *attribute*: contiene un tipo y un identificador de unicidad.
- Clase *annotatable element*: es una clase abstracta para permitir que una clase contenga anotaciones.
- Clase *annotation*: es un indicador de texto que proveen sobre información extra para generar el modelo lógico.

Access queries elements

- Clase *query*: tiene solo un elemento de la clase *from*, n elementos de la clase *attribute selection*, n elementos de la clase *inclusion* y tiene o no un único elemento de la clase *boolean expression*.
- Clase *from*: asocia una clase *query* con la clase *entity*; es la clase que permite referenciar un tipo de entidad.

- Clase *attribute selection*: accede a los atributos de la *entity* referenciada por la clase *from* o la clase *inclusion*.
- Clase *boolean expression*: expresa una expresión booleana para declarar alguna restricción.
- Clase *inclusion*: permite acceder en una *query* a los atributos de otros tipos de entidad.

La figura 2.3 es una instancia del modelo conceptual GDM en su notación textual en la que se nota que, por ejemplo, en la tercera consulta se accede a los elementos de la entidad *category* a través del elemento *ref* de la entidad *product*.

```
// Entities
entity Product {
  id productId
  text name
  text description
  number price
  ref Category[*]
    categories
  ref Provider[1]
    provider
}

entity Category {
  id categoryId
  text name
  text description
}

entity Provider {
  id providerId
  text name
}

// Queries
query Q1_productsById:
  select prod.productId,
    prod.name, prod.price,
    prod.description
  from Product as prod
  where prod.productId = "?"

query Q2_productsAndCategoryByName:
  select prod.name, prod.price,
    prod.description,
    cat.name
  from Product as prod
  including prod.categories as cat
  where prod.name = "?"

query Q3_productsByCategory:
  select prod.name, prod.price,
    prod.description,
    cat.name
  from Product as prod
  including prod.categories as cat
  where cat.name = "?"
  order by prod.price
```

Figura 2.3: Notación textual del GDM

Mortadelo: modelo lógico orientado a documentos

De acuerdo con de la Vega[8], las bases de datos orientadas a documentos tienen como objetivo almacenar jerarquías de objetos que son probables que se consultarán juntas.

Estas jerarquías de objetos se conocen como documentos y se agrupan en colecciones; asimismo, una colección almacena documentos de la misma entidad y por lo general un documento está compuesto de pares clave-valor y contiene otros documentos incrustados.

En la figura 2.4 se muestra el modelo lógico orientado a documentos de la herramienta Mortadelo y a continuación se explica cada clase que conforma el modelo.



Figura 2.4: Modelo lógico orientado a documentos de Mortadelo

- Clase *document data model*: está compuesta de n *collections*.
- Clase *collection*: tiene un nombre que la identifica y en una colección se almacenan documentos que, en general, comparten la misma estructura.
- Clase *document type*: define la estructura de los documentos a guardar con un conjunto de instancias de la clase *field*.
- Clase *field*: tiene instancias de las clases *primitive field* (atributos simples), *array field* (más instancias de la clase *field*) o *document type* (otras estructuras de documentos), permitiendo guardar elementos anidados.
- Clase *primitive field*: contiene un tipo primitivo de dato
- Clase *array field*: permite anidar más instancias de la clase *field*

Como se ha mostrado, Mortadelo es quizá la herramienta más completa de las estudiadas y Alfonso de la Vega expone en su *paper* no solo el modelado conceptual, lógico y físico de una base de datos NoSQL, sino también explica a detalle los algoritmos usados con casos de uso.

2.5. NoSE: Schema Design for NoSQL Applications

De acuerdo con Michael J. Mior[5], NoSE usa un modelo conceptual junto con el *workload* para describir cómo se accederán a los datos y así genera un modelo físico de una base de datos NoSQL orientado a columnas.

NoSE debe tener un modelo conceptual que describa la información que se almacenará, por ello NoSE espera este modelo conceptual en forma de un grafo de entidad.

Los grafos de entidad son un tipo restringido del modelo de entidad-relación; cada cuadro representa un tipo de entidad, tienen atributos en los que uno más

sirven como clave para identificarla, cada borde es una relación entre entidades y la cardinalidad asociada de la relación (uno a muchos, uno a uno o muchos a muchos).

Respecto al *workload*, se describe como un conjunto de consultas parametrizadas y declaraciones de “actualización”; cada consulta y actualización está asociada con un peso que indica su frecuencia relativa en la carga de trabajo.

2.6. Conclusiones

Como se ha podido ver en este capítulo, son pocas las herramientas para modelado conceptual NoSQL; de ellas, dos usan la propuesta de Chebotko para modelar conceptualmente bases de datos orientadas a columnas y las demás herramientas modelan bases de datos NoSQL con base en diferentes propuestas.

Asimismo, es de notar que ninguna herramienta permite la validación estructural del modelo conceptual que usan, ni tampoco generan el esquema del modelo relacional, porque no están enfocadas a modelar bases de datos relacionales y por la misma razón no producen los *scripts* de sentencias SQL del esquema relacional.

Para finalizar este capítulo, la tabla 2.1 muestra una comparación de las diferentes características de cada herramienta expuesta en este capítulo.

Herramienta	Creador	Objetivo	Licencia	Sistema Operativo	Fecha de publicación	Modelo			Patrones de Acceso	Metodología	Validación Estructural	Modelado Relacional	Esquema SQL
						Conceptual	Lógico	Físico					
KDM	Andrii Kashliev	comercial	privativa	Web	2015	entidad-relación	columnas	Cassandra	queries	query-driven	N/A	N/A	N/A
HacKolade	IntegrIT SA / NV	comercial	privativa	Windows, Mac, Linux	2016	entidad-relación	multiparadigma	Multiparadigma	sin especificar	sin especificar	N/A	N/A	N/A
NoSQL Workbench for Amazon DynamoDB	Amazon	comercial	privativa	Windows, Mac	2019	entidad-relación	columnas	MongoDB	queries	query-driven	N/A	N/A	N/A
Mortadelo	Alfonso de la Vega	investigación	libre	Web	2018	GDM	columnas documentos	Cassandra MongoDB	queries	query-driven	N/A	N/A	N/A
NoSE	Michael J. Mior	investigación	libre	Web	2016	grafos de entidades	documentos	MongoDB	workload	workload-driven	N/A	N/A	N/A
Propuesta de solución	TT 2019-B052	investigación	libre	Web	2020	entidad-relación	GDM	MongoDB	queries	query-driven	Sí	Sí	Sí

Tabla 2.1: Tabla comparativa de las herramientas estudiadas y la propuesta de solución.

Capítulo 3

Marco Teórico

Del estado del arte se ha concluido que el modelo entidad-relación es el modelo conceptual más usado para describir modelos de datos NoSQL; también es de notar que ninguna herramienta de las estudiadas ofrece validación estructural de su modelo conceptual, obtención del esquema relacional o de las sentencias SQL.

En este apartado se muestran los diferentes conceptos con el que el lector debe estar relacionado para comprender la solución que se propone y el resto del capítulo está organizado de la siguiente manera: primero se muestran los modelos de datos para bases de datos, empezando con el modelo entidad-relación, donde está cómo interactúan entre sí sus elementos (entidades, atributos y relaciones) para representar una base de datos relacional; después se expone el modelo relacional y una descripción de SQL (su lenguaje de consultas); también se muestra de forma concisa y breve los cuatro modelos de datos NoSQL (clave-valor, orientado a columnas, orientado a documentos y orientado a grafos); se explica cada tecnología y el porqué ha sido elegida para la propuesta de solución; finalmente, se termina con las conclusiones del capítulo.

3.1. Modelos de datos para bases de datos

De acuerdo con Elmasri [14], un modelo de datos es una colección de conceptos que describen la estructura de una base de datos.

Los modelos de datos conceptuales o de alto nivel ofrecen conceptos visuales simples que representan un modelo de datos, mientras que los modelos de datos físicos o de bajo nivel son detalles de cómo se implementa el almacenamiento de los datos en el sistema de la base de datos.

Una entidad representa un objeto o concepto del mundo real; un atributo representa alguna propiedad que describe una entidad y una relación es una asociación entre entidades.

A continuación se presentan varios de los modelos de datos existentes.

3.1.1. Modelo entidad-relación

De acuerdo con Elmasri [14], el modelo entidad-relación (o modelo ER) fue creado por Peter Chen en 1976[15] para el diseño conceptual de bases de datos y está conformado de entidades, atributos y relaciones.

3.1.1.1. Entidades

El objeto básico representado por el modelo ER es una entidad, que es una cosa del mundo real con una existencia independiente; una entidad es un objeto con existencia física o conceptual.

Tipo de entidad débil vs regular Un tipo de entidad define un conjunto de entidades con los mismos atributos; un tipo de entidad sin atributo clave se denomina tipo de entidad débil; en contraposición, un tipo de entidad regular con atributo clave se denomina tipo de entidad fuerte.

Las entidades que pertenecen a un tipo de entidad débil se identifican como relacionadas con un tipo de entidad propietaria en combinación con uno de sus valores de atributo.

El tipo de relación que relaciona un tipo de entidad débil con su propietaria se llama relación identificativa del tipo de entidad débil.

Un tipo de entidad débil siempre tiene una restricción de participación total (dependencia de existencia) respecto a su relación identificativa, porque una entidad débil no se identifica sin una entidad propietaria; no obstante, no toda dependencia de existencia produce un tipo de entidad débil.

Un tipo de entidad débil posee una clave parcial, que es el conjunto de atributos que identifican inequívocamente las entidades débiles que están relacionadas con la misma entidad propietaria.

En los diagramas ER tanto el tipo de la entidad débil como la relación identificativa se distinguen rodeando sus rectángulos y rombos mediante líneas dobles.

El atributo de clave parcial aparece subrayado con una línea discontinua o punteada; los tipos de entidades débiles se representan a veces como atributos complejos (compuestos o multivalor).

En general, se define cualquier cantidad de niveles de tipos de entidad débil; un tipo de entidad propietaria puede ser un tipo de entidad débil.

Además, es posible que un tipo de entidad débil tenga más de un tipo de entidad identificativa y un tipo de relación identificativa de grado superior a dos.

3.1.1.2. Atributos

Cada entidad posee atributos, que son propiedades particulares que la describen y en el modelo ER hay varios tipos: simple, compuesto, monovalor,

multivalor, almacenado, derivado y nulo.

Atributos compuestos vs atributos simples Los atributos compuestos se dividen en subpartes más pequeñas que representan atributos más básicos con significados independientes.

Los atributos que no son divisibles se denominan atributos simples o atómicos, mientras que los atributos compuestos forman una jerarquía.

El valor de un atributo compuesto es la concatenación de los valores de sus atributos simples.

Atributos monovalor vs multivalor La mayoría de los atributos poseen un solo valor para una entidad en particular; dichos atributos reciben el nombre de monovalor o de un solo valor.

En algunos casos, un atributo es un conjunto de valores para la misma entidad y se denominan multivalor.

Un atributo multivalor tiene un límite superior y uno inferior para restringir el número de valores permitidos para cada entidad individual.

Atributos almacenados vs derivados El atributo derivado se calcula u obtiene a partir de un atributo almacenado.

Atributos complejos Los atributos complejos son atributos compuestos y multivalor que se anidan arbitrariamente.

Atributos clave de un tipo de entidad Un atributo clave identifica inequívocamente cada tipo de entidad, cuenta con un nombre subrayado dentro del óvalo y algunos tipos de entidad poseen más de un atributo clave y si carece de clave, se le denomina tipo de entidad débil.

Atributos de los tipos de relación Los atributos de los tipos de relación 1:1 o 1:N se trasladan a uno de los tipos de entidad participantes.

En el caso de un tipo de relación 1:N, un atributo de relación solo se migra al tipo de entidad que se encuentra en el lado N de la relación.

Para los tipos de relación M:N, algunos atributos se determinan mediante la combinación de entidades participantes en una instancia de relación, no mediante una sola relación; dichos atributos deben especificarse como atributos de relación.

3.1.1.3. Relaciones

Un tipo de relación R entre n tipos de entidades E_1, E_2, \dots, E_n define un conjunto de relaciones entre las entidades de esos tipos de entidades.

Como en el caso de los tipos de entidades y los conjuntos de entidades, normalmente se hace referencia a un tipo de relación y su correspondiente conjunto de relaciones con el mismo nombre R .

Matemáticamente, el conjunto de relaciones R es un conjunto de instancias de relación r_i , donde cada r_i asocia n entidades individuales (e_1, e_2, \dots, e_n) y cada entidad e_j de r_i es un miembro del tipo de entidad E_j , $1 \leq j \leq n$. Por tanto, un tipo de relación es una relación matemática en E_1, E_2, \dots, E_n .

En los diagramas ER, los tipos de relación se muestran mediante rombos, conectados a su vez mediante líneas a los rectángulos que representan los tipos de entidad participantes y el nombre de la relación se muestra dentro del rombo.

Grado de un tipo de relación El grado de un tipo de relación es el número de tipos de entidades participantes; un tipo de relación de grado dos se denomina binario, de grado tres ternario y de grado n n-ario.

Nombres de rol y relaciones recursivas Cada tipo de entidad que participa en un tipo de relación juega un papel o rol particular en la relación.

El nombre de rol hace referencia al papel que una entidad participante del tipo de entidad juega en cada instancia de relación y ayuda a explicar el significado de la relación.

Los nombres de rol no son técnicamente necesarios en los tipos de relación donde todos los tipos de entidad participantes son distintos, puesto que cada nombre de tipo de entidad participante se utiliza como participación.

Cuando un tipo de entidad se relaciona consigo misma, se tiene una relación recursiva y es necesario indicar los roles que juegan los miembros en la relación.

Restricciones en los tipos de relación Los tipos de relación normalmente tienen ciertas restricciones que limitan las posibles combinaciones entre las entidades que participan en el conjunto de relaciones correspondiente y están determinadas por la situación del minimundo representado; se distinguen dos tipos principales de restricciones de relación: razón de cardinalidad y participación.

Razones de cardinalidad para las relaciones binarias La razón de cardinalidad de una relación binaria especifica el número máximo de instancias de relación en las que una entidad participa.

Las posibles razones de cardinalidad para los tipos de relación binaria son 1:1, 1:N, N:1 y M:N.

1. Uno a uno: una relación R de X a Y es uno a uno si cada entidad en X se asocia con cuando mucho una entidad en Y e, inversamente, cada entidad en Y se asocia con cuando mucho una entidad en X .
2. Uno a muchos: una relación R de X a Y es uno a muchos si cada entidad en X se asocia con muchas entidades en Y , pero cada entidad en Y se asocia con

cuando mucho una entidad en X .

3. Muchos a uno: una relación R de X a Y es muchos a uno si cada entidad en X se asocia con cuando mucho una entidad en Y , pero cada entidad en Y se asocia con muchas entidades en X .
4. Muchos a muchos: una relación R de X a Y es muchos a muchos si cada entidad en X se asocia con muchas entidades en Y y cada entidad en Y se asocia con muchas entidades en X .

Restricciones de participación y dependencias de existencia Hay dos tipos de restricciones de participación, total y parcial; la restricción de participación determina si la existencia de una entidad depende de su relación con otra entidad y especifica el número mínimo de instancias de relación en las que participa cada entidad.

1. Participación total: si todo miembro de un conjunto de entidades debe participar en una relación, es una participación total del conjunto de entidades en la relación. Esto se denota al dibujar una línea doble desde el rectángulo de entidades hasta el rombo de relación.
2. Participación parcial: una línea sencilla indica que algunos miembros del conjunto de entidades no deben participar en la relación.

La razón de cardinalidad y las restricciones de participación, en conjunto, son restricciones estructurales de un tipo de relación.

3.1.1.4. Resumen de la notación para los diagramas ER

Finalmente, la tabla 3.1 es un resumen de la simbología usada en los diagramas entidad-relación en la que se muestra en la primera columna la representación gráfica y a su lado su significado en el modelo.


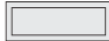










Notación del modelo entidad-relación	
Símbolo	Significado
	Entidad
	Entidad débil
	Relación
	Relación de identificación
	Atributo
	Atributo clave
	Atributo multivalor
	Atributo compuesto
	Atributo derivado
	Participación total
	Razón de cardinalidad
	Restricción estructural

Tabla 3.1: Notación del modelo entidad-relación

3.1.2. Modelo relacional

De acuerdo con Elmasri[14], el modelo relacional introducido por Ted Codd en 1970[2] utiliza el concepto de una relación matemática como bloque de construcción básico y tiene su base teórica en la teoría de conjuntos y la lógica del predicado.

La lógica de predicado, utilizada ampliamente en matemáticas, proporciona un marco en el que una afirmación (declaración de hecho) se verifica como verdadera o falsa.

La teoría de conjuntos es una ciencia matemática que trata con conjuntos o grupos de cosas y se utiliza como base para la manipulación de datos en el modelo relacional.

La figura 3.1 muestra de manera visual el modelo relacional; una tabla en el modelo relacional está compuesto de atributos, tiene un nombre de relación y tiene n tuplas.



Figura 3.1: Tabla en el modelo relacional

Como se observa en la imagen 3.1, el modelo relacional representa la base de datos como una colección de relaciones, siendo cada relación una tabla de valores, donde cada fila representa una colección de valores relacionados.

Asimismo, cada fila de la tabla representa un hecho que, por lo general, corresponde con una relación o entidad real; el nombre de la tabla y de las columnas se utiliza para ayudar a interpretar el significado de cada uno de los valores de las filas.

En terminología formal, una fila recibe el nombre de tupla, una cabecera de columna es un atributo y el nombre de la tabla una relación; el tipo de dato que describe los valores en cada columna está representado por un dominio de posibles valores.

Basado en estos conceptos, el modelo relacional tiene tres componentes bien definidos:

1. Una estructura de datos lógica representada por relaciones.
2. Un conjunto de reglas de integridad para garantizar que los datos sean consistentes.
3. Un conjunto de operaciones que define cómo se manipulan los datos.

3.1.2.1. Relaciones

En este modelo, las tablas se usan para contener información acerca de los objetos a representar en la base de datos.

Una relación se representa como una tabla bidimensional en la que las filas de la tabla corresponden a registros individuales y las columnas corresponden a atributos.

Formalmente, un esquema de relación R , denotado por $R(A_1, A_2, \dots, A_n)$ está constituido por un nombre de relación R y una lista de atributos A_1, A_2, \dots, A_n .

Un esquema de relación se utiliza para describir una relación; se dice que R es el nombre de la misma y el grado de una relación es su número de atributos n .

En el modelo relacional cada fila se llama tupla y la tabla que representa una relación tiene las siguientes características:

-
- Cada celda de la tabla contiene solo un valor.
 - Cada columna tiene el nombre del atributo que representa.
 - Todos los valores en una columna provienen del mismo dominio, pues todos son valores del atributo correspondiente.
 - Cada tupla o fila es distinta; no hay tuplas duplicadas.
 - El orden de las tuplas o filas es irrelevante.

Relaciones y tablas de bases de datos Una relación r del esquema $R(A_1, A_2, \dots, A_n)$, también especificado como $r(R)$ es un conjunto de n -tuplas $r = t_1, t_2, \dots, t_m$.

Cada tupla t es una lista ordenada de n valores $t = \langle v_1, v_2, \dots, v_n \rangle$, donde v_i , $1 \leq i \leq n$ es un elemento de $dom(A_i)$ o un valor especial nulo.

3.1.2.2. Claves

En el modelo relacional, las claves son importantes porque aseguran que cada fila en una tabla sea unívocamente identificable; son usadas para establecer relaciones entre tablas y asegurar la integridad de los datos.

Una clave es un atributo o grupo de atributos que identifican los valores de otros atributos y puede ser compuesta, clave o superclave.

Clave compuesta Una clave compuesta es una clave que se compone de más de un atributo y si forma parte de una clave se denomina atributo clave.

Superclave Un atributo o atributos que identifican de manera única cualquier fila de una tabla.

3.1.2.3. Restricciones de integridad

Integridad de dominio La integridad de dominio es la validez de las restricciones que debe cumplir una determinada columna de la tabla.

Integridad de entidad Todas las claves principales son únicas y ninguna clave primaria debe ser nula.

Integridad referencial Una clave externa puede ser nula siempre que no sea parte de la clave principal de su tabla o tiene el valor que coincida con el valor de la clave primaria en una tabla con la que está relacionada.

3.1.2.4. Propiedades de las relaciones

Grado El número de columnas en una tabla se llama grado de la relación, es parte de la intensión de la relación y nunca cambia; una relación con una sola columna es de grado uno y se llama relación unaria; con dos columnas se llama binaria; con tres columnas se llama ternaria y con más columnas n-aria.

Cardinalidad La cardinalidad de una relación es el número de entidades a las que otra entidad mapea dicha relación.

3.1.2.5. Structured Query Language

Structured Query Language (o SQL) está basado en el álgebra relacional, en el cálculo relacional y es un lenguaje de manipulación de datos, un lenguaje de definición de datos, un lenguaje de control de transacciones y un lenguaje de control de datos.

Lenguaje de manipulación de datos (DML) Un *Data Manipulation Language* (o DML) incluye comandos para insertar, actualizar, eliminar y recuperar datos dentro de las tablas de la base de datos.

Lenguaje de definición de datos (DDL) Un *Data Definition Language* (o DDL) incluye comandos para crear objetos de base de datos como tablas, índices y vistas, así como comandos para definir accesos a objetos de la base de datos.

Lenguaje de control de transacciones (TCL) Los comandos de un *Transaction Control Language* (o TCL) se ejecutan dentro del contexto de una transacción, que es una unidad lógica de trabajo compuesta por una o más instrucciones SQL y proporciona comandos para controlar el procesamiento de estas transacciones atómicas.

Lenguaje de control de datos (DCL) Los comandos de un *Data Control Language* (o DCL) se utilizan para controlar el acceso a los objetos de datos, como otorgar a un usuario permiso para ver solo una tabla y otorgar a otro usuario permiso para cambiar los datos de la misma tabla.

3.1.3. Modelos NoSQL

De acuerdo con Catherine [16], el término NoSQL significa *Not only SQL* y hay cuatro tipos principales de bases de datos NoSQL: clave-valor, orientado a columnas, orientado a documentos y orientado a grafos.

Se usan para agrupar sistemas de bases de datos diferentes a los relacionales y surgieron por los nuevos requerimientos de disponibilidad total, tolerancia a fallos, almacenamiento de penta *bytes* de información distribuida en miles de servidores, la

necesidad de nodos con escalabilidad horizontal, entre otros; a continuación se una breve explicación de cada modelo de datos NoSQL.

3.1.3.1. Clave-Valor

De acuerdo con Coronel[17], una base de datos de clave-valor es un paradigma de modelo de datos diseñado para almacenar, recuperar y administrar arreglos asociativos.

Comúnmente se usa un diccionario o tabla *hash* que contiene una colección de registros anidados, secuencias de bits que se almacenan y se recuperan utilizando una clave que identifica de manera única el registro y se utiliza para encontrar rápidamente los datos dentro de la base de datos.

No obstante, es responsabilidad de las aplicaciones que hagan uso de este tipo de base de datos interpretar el significado de los datos; no hay claves foráneas y las relaciones no son rastreables entre claves, lo que permite que el DBMS sea rápido y escalable.

La figura 3.2 es una representación visual de un *bucket* usado en las bases de datos NoSQL de clave-valor.

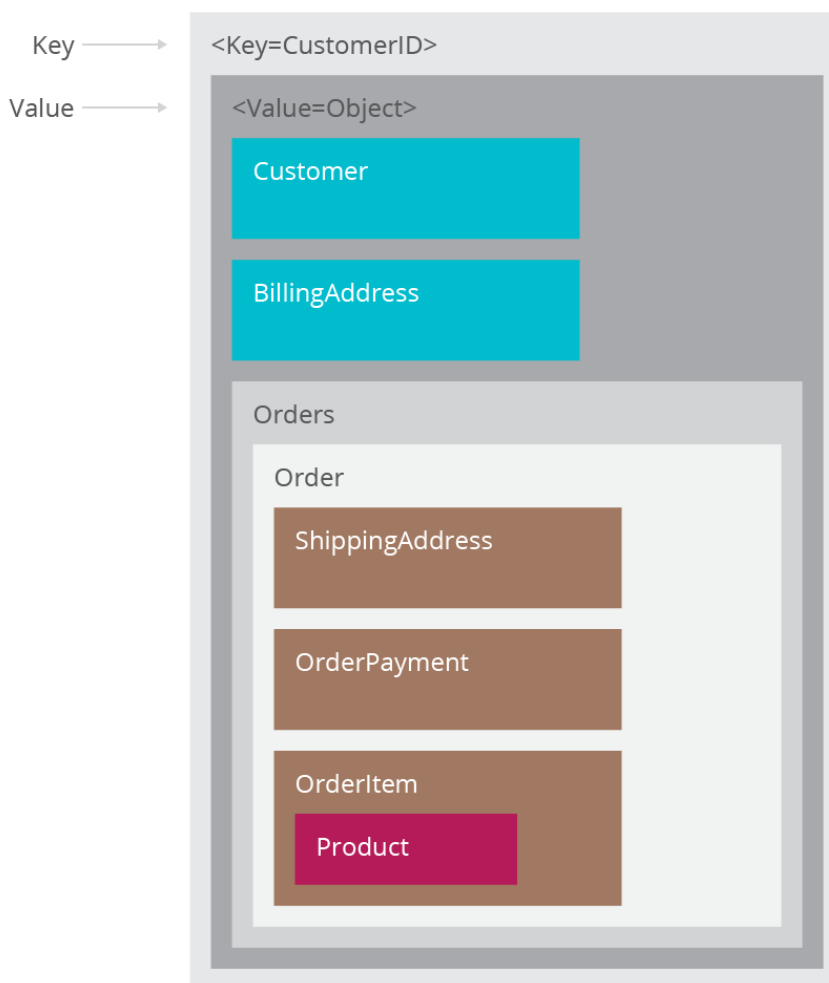


Figura 3.2: *Bucket* en el almacenamiento clave-valor

Los pares de clave-valor generalmente se organizan en *buckets*; todas las claves dentro un *bucket* deben ser únicas, pero está permitido que se repitan en otros *buckets* y todas las operaciones se basan en el *bucket* + la clave.

En este tipo de bases de datos se usan las operaciones de *get*, *store* y *delete*; la operación *get* o *fetch* es usada para obtener el valor de un par; el operador de *store* almacena datos en una clave.

Si la combinación de *bucket* + clave no existe, se añade como un nuevo par de clave-valor; en cambio, si existe la combinación de *bucket* + clave, el valor es reemplazado por el nuevo; el operador de *delete* es para eliminar un par de clave-valor.

De acuerdo con Sadalge[18], algunas de las bases de datos de clave-valor populares son Riak, Redis, Memcached DB, Berkeley DB, HamsterDB, Amazon DynamoDB y Project Voldemort (una implementación de código abierto de Amazon DynamoDB).

3.1.3.2. Orientado a documentos

De acuerdo con Coronel[17], una base de datos NoSQL orientada a documentos almacena datos en documentos etiquetados en pares clave-valor; sin embargo, a diferencia de una base de datos clave-valor donde el componente de valor contiene cualquier tipo de datos, una base de datos de documentos siempre almacena un documento en el componente de valor y puede estar en cualquier formato codificado como XML, JSON o BSON.

La figura 3.3 es la representación visual de una colección, donde tiene una clave como identificador único y documentos anidados.



Figura 3.3: Colección en el almacenamiento de documentos

Otra diferencia importante es que si bien las bases de datos clave-valor no

intentan comprender el contenido del componente de valor, las bases de datos de documentos sí lo hacen; por ejemplo, hay documentos con etiquetas para identificar qué texto en el documento representa el título, el autor y el cuerpo del documento.

Como se ve en la figura 3.3, dentro del cuerpo del documento existen etiquetas adicionales para indicar capítulos y secciones; a pesar del uso de etiquetas en los documentos, las bases de datos de documentos se consideran sin esquema, es decir, no imponen una estructura predefinida en los datos almacenados.

Para una base de datos de documentos, no tener esquemas significa que aunque todos los documentos tienen etiquetas, no todos tienen las mismas etiquetas, por lo es posible que cada documento tenga su propia estructura.

Las etiquetas en una base de datos de documentos son extremadamente importantes porque son la base de la mayoría de las capacidades adicionales que tienen las bases de datos de documentos sobre las bases de datos clave-valor.

Las etiquetas dentro del documento hacen posible consultas complejas para el DBMS; asimismo, al igual que las bases de datos clave-valor agrupan pares clave-valor en grupos lógicos llamados *buckets*, las bases de datos de documentos agrupan documentos en grupos lógicos llamados colecciones.

Si bien es posible recuperar un documento especificando la colección y la clave, también es posible realizar consultas en función del contenido de las etiquetas.

Las bases de datos de documentos tienden a funcionar bajo el supuesto de que un documento es independiente, o sea que no está en diferentes tablas como en una base de datos relacional.

Una base de datos de documentos asume que todos los datos relacionados de una consulta están en un solo documento; por ejemplo, cada consulta en una colección contendría datos sobre el cliente, el pedido en sí y los productos comprados.

Por último, las bases de datos de documentos no almacenan relaciones como se hace en el modelo relacional y generalmente no tienen soporte para operaciones como la unión.

3.1.3.3. Orientado a columnas

De acuerdo con Coronel[17], el modelo de base de datos NoSQL orientado a columnas se originó con el BigTable de Google.

La figura 3.4 representa una familia de columnas; la imagen de arriba es la partición de las diferentes familias de columnas y en la imagen de abajo se nota cada partición individual.



Figura 3.4: Familia de columnas

Las bases de datos de la familia de columnas son parecidas a las relaciones del modelo relacional y organizan datos en pares nombre-valor donde el nombre actúa también como la clave; como se nota en la figura 3.4, un par de clave-valor representa una columna y siempre contiene una fecha que sirve para resolver conflictos de escritura o datos expirados.

3.1.3.4. Orientado a grafos

De acuerdo con Coronel[17], una base de datos NoSQL orientada a grafos está basada en la teoría de grafos para almacenar datos con muchas relaciones.

La figura 3.5 representa un grafo de una biblioteca, donde cada rectángulo es un nodo y están asociados entre sí por relaciones.

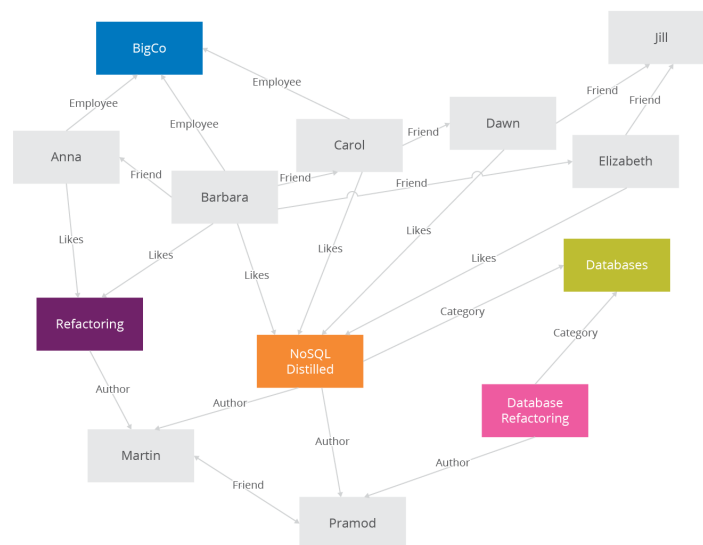


Figura 3.5: modelo conceptual orientado a grafos

Como se muestra en la figura 3.5, los componentes principales de las bases de

datos de grafos son nodos, aristas y propiedades; el nodo es una instancia específica para guardar datos.

Las propiedades son como atributos; son los datos que se necesitan almacenar sobre el nodo; todos los nodos tienen propiedades como nombre y apellido, pero no todos los nodos deben tener las mismas propiedades.

Un borde es una relación entre nodos, está representada por una flecha en la figura 3.5 y es posible que estén en una dirección o ser bidireccionales.

Para hacer una consulta se atraviesa el grafo y los recorridos se enfocan en las relaciones entre nodos, como la ruta más corta y el grado de conexión.

3.2. Tecnologías a usar

Para seleccionar las tecnologías a usar para el desarrollo de la aplicación web se ha optado por hacer un estudio y comparación de tecnologías similares.

Lo que resta de esta sección está organizado de la siguiente manera: primero se muestra cada tecnología usar; en caso de que sea la única opción se describirá qué es la herramienta y en caso de que haya varias opciones, se explicará cada opción y se tendrá un apartado al final de cada comparación sobre la herramienta que se ha elegido; finalmente, en la subsección de conclusiones se pondrá un resumen de cada tecnología escogida y el porqué.

3.2.1. Hypertext Transfer Protocol

De acuerdo con la W3C[19], Hypertext Transfer Protocol (HTTP, o protocolo de transferencia de hipertexto) es el protocolo de comunicación que permite transferir información en la World Wide Web.

HTTP fue desarrollado por el World Wide Web Consortium y la Internet Engineering Task Force, colaboración que culminó en 1999 con la publicación de varios RFC, siendo el más importante el RFC 2616 que especifica la versión 1.1 del protocolo.

HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores; sin embargo, el desarrollo de aplicaciones web necesita frecuentemente mantener un estado, por lo que se usan *cookies*, que son archivos generados en un servidor que son almacenados en el sistema cliente.

Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor; al cliente se le suele llamar “agente de usuario” (o *user agent*), que realiza una petición enviando un mensaje con cierto formato al servidor, mientras que al servidor se le suele llamar servidor web y envía un mensaje de respuesta.

HTTP tiene métodos de petición flexibles que permiten añadir nuevos métodos o funcionalidades; el número de métodos ha ido en aumento según se avanza en las versiones del protocolo donde los más importantes son:

-
1. Método *get*: solicita una representación del recurso especificado; solo deben recuperar datos.
 2. Método *head*: pide una respuesta idéntica a la que correspondería a una petición *get*, pero en la respuesta no se devuelve el cuerpo; esto es útil para poder recuperar los metadatos de los encabezados de respuesta, sin tener que transportar todo el contenido.
 3. Método *post*: envía datos para que sean procesados por un recurso identificado que se incluirán en el cuerpo de la petición.
 4. Método *put*: sube o carga un recurso especificado (archivo o fichero) y es más eficiente que el método *post*, porque permite escribir un archivo en una conexión socket establecida con el servidor.

3.2.2. HTML 5

De acuerdo con la documentación de Mozilla[20], HyperText Markup Language (HTML o lenguaje de marcado de hipertextos) es la pieza más básica en la construcción de la web, usada para definir el sentido y estructura del contenido en una página web.

Es un estándar a cargo del World Wide Web Consortium (W3C o Consorcio WWW), organización dedicada a la estandarización de casi todas las tecnologías ligadas a la web.

HTML hace uso de enlaces que conectan las páginas web entre sí, ya sea dentro de un mismo sitio web o entre diferentes sitios web.

Un elemento HTML se separa de otro texto en un documento por medio de “etiquetas”, las cuales consisten en elementos rodeados por “<,>”.

HTML 5 (HyperText Markup Language, versión 5) es la última revisión importante del lenguaje HTML en el que establece elementos y atributos que reflejan el uso de sitios web modernos.

Características:

1. Incorpora etiquetas: *canvas* 2D & 3D, audio y vídeo con códecs para mostrar los contenidos multimedia; actualmente hay una lucha entre imponer códecs libres (WebM + VP8) o privados (H.264/MPEG-4 AVC).
2. Etiquetas para manejar grandes conjuntos de datos: permiten generar tablas dinámicas para filtrar, ordenar y ocultar contenido en cliente.
3. Mejoras en los formularios: nuevos tipos de datos como *email*, *number*, *url*, *datetime* y facilidades para validar contenido sin JavaScript.
4. Visores: MathML (fórmulas matemáticas) y SVG (gráficos vectoriales).
5. Drag & Drop: nueva funcionalidad para arrastrar objetos como imágenes.

Respecto a la compatibilidad con los navegadores, la mayoría de elementos de HTML5 son compatibles con Firefox 19, Chrome 25, Safari 6 y Opera 12 en adelante.

3.2.3. Style Sheet Language: CSS vs SASS

De acuerdo con Lie[21], un *style sheet language* es un lenguaje que representa los estilos o elementos visuales de documentos estructurados.

CSS

De acuerdo a la documentación de la W3C[22], Cascading Style Sheets (CSS, o hojas de estilo en cascada) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado.

La separación entre el contenido del documento y la presentación busca mejorar la accesibilidad, proveer más flexibilidad y control, permitir que varios documentos HTML compartan un mismo estilo usando una sola hoja de estilos separada en un archivo .css, reducir la complejidad y la repetición de código.

La especificación CSS describe un esquema prioritario para determinar qué reglas de estilo se aplican si más de una regla coincide para un elemento en particular; estas reglas son aplicadas con un sistema llamado de cascada, de modo que las prioridades son calculadas y asignadas a las reglas, así que los resultados son predecibles.

La especificación CSS es mantenida por el World Wide Web Consortium; el tipo MIME *text/css* está registrado para su uso por CSS descrito en el RFC 23185.

CSS se ha creado en varios niveles y perfiles, donde cada nivel de CSS se construye sobre el anterior, generalmente añadiendo funciones al nivel previo.

La última versión del estándar, CSS3.1, está dividida en varios documentos separados, llamados “módulos”; cada módulo añade nuevas funcionalidades a las definidas en CSS2, de manera que se preservan las anteriores para mantener la compatibilidad.

Los trabajos en CSS3.1 comenzaron a la vez que se publicó la recomendación oficial de CSS2 y su primeros borradores fueron liberados en junio de 1999.

Debido a la modularización de CSS3.1, diferentes módulos están en diferentes estados de su desarrollo, hay alrededor de cincuenta módulos publicados, tres de ellos se convirtieron en recomendaciones oficiales de la W3C en 2011: “selectores”, “espacios de nombres” y “color”.

Respecto al soporte de los navegadores web, cada navegador web usa un motor de renderizado para renderizar páginas web y el soporte de CSS no es exactamente igual en ninguno de los motores de renderizado; ya que los navegadores no aplican el CSS correctamente, muchas técnicas de programación han sido desarrolladas para ser aplicadas por un navegador específico (comúnmente conocida esta práctica como *CSS hacks* o *CSS filters*).

Sass

De acuerdo con la documentación de Sass[23], Sass es un lenguaje de preprocesado que genera hojas de estilo en cascada (CSS) y consta de dos sintaxis.

Características

Variables Las variables comienzan con un signo de dólar y la asignación de valor se realiza con dos puntos y permite 4 tipos de datos:

1. Números (incluyendo las unidades).
2. Strings (con comillas o sin ellas).
3. Colores (código o nombre).
4. Booleanos.

Las variables son resultados o argumentos de varias funciones disponibles.

Anidamiento CSS soporta anidamiento lógico, pero los bloques de código no son anidados; Sass permite que el código anidado sea insertado dentro de cualquier otro bloque.

Mixins Como CSS no soporta *mixins*, cualquier código duplicado debe ser repetido en cada lugar; un *mixin* en Sass es una sección de código que contiene código Sass.

Cada vez que se llama un *mixin* en el proceso de conversión, el contenido del mismo es insertado en el lugar de la llamada; los *mixins* permiten una solución limpia a las repeticiones de código, así como una forma fácil de alterar el mismo.

Elección

Tomando en cuenta la experiencia del equipo con CSS, además de que el proyecto no se enfocará en hacer muchas hojas de estilo para cada componente, página o vista de la aplicación web, se usará CSS en lugar de Sass.

3.2.4. JavaScript vs TypeScript

De acuerdo con [stack overflow\[24\]](#), los web *frameworks* más populares están escritos en JavaScript/TypeScript, por ello se realizó una investigación del lenguaje más apto para el proyecto.

JavaScript

De acuerdo con la documentación de Mozilla[25], JavaScript es una marca registrada con licencia de Sun Microsystems (ahora Oracle) que se usa para describir la implementación del lenguaje de programación JavaScript.

Debido a problemas de registro de marcas en la Asociación Europea de Fabricantes de Computadoras, la versión estandarizada del lenguaje tiene el

nombre de ECMAScript, sin embargo, en la práctica se conoce como lenguaje JavaScript.

Abreviado como JS, es un lenguaje ligero e interpretado, orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico; es usado en node.js, Apache CouchDB y Adobe Acrobat.

El núcleo del lenguaje JavaScript está estandarizado por el Comité ECMA TC39 como un lenguaje llamado ECMAScript y la última versión de la especificación es ECMAScript 6.0 que define:

1. Sintaxis: reglas de análisis, palabras clave, flujos de control, inicialización literal de objetos.
2. Mecanismos de control de errores: *throw*, *try/catch*, habilidad para crear tipos de errores definidos por el usuario.
3. Tipos: *boolean*, *number*, *string*, *function*, *object*.
4. Objetos globales: en un navegador, los objetos globales son los objetos de la ventana, pero ECMAScript solo define una API no específica para navegadores, como *parseInt*, *parseFloat*, *decodeURI* o *encodeURI*.
5. Mecanismo de herencia basada en prototipos.
6. Objetos y funciones incorporadas.
7. Modo estricto.

La sintaxis básica es similar a Java y C++ con la intención de reducir el número de nuevos conceptos necesarios para aprender el lenguaje; las construcciones del lenguaje, como sentencias *if*, bucles *for*, *while*, bloques *switch* y *try catch* funcionan de la misma manera que en estos lenguajes (o casi).

JavaScript funciona como lenguaje procedimental y como lenguaje orientado a objetos; los objetos se crean añadiendo métodos y propiedades a lo que de otra forma serían objetos vacíos en tiempo de ejecución, en contraposición a las definiciones sintácticas de clases comunes en los lenguajes compilados como C++ y Java.

Las capacidades dinámicas de JavaScript incluyen construcción de objetos en tiempo de ejecución, listas variables de parámetros, variables que contienen funciones, creación de scripts dinámicos (mediante *eval*), introspección de objetos (mediante *for ... in*), y recuperación de código fuente.

Desde 2012 todos los navegadores modernos soportan completamente ECMAScript 5.1 y el 17 de Julio de 2015 ECMA International publicó la sexta versión de ECMAScript, oficialmente llamada ECMAScript 2015 y fue inicialmente nombrada como ECMAScript 6 o ES6. Desde entonces, los estándares ECMAScript están en ciclos de lanzamiento anuales.

TypeScript

De acuerdo con TypeScript Publishing[26], TypeScript es por definición JavaScript para el desarrollo de aplicaciones, siendo también un superconjunto del mismo.

TypeScript es un lenguaje compilado orientado a objetos, fue diseñado por Anders Hejlsberg (diseñador de C#) en Microsoft; es tanto un lenguaje como un conjunto de herramientas y es un superconjunto de JavaScript porque genera código JavaScript.

Características

- **Compilación:** cuenta con un transpilador para la verificación de errores si hay errores de compilación, cosa que no es posible con JavaScript.
- **Tipeo estático fuerte:** provee un sistema opcional de tipeo estático y de inferencia de tipos a través del TypeScript Language Service, lo que permite inferir el tipo de una variable declarada sin tipo en función de su valor.
- **Definiciones de tipo:** permite la extensión del lenguaje con bibliotecas externas JavaScript.
- **Programación orientada a objetos:** admite conceptos como clases, interfaces, herencia, etc.

Elección

De acuerdo con el sitio *stack overflow*[27], JavaScript es el lenguaje más popular de 2019 y aunque TypeScript es de los lenguajes que tienen un mayor nivel de aceptación, se usará JavaScript no solo por ser el lenguaje más popular y, en consecuencia, con más compatibilidad y material de ayuda, sino también porque el equipo está acostumbrado a este lenguaje y tiene experiencia con *web frameworks* escritos en JavaScript.

3.2.5. JavaScript Web Frameworks: Vue/Nuxt vs React vs Angular

De acuerdo con *Wired*[28], un *web framework* es un conjunto de *software* que permite el desarrollo de una aplicación web y en el lenguaje JavaScript hay varias opciones, incluidas las más populares Vue, React y Angular.

React

De acuerdo con el sitio web de React[29], es una biblioteca JavaScript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de *single page applications*.

Características

1. Virtual DOM: React usa un virtual DOM propio en lugar del navegador.
2. Props: son definidos como atributos de configuración para cada componente.
3. Estado de cada componente: lleva un registro de las propiedades y atributos del componente.
4. Ciclos de vida: son la serie de estados por los cuales pasan los componentes *statefull* a lo largo de su existencia.

Angular

De acuerdo con el sitio web de Angular[30], Angular es un web *framework* desarrollado en TypeScript de código abierto mantenido por Google que se utiliza para crear y mantener *single page applications*.

1. Generación de código.
2. Componentes.
3. Ciclos de vida de componentes.

Vue

De acuerdo con la documentación de Vue.js[31], Vue es un *framework* progresivo para desarrollar interfaces de usuario; a diferencia de otros *frameworks*, Vue está diseñado desde para ser utilizado incrementalmente.

La biblioteca central está enfocada solo en la capa de visualización y es fácil de utilizar e integrar con otras bibliotecas o proyectos existentes; por otro lado, Vue también es capaz de impulsar sofisticadas *single-page applications* cuando se utiliza en combinación con bibliotecas de apoyo.

Comparación con React React y Vue comparten muchas similitudes; ambos utilizan un DOM virtual, proporcionan componentes de vista reactivos, enrutamiento y la gestión global del estado manejado por bibliotecas asociadas.

Tanto React como Vue ofrecen un rendimiento comparable en los casos de uso más comunes, con Vue normalmente un poco por delante debido a su implementación más ligera del DOM virtual.

En Vue las dependencias de un componente se rastrean automáticamente durante su renderizado, por lo que el sistema sabe con precisión qué componentes deben volver a renderizarse cuando cambia el estado; se considera que cada componente tiene un *shouldComponentUpdate* automáticamente implementado.

Comparación con Angular En términos de rendimiento, ambos *frameworks* son excepcionalmente rápidos y no hay suficientes datos de casos de uso en el mundo real para hacer un veredicto.

Vue es mucho menos intrusivo en las decisiones del desarrollador que Angular, ofreciendo soporte oficial para una variedad de sistemas de desarrollo, sin restricciones sobre cómo estructurar su aplicación; muchos desarrolladores disfrutan de esta libertad, mientras que algunos prefieren tener solo una forma correcta de desarrollar cualquier aplicación.

Para empezar con Vue, todo lo que se necesita es familiarizarse con HTML y ES5 JavaScript, mientras que la curva de aprendizaje de Angular es mucho más pronunciada.

La complejidad de Angular se debe en su enfoque para diseñar aplicaciones grandes y complejas, pero eso hace que el *framework* sea mucho más difícil de entender.

Nuxt.js

De acuerdo a la documentación de Nuxt[32], el objetivo de Nuxt.js es hacer que el desarrollo web en Vue sea eficaz con herramientas de desarrollo como Webpack, Babel y PostCSS;

Características

1. Manejo de archivos Vue (*.vue).
2. División automática de código.
3. Representación del lado del servidor.
4. Potente sistema de enrutamiento con datos asincrónicos.
5. Servicio de archivos estáticos.
6. Soporte sintaxis ES2015+ (Javascript ES6).
7. Gestión de elementos <head> <title>, <meta> y similares.
8. Preprocesador: Sass, Less, Stylus, etc..

Elección

Se usará Vue/Nuxt por ser el *framework* con el que el equipo está más acostumbrado, además de ser el más flexible de las opciones expuestas.

3.2.6. CSS Frameworks: Vuetify vs Bootstrap

De acuerdo con Wikipedia[33], un CSS *framework* es una biblioteca de estilos genéricos usada para implementar diseños web y aportan una serie de utilidades que son aprovechadas frecuentemente en los distintos diseños web.

Vuetify

De acuerdo con la documentación de Google[34], Material Design es un lenguaje visual que sintetiza los principios clásicos del buen diseño respecto a las ideas de Google y en estos principios está basado Vuetify.

El objetivo de Material Design es crear un lenguaje visual que sintetice los principios clásicos del buen diseño, unificar el desarrollo de un único sistema subyacente para la experiencia del usuario en plataformas y dispositivos, así como personalizar el lenguaje visual de Material Design.

Asimismo, de acuerdo con la documentación de Vuetify[35], este CSS *framework* está integrado para ser usado en los componentes de Vue/Nuxt como botones, barras de navegación, *layouts* y demás.

Bootstrap

De acuerdo con la documentacion de Bootstrap[36], es un CSS *framework* orientado al diseño responsivo de una aplicación web.

Tiene *templates* para botones, barras de navegación, estilos de tipografía entre otros; es de fácil integración con React, Angular o Vue y tiene una comunidad extensa por los años y popularidad que tiene.

Elección

Se ha elegido usar en una primera instancia Vuetify porque es un CSS *framework* que está integrado en las tecnologías asociadas de Vue, como Vue Router, Vue Meta; asimismo, sus componentes son simples de entender y de implementar.

3.2.7. MySQL vs MongoDB

MongoDB

De acuerdo con la documentación de MongoDB[37], MongoDB (del inglés humongous, “enorme”) es un sistema de base de datos NoSQL, orientado a documentos y de código abierto.

En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

Características

1. Consultas ad hoc: MongoDB soporta la búsqueda por campos, consultas de rangos y expresiones regulares.
2. Indexación: es posible que cualquier campo en un documento de MongoDB sea indexado, al igual que es posible hacer índices secundarios.
3. Replicación: MongoDB soporta el tipo de replicación primario-secundario.
4. Balanceo de carga; MongoDB escala de forma horizontal usando el concepto de *sharding*.
5. Almacenamiento de archivos: MongoDB es utilizado como un sistema de archivos, aprovechando la capacidad de MongoDB para el balanceo de carga y la replicación de datos en múltiples servidores.
6. Agregación: MongoDB proporciona un framework de agregación que permite realizar operaciones similares al GROUP BY de SQL.

MySQL

De acuerdo con la documentación de MySQL[38], es un gestor de base de datos relacionales de código abierto con un modelo cliente-servidor.

Como es un gestor de base de datos relacional, archiva datos en tablas separadas en lugar de guardar todos los datos en un gran archivo, permitiendo tener mayor velocidad y flexibilidad; estas tablas están relacionadas de formas definidas, por lo que se hace posible combinar distintos datos en varias tablas y conectarlos.

Características

1. Permite escoger múltiples motores de almacenamiento para cada tabla.
2. Agrupación de transacciones, pudiendo reunir las de forma múltiple desde varias conexiones con el fin de incrementar el número de transacciones por segundo.
3. Conectividad segura.
4. Ejecución de transacciones y uso de claves foráneas.
5. Presenta un amplio subconjunto del lenguaje SQL.

Elección

De acuerdo con Mosquera[6], MongoDB es la base de datos NoSQL orientada a documentos más popular y usada en los *papers* de investigación; por ello el equipo ha decidido usar MongoDB como base de datos.

3.2.8. Bibliotecas JavaScript para diagramado: GoJS vs Fabric.js vs D3.js

GoJS

De acuerdo a la documentación de GoJS[39], GoJS es una biblioteca de JavaScript y TypeScript para crear diagramas y gráficos interactivos.

GoJS le permite crear todo tipo de diagramas y gráficos para sus usuarios, desde simples diagramas de flujo y organigramas hasta diagramas industriales altamente específicos, diagramas SCADA y BPMN, diagramas médicos como genogramas y diagramas de modelos de brotes, y más.

GoJS facilita la construcción de diagramas JavaScript de nodos complejos, enlaces y grupos con plantillas y diseños personalizables.

GoJS ofrece muchas funciones avanzadas para la interactividad del usuario, como arrastrar y soltar, copiar y pegar, edición de texto en el lugar, información sobre herramientas, menús contextuales, diseños automáticos, plantillas, enlace de datos y modelos, gestión de estado y deshacer transaccional, paletas, descripciones generales, controladores de eventos, comandos, herramientas extensibles para operaciones personalizadas y animaciones personalizables.

GoJS se implementa en TypeScript y puede usarse como una biblioteca de JavaScript o incorporarse a su proyecto desde fuentes de TypeScript.

GoJS normalmente se ejecuta completamente en el navegador, renderizando a un elemento HTML Canvas o SVG sin ningún requisito del lado del servidor.

Fabric.js

De acuerdo con la documentación de Fabric[40], Fabric.js es una biblioteca de JavaScript que proporciona un modelo para trabajar sobre un canvas HTML5 para poder agregar objetos como rectas, circunferencias, rectángulos, etc.

Características

1. Drag & Drop integrado en cada objeto de Fabric.js
2. Permite la especialización de clases para crear objetos personalizados.

D3.js

De acuerdo con Wikipedia[41], D3.js es una biblioteca de JavaScript para producir a partir de datos infogramas dinámicos e interactivos en navegadores web.

Características

1. Selecciones: se pueden seleccionar elementos del documento HTML y asignarle propiedades.

-
2. Transiciones: permiten interpolar en el tiempo valores de atributos, lo que produce cambios visuales en los infogramas.
 3. Asociación de datos: se asocia a cada elemento un objeto SVG con propiedades (forma, colores, valores) y comportamientos (transiciones, eventos).

Elección

Se llevó a la práctica en el prototipo funcional las tres opciones antes expuestas junto con algunas otras y se decidió usar GoJS para el proyecto por ser la biblioteca JavaScript más completa para diagramado y que genere los diagramas con un JSON simple para parsear esos datos y realizar la conversión.

3.2.9. Python 3

De acuerdo con Mark Lutz[42], Python es un lenguaje de programación interpretado, interactivo y orientado a objetos. Incorpora módulos, excepciones, tipo dinámico, tipos de datos dinámicos y clases.

Tiene sintaxis clara, interfaces para muchas llamadas de sistema y bibliotecas, así como para varios sistemas de ventanas. Además, es extensible en C o C++.

También se puede usar como un lenguaje de extensión para aplicaciones que necesitan una interfaz programable y es portátil: se ejecuta en muchas variantes de Unix, en Mac y en Windows 2000 y versiones posteriores.

La biblioteca estándar del lenguaje, cubre áreas como el procesamiento de cadenas (expresiones regulares, Unicode, cálculo de diferencias entre archivos), protocolos de Internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, programación CGI), ingeniería de software (pruebas unitarias, registro, creación de perfiles, análisis del código Python) e interfaces del sistema operativo (llamadas al sistema, sistemas de archivos, sockets TCP/IP).

Fortalezas

Es orientado a objetos y funcional Python es un lenguaje orientado a objetos; su modelo de clase admite nociones avanzadas como el polimorfismo, la sobrecarga del operador y la herencia múltiple; sin embargo, en el contexto de la simple sintaxis y escritura de Python, la programación orientada a objetos es fácil de aplicar.

Además de servir para la estructuración y reutilización de código, la naturaleza orientada a objetos de Python lo hace ideal como herramienta de secuencias de comandos para otros lenguajes de sistemas orientados a objetos. Por ejemplo, con el código apropiado, los programas Python pueden especializar clases implementadas en C++, Java y C#.

No obstante, la programación orientada a objetos es una opción en Python. Al igual que C++, Python admite modos de programación tanto procedimentales como orientados a objetos. Las herramientas orientadas a objetos se pueden aplicar siempre que las restricciones lo permitan.

Además de sus paradigmas originales de procedimientos (basados en declaraciones) y orientados a objetos (basados en clases), Python en los últimos años ha adquirido soporte incorporado para la programación funcional, un conjunto que incluye generadores, comprensiones, cerraduras, mapas, decoradores, funciones anónimas lambdas.

Es extensible Su conjunto de herramientas lo ubica entre los lenguajes de *scripting* tradicionales como Tcl, Scheme y Perl; y los lenguajes de desarrollo de sistemas como C, C++ y Java.

Python proporciona toda la simplicidad y facilidad de uso de un lenguaje de programación, junto con herramientas de ingeniería de software más avanzadas que normalmente se encuentran en lenguajes compilados.

A diferencia de algunos lenguajes de secuencias de comandos, esta combinación hace que Python sea útil para proyectos de desarrollo a gran escala. Algunas de las herramientas de Python son:

Escritura dinámica Python realiza un seguimiento de los tipos de objetos que utiliza su programa cuando se ejecuta; eso no requiere declaraciones complicadas de tipo y tamaño en su código. De hecho, no existe una declaración de tipo o variable en Python.

Debido a que el código Python no restringe los tipos de datos, también se aplica automáticamente a toda una gama de objetos.

Gestión automática de la memoria Python asigna automáticamente objetos y los reclama el recolector de basura cuando ya no se usan y la mayoría puede crecer y reducirse según la demanda. Es decir, Python realiza un seguimiento de los detalles de la memoria de bajo nivel.

Tipos de objetos incorporados Python proporciona estructuras de datos de uso común como listas, diccionarios y cadenas como partes intrínsecas del lenguaje. Son flexibles y fáciles de usar. Por ejemplo, los objetos integrados pueden crecer y reducirse según demanda, pueden anidarse arbitrariamente para representar información compleja, y más.

Herramientas incorporadas Para procesar todos esos tipos de objetos, Python viene con operadores potentes y estándar, que incluyen concatenación (unir colecciones), segmentar (extraer secciones), ordenar, mapear y más.

Utilidades de biblioteca Para tareas más específicas, Python también viene con una gran colección de herramientas de biblioteca precodificadas que admiten todo, desde la coincidencia de expresiones regulares hasta la creación de redes. Una vez que aprende el lenguaje en sí, las herramientas de la biblioteca de Python son donde ocurre gran parte de la acción a nivel de aplicación.

Utilidades de terceros Debido a que Python es de código abierto, los desarrolladores pueden contribuir con herramientas precodificadas que admitan tareas que aún no son herramientas estándar; en la Web, encontrará soporte gratuito para COM, imágenes, programación numérica, XML y acceso a bases de datos.

Elección

Se ha elegido Python para desarrollar los algoritmos del proyecto dado que es multiplataforma y es de fácil integración con el *framework* web elegido para el back end.

3.2.10. Django vs Flask

Para elegir el lenguaje a usar para el back end se realizó un estudio de lenguajes apropiados para usar con un JavaScript web *framework*.

Flask

Flask es un micro *web framework* escrito en Python. Se clasifica como micro porque no requiere herramientas o bibliotecas particulares.

No tiene capa de abstracción de base de datos, validación de formularios ni ningún otro componente donde las bibliotecas de terceros preexistentes brinden funciones comunes. Sin embargo, Flask admite extensiones que pueden agregar características de la aplicación como si se implementaran en el propio Flask.

Existen extensiones para mapeadores relacionales de objetos, validación de formularios, manejo de carga, varias tecnologías de autenticación abiertas y varias herramientas relacionadas con el marco común. Las extensiones se actualizan con mucha más frecuencia que el programa central Flask.

Ventajas y desventajas de Flask Está basado en la especificación WSGI de Werkzeug y el motor de templates Jinja2; además, tiene una licencia BSD.

Entre las ventajas y desventajas, destacamos:

Ventajas

1. Es un framework que se destaca en instalar extensiones o complementos de acuerdo al tipo de proyecto que se va a desarrollar, es decir, es perfecto para el prototipado rápido de proyectos.
2. Incluye un servidor web, así podemos evitamos instalar uno como Apache o Nginx. Además, nos ofrece soporte para pruebas unitarias y para Cookies de seguridad (sesiones del lado del cliente), apoyándose en el motor de plantillas Jinja2.

-
3. Su velocidad es mejor a comparación de Django. Generalmente el desempeño que tiene Flask es superior debido a su diseño minimalista que tiene en su estructura.
 4. Flask permite combinarse con herramientas para potenciar su funcionamiento, por ejemplo: Jinja2, SQLAlchemy, Mako y Peewee entre otras.

Desventajas

1. Su sistema de autenticación de usuarios es muy básico, a comparación del potente sistema de autenticación que utiliza Django, este puede crear un sistema de Login API sencillo para aplicaciones más pequeñas.
2. Su representación de Plugins no es tan extensa como la tiene Django.
3. Es complicado en las pruebas unitarias o migraciones.
4. El ORM (Mapeo objeto relacional) para conectar con las bases de datos, SQLAlchemy es externo.

Django

De acuerdo con Wikipedia[43], Django es un framework de desarrollo web de código abierto, escrito en Python, que respeta el patrón de diseño conocido como MVC (Modelo–Vista–Controlador).

Características

1. Aplicaciones “enchufables”que pueden instalarse en cualquier página gestionada con Django.
2. Una API de base de datos robusta.
3. Un sistema incorporado de “vistas genéricas”que ahorra tener que escribir la lógica de ciertas tareas comunes.
4. Un sistema extensible de plantillas basado en etiquetas, con herencia de plantillas.
5. Un despachador de URL basado en expresiones regulares.
6. Un sistema “middleware”para desarrollar características adicionales.
7. Documentación incorporada accesible a través de la aplicación administrativa.

Elección

Se ha elegido Flask por ser un *framework* web conocido por el equipo y que ha dado resultados.

3.3. Conclusiones

1. Tomando en cuenta la experiencia del equipo con CSS, además de que el proyecto no se enfocará en hacer muchas hojas de estilo para cada componente, página o vista de la aplicación web, se usará CSS en lugar de Sass.
2. De acuerdo con el sitio [stack overflow\[27\]](#), JavaScript es el lenguaje más popular de 2019 y aunque TypeScript es de los lenguajes que tienen un mayor nivel de aceptación, se usará JavaScript no solo por ser el lenguaje más popular y, en consecuencia, con más compatibilidad y material de ayuda, sino también porque el equipo está acostumbrado a este lenguaje y tiene experiencia con web *frameworks* escritos en JavaScript.
3. Se usará Vue/Nuxt por ser el *framework* con el que el equipo está más acostumbrado, además de ser el más flexible de las opciones expuestas.
4. Se ha elegido usar en una primera instancia Vuetify porque es un CSS *framework* que está integrado en las tecnologías asociadas de Vue, como Vue Router, Vue Meta. Asimismo, sus componentes son simples de entender y de implementar.
5. De acuerdo con Mosquera[6], MongoDB es la base de datos NoSQL orientada a documentos más popular y usada en los *papers* de investigación. Por ello se usará MongoDB como base de datos.
6. Se llevó a la práctica en el prototipo funcional las tres opciones antes expuestas junto con algunas otras y se decidió usar GoJS para el proyecto por ser la biblioteca JavaScript más completa para diagramado y que permite generar los diagramas con un JSON simple para parsear y realizar las transformaciones pertinentes.
7. Se ha elegido Python para desarrollar los algoritmos del proyecto dado que es multiplataforma y es de fácil integración con el *framework* web elegido para el back end.
8. Se ha elegido Flask por ser un *framework* web conocido por el equipo y que ha dado resultados.

Capítulo 4

Análisis y Diseño del Sistema

Aquí falta poner: El capítulo está organizado de la siguiente manera blablabla y tiene tal, tal y tal.

4.1. Metodología

De acuerdo a la Universidad Católica los Ángeles[44], en el campo del desarrollo de *software* hay dos grupos de metodologías: las tradicionales y las ágiles.

En cuanto a la documentación, las tradicionales suelen ser rígidas, exigiendo que sea exhaustiva y centrándose en cumplir con un plan de trabajo establecido en la etapa inicial del proyecto, mientras que las ágiles permiten realizar cambios en los requerimientos conforme a los avances del mismo.

Dado que cualquier cambio en el proceso de una metodología tradicional genera la necesidad de una reconstrucción del plan de trabajo y requiere invertir tiempo de desarrollo, surgieron las llamadas “metodologías ágiles”, que permiten realizar cambios en los requerimientos conforme a los avances en el proyecto. Tomando en cuenta las habilidades del equipo de desarrollo y una relación con el cliente; esta forma de trabajo permite mostrar avances funcionales en el producto en un periodo de tiempo corto para realizar una evaluación y en caso de ser requerido se sugieran cambios.

Se han propuesto muchos modelos ágiles de proceso y están en uso en toda la industria. Entre ellos se encuentran los siguientes:

- Desarrollo adaptativo de *software* (DAS).
- Scrum.
- Método de desarrollo de sistemas dinámicos (MDSD).
- Cristal.
- Desarrollo impulsado por las características (DIC).
- Desarrollo esbelto de *software* (DES).

-
- Modelado ágil (MA).
 - Proceso unificado ágil (PUA).

4.1.1. Scrum

De acuerdo con *The Scrum Guide*[45], Scrum es un marco de trabajo para la entrega de productos incrementales y de máximo valor productivo.

Un artefacto es un elemento que garantiza la transparencia, es el registro de la información fundamental del proceso Scrum y a continuación se describen sus cuatro artefactos principales:

- Lista de producto (*product backlog*): es el listado de todas las tareas que necesita el proyecto para alcanzar su realización. Al iniciar el desarrollo del proyecto, esta lista no se encuentra completa y conforme avanzan los *sprints* se le añaden tareas para solventar las necesidades que van surgiendo gracias a la retroalimentación del cliente.
- Lista de pendientes del *sprint* (*sprint backlog*): es la lista de tareas seleccionadas del *product backlog* que se planifica realizar durante el periodo del *sprint* y se definen a los responsables de cada tarea.
- *Sprint*: es el corazón de Scrum; tiene un periodo de tiempo determinado de un mes o incluso menos donde el equipo completa conjuntos de tareas incluidas en el *backlog* para crear un incremento del producto utilizable.
- Incremento: es la suma de todos los elementos de la lista de productos completados durante un *sprint*, unido con los incrementos de los *sprints* anteriores. Al finalizar el *sprint* el nuevo incremento debe estar en condiciones de ser utilizado.

La metodología nos permite definir un periodo de hasta un mes para cada *sprint* y se ha optado por un periodo de 30 días, contemplándose un total de ocho *sprints*, donde al término de cada uno se tendrá un avance del sistema.

Asimismo, Scrum tiene cuatro eventos formales contenidos dentro del *sprint* para la inspección y adaptación del producto que se describen a continuación:

- Planificación del *sprint* (*sprint planning*): es una reunión con todo el equipo Scrum que tiene una duración máxima de 8 horas para el *sprint* de un mes en la que el Scrum master es el encargado de que los asistentes entiendan el propósito de dicha reunión.
- Scrum diario (*daily scrum*): es una reunión de un máximo de 15 minutos en la cual el equipo de desarrollo expone sus actividades y planifica las tareas de las próximas 24 horas.
- Revisión del *sprint* (*sprint review*): al finalizar cada *sprint* se lleva a cabo una reunión para la revisión del incremento del producto y en caso de ser necesario realizar ajustes a la lista de producto.

-
- Retrospectiva del *sprint* (*sprint retrospective*): en esta fase, el equipo Scrum tiene la oportunidad de pensar en mejoras para el próximo *sprint* que contribuyan al proyecto.

El equipo Scrum consiste en los siguientes roles:

- El dueño de producto (*product owner*): es la persona responsable de maximizar el valor del producto y el trabajo del equipo de desarrollo; es el único responsable de gestionar la lista de producto y cualquier cambio a esa lista debe ser revisada y aprobada por él.
- El equipo de desarrollo (*development team*): es el conjunto de profesionales que realizan el trabajo para la entrega de un incremento en el producto en cada *sprint*; es un grupo autoorganizado y multifuncional donde cada miembro del equipo tiene habilidades especializadas pero que la responsabilidad de las tareas completadas, incrementos del producto o retrasos recaen en el equipo como un todo.
- El Scrum master: es la persona responsable de asegurar que Scrum es entendido y adoptado por todos los involucrados en el proyecto, asegurándose de ayudar a las personas externas al equipo Scrum a entender qué interacciones con el equipo lleguen a ser útiles y cuáles no.

4.1.1.1. Historias de usuario

De acuerdo con Scrum México[46], las historias de usuario es la técnica por la que el usuario especifica de manera general los requerimientos que el sistema debe cumplir.

Normalmente estas redacciones se llevan a cabo en tarjetas de papel donde se describen brevemente las funciones que el producto final debe poseer, ya sean requisitos funcionales o no.

El tratamiento de las historias de usuario es flexible y dinámico, cada una de ellas es lo suficientemente detallada y delimitada para que el equipo de desarrollo pueda implementarla durante la duración del *sprint*.

Es habitual que se siga una plantilla para estas tarjetas, como la que se expone a continuación:

- Como <Usuario>
- Quiero <algún objetivo>
- Para <motivo>

Una de sus grandes ventajas, dado el caso de que un usuario no sea lo suficientemente detallista con la historia, es que esta se puede partir en historias más pequeñas antes de que el equipo empiece a trabajar en ella.

Este es un ejemplo de historia de usuario para el desarrollo:

-
- Como usuario,
 - quiero poder ingresar al sistema con mi correo y contraseña
 - para tener acceso a sus funciones.

Otra forma de darle detalle a las historias de usuario es mediante el añadido de un criterio de aceptación. Un criterio de aceptación es una prueba que será cierta cuando el equipo de desarrollo complete la descripción de la tarjeta.

A continuación se listan las principales historias de usuario que se consideraron para el desarrollo de la propuesta de solución; tenga en cuenta que algunas de ellas tienen criterios de aceptación pero en otras no se consideró necesarias porque son explícitamente claras.

- N.º 1
 - Como usuario
 - quiero poder ingresar al sistema con mi correo y contraseña
 - para tener acceso a sus funciones.
 - Criterios de aceptación:
 - el usuario recibirá un correo electrónico de confirmación de su alta en el sistema con el correo y contraseña que ingresó para tenerlos de respaldo.
-

- N.º 2
 - Como usuario
 - quiero poder recuperar mi contraseña en caso de olvidarla
 - para no perder el trabajo realizado en el sistema.
 - Criterios de aceptación:
 - el usuario podrá ingresar una nueva contraseña siempre y cuando recuerde el correo electrónico con el que se dio de alta en el sistema.
 - al ingresar una nueva contraseña, recibirá un correo de confirmación del cambio de contraseña y sus datos permanecerán intactos.
-

- N.º 3
- Como usuario del sistema, quiero darme de alta con una contraseña facil de recordar

-
- pero que esté segura en la base de datos
 - para no tener comprometidos los diagramas que yo pueda generar en el sistema.
 - Criterios de aceptación:
 - asegurarse que el usuario ingrese una contraseña de al menos 8 caracteres.
 - se le solicitará al usuario que ingrese 2 veces la misma contraseña para asegurarse que le es fácil recordarla y que efectivamente es la misma.
 - antes de guardar la contraseña, esta deberá pasar por un método que la haga ilegible para el usuario (algún algoritmo de digestión o cifrado).
-

- N.º 4
 - Como usuario quiero poder crear un diagrama ER arrastrando y soltando elementos de una “paleta”
 - para hacerlo de manera mas fácil e intuitiva.
 - Criterios de aceptación:
 - el usuario podrá empezar un nuevo diagrama al seleccionar la opción de “diagramador ER”.
 - tendrá a su disposición una paleta con los elementos permitidos en un diagrama ER estándar.
 - podrá arrastrar y soltar los elementos de la paleta a un área delimitada para empezar con el diseño de su diagrama.
-

- N.º 5
 - Como usuario quiero poder guardar mi último trabajo realizado en el diagramador ER
 - para poder consultarlo en otro momento.
 - Criterios de aceptación:
 - dispondrá de un botón para poder guardar en la base de datos el diagrama que esté creando o editando.
 - antes de almacenar el diagrama en el *canvas* o zona de diagramado, se le mostrará un mensaje de confirmación para guardar su diagrama actual.
-

- N.º 6

-
- Como usuario me gustaría poder ver el último trabajo que realicé
 - cuando seleccione la opción “Entidad-Relación”
 - para poder modificar el diseño.
-

- N.º 7
 - Como usuario quiero tener la opción de cargar un diagrama a partir de un archivo
 - para hacer modificación de dicho diagrama y guardarlo de ser necesario.
 - Criterios de aceptación:
 - el usuario tendrá un botón “cargar” en el menú del diagramador ER para poder importar un archivo con extensión .json.
 - al importar el archivo, este pasará por un proceso de validación para asegurarse que es un archivo .json válido.
 - durante el proceso de validación, se verificará que el contenido del archivo es un diagrama compatible con la estructura de los generados por el diagramador ER.
 - al contener información compatible, se mostrará en la zona de diagramado el contenido del archivo.
-

- N.º 8
 - Como usuario quiero poder descargar el diagrama que esté visible en la página web
 - para poder distribuirlo como yo desee.
 - Criterios de aceptación:
 - el usuario dispondrá de un botón “Descargar” en el diagramador ER para obtener un archivo con el contenido del diagrama visible en la zona de diagramado.
 - el archivo generado será de extensión .json con la información necesaria para que el diagramador pueda cargarlo en otro momento.
-

- N.º 9
- Como usuario quiero tener una forma de validar mi diagrama ER

-
- porque es importante saber si el diagrama que estoy creando es un diagrama válido del modelo ER.
 - Criterios de aceptación:
 - el usuario tendrá disponible un botón que al darle clic iniciará un proceso de validación del diagrama actual en la zona de diagramado.
 - al término del proceso de validación, se le mostrará un mensaje al usuario indicando si el diagrama cumple o no las reglas del modelo ER.
-

- N.º 10
 - Como usuario, en caso de tener un diagrama ER válido,
 - me gustaría poder transformar el diagrama ER en su versión del modelo relacional
 - para poder ver el equivalente del diagrama ER en el modelo Relacional.
 - Criterios de aceptación:
 - el usuario dispondrá de la opción de transformar al modelo relacional solamente después de haber validado que el diagrama ER cumple con las reglas.
 - posterior a la validación, se le mostrará al usuario un mensaje de confirmación y un botón para disparar el proceso de transformación a su equivalente relacional.
 - al terminar el proceso de transformación equivalente, se le redirijirá al menú “Relacional” donde podrá visualizar el equivalente al modelo relacional.
-

- N.º 11
- Como usuario, después de observar el diagrama relacional,
- quiero poder obtener las sentencias SQL equivalentes
- para poder crear el esquema de base de datos relacional en un SBGB.
- Criterios de aceptación:
 - las sentencias SQL solo podrán ser descargadas en el menú “Relacional” a un archivo con extension .sql dando clic a un botón con la leyenda “Descargar SQL”.
 - solo se obtendrán las sentencias SQL de un diagrama ER creado y/o validado por el sistema.

-
- N.º 13
 - Como usuario quiero poder transformar mi diagrama ER en su equivalente modelo concepcual NoSQL
 - para poder observar el cambio entre modelos.
 - Criterios de aceptación:
 - el usuario dispondrá de la opción de tranformar al modelo NoSQL solamente después de haber validado que el diagrama ER cumple con las reglas.
 - posterior a la validación, se le mostrará al usuario un mensaje de confirmación y un botón para disparar el proceso de transformación al modelo conceptual NoSQL.
 - una vez validado el diagrama ER e iniciado el proceso para la transformación al modelo conceptual NoSQL, se le indicará al usuario que el proceso tardará un tiempo.
 - al término del proceso de transformación, se le redirigirá al menú “No Relacional” donde observará el modelo conceptual NoSQL equivalente a su diagrama ER.
-

- N.º 14
 - Como usuario quiero poder obtener el *script* desde el modelo conceptual NoSQL
 - para poder generar la base de datos en un gestor de base de datos NoSQL orientado a documentos.
 - Criterios de aceptación:
 - el usuario dispondrá de la opción de obtener el *script* solamente después de haber validado que el diagrama ER cumple con las reglas.
 - Posterior a la validación, se le mostrará al usuario un mensaje de confirmación y un botón para disparar el proceso de generación de *scripts* para el gestor de base de datos orientado a documentos.
 - una vez empezado el proceso de generación de *scripts*, se le indicará al usuario que el proceso tardará un tiempo.
 - al término del proceso de transformación, se le redirigirá al menú “No Relacional” donde observará los *scripts* NoSQL.
-

- N.º 15

- Como usuario me gustaría tener un reporte técnico y
- quiero que la redacción sea legible y referenciada
- para compartirlo en el futuro con equipos de desarrollo y ver la posibilidad de agregar nuevas funciones al sistema.

Teniendo en cuenta que se está trabajando con una metodología ágil, estas historias de usuario pueden aumentar o en su defecto dividirse en historias más pequeñas dependiendo de los criterios del equipo de desarrollo durante el proceso de la implementación de cada historia.

4.1.1.2. Lista de producto (*product backlog*)

De acuerdo a Trigas Gallego[47], la lista de producto es una lista ordenada de todo lo que sería necesario en el producto y es la fuente de requisitos para cualquier cambio a realizarse en el mismo. Enumera todas las características, funcionalidades, requisitos, mejoras y correcciones que constituyen cambios a realizarse en el producto para entregas futuras.

Se considera la siguiente lista como la lista de producto con las tareas necesarias para cumplir con todas las historias de usuario mencionadas en la sección anterior, considerando que es posible que cambien conforme avancen los *sprints* y así añadir nuevas tareas.

N.º de Historia de Usuario	Requerimiento/Tarea	Responsable
14	Investigación de bases de datos relacionales.	Eduardo/Omar
14	Redacción y selección de las tecnologías a utilizar para el desarrollo de la plataforma.	Eduardo
14	Investigación de bases de datos relacionales.	Eduardo/Omar
14	Redacción de bases de datos relacionales en el documento técnico.	Eduardo
14	Investigación de bases de datos no relacionales.	Eduardo/Omar
14	Redacción de bases de datos no relacionales en el documento técnico.	Eduardo
14	Investigación y selección del modelo de base de datos no relacional a utilizar junto a las tecnologías a utilizar.	Eduardo/Omar
14	Análisis y diseño de la arquitectura web.	Eduardo/Omar
1	Desarrollo de la estructura básica del <i>backend</i> .	Omar
1	Desarrollo de la estructura básica del <i>frontend</i> .	Eduardo

Continuación de tabla de lista de producto

1	Agregar servicio <i>backend</i> para registrar un usuario.	Omar
1	Agregar formulario para captura de datos de registro de un usuario en el <i>frontend</i> .	Eduardo
2	Agregar servicio <i>backend</i> para recuperar contraseña del usuario.	Omar
2	Agregar servicio <i>backend</i> para envío de correo al usuario registrado y de recuperación de contraseña.	Omar
2	Agregar vista con formulario para recuperación de contraseña del usuario en el <i>frontend</i> .	Eduardo
2	Integración de los servicios de registro y recuperación de contraseña en el <i>frontend</i> .	Eduardo
3	Agregar servicio <i>backend</i> para hacer ilegible la contraseña del usuario en la base de datos.	Omar
4	Planteamiento de escenarios de los esquemas entidad-relación.	Eduardo
4	Agregar a la interfaz gráfica de la aplicación web el menú “Entidad-Relación”.	Eduardo
4	Agregar íconos de los elementos básicos de un diagrama ER en el diagramador.	Eduardo
5	Agregar servicio <i>backend</i> para guardar un diagrama ER en formato JSON en la base de datos e integrarlo al <i>frontend</i> .	Omar
5	Agregar servicio <i>backend</i> para recuperar el diagrama guardado del usuario de la base de datos y regresarlo en formato JSON.	Omar
6	Recuperar el último diagrama del usuario del <i>backend</i> y mostrarlo en el <i>frontend</i> .	Eduardo
6	Manejar el estado de la interfaz web para no perder el diagrama ER que está editando el usuario.	Eduardo
6	Definición de las reglas del modelo entidad-relación.	Eduardo
4	Implementar la edición de diagramas ER en el <i>frontend</i> .	Eduardo
7	Habilitar la carga de un archivo en la aplicación web.	Omar
7	Agregar la función para validar el contenido del archivo .json y pintarlo en la zona de diagramado.	Eduardo
8	Agregar la descarga del diagrama visible en la zona de diagramado a un archivo .json.	Eduardo
9	Agregar botón de validar al <i>frontend</i> y mostrar el <i>loader</i> mientras se procesa el diagrama ER.	Eduardo/Omar
9	Agregar servicio <i>backend</i> para la validación del diagrama entidad-relación.	Eduardo/Omar
9	implementación de algoritmo para validación del diagrama ER en el <i>backend</i> .	Eduardo/Omar

Continuación de tabla de lista de producto

9	Pruebas de captura de distintos diagramas entidad-relación.	Eduardo/Omar
9	Pruebas para validar el algoritmo de validación.	Eduardo/Omar
10	Agregar servicio al <i>backend</i> para transformación del esquema entidad-relación al modelo relacional.	Omar
10	Implementación del algoritmo de transformación ER -> relacional	Eduardo/Omar
10	Agregar menú relacional a la interfaz gráfica.	Eduardo/Omar
10	Prueba de transformación de distintos diagramas ER al modelo relacional.	Eduardo/Omar
10	Visualización de la transformación del modelo ER al modelo relacional.	Eduardo
14	Revisión de la redacción del reporte técnico para presentación de TT1	Eduardo
11	Agregar servicio <i>backend</i> para la descarga del archivo .sql con las sentencias equivalentes.	Eduardo/Omar
11	Pruebas de coherencia de las sentencias equivalentes de distintos en el SGBD.	Eduardo/Omar
12	Definición de las reglas de transformación al modelo NoSQL.	Eduardo/Omar
12	Pruebas de distintos escenarios del modelo relacional al modelo NoSQL.	Eduardo/Omar
12	Agregar servicio al <i>backend</i> para transformación del esquema relacional al modelo NoSQL.	Omar
12	Agregar servicio al <i>backend</i> para guardar el modelo NoSQL en la base de datos.	Omar
12	Agregar menú no relacional a la interfaz gráfica.	Omar
12	Implementación del algoritmo de transformación de modelo relacional al modelo conceptual NoSQL.	Eduardo
12	Comprobación de la coherencia de la transformación entre modelos relacional a no relacional.	Eduardo/Omar
13	Agregar servicio <i>backend</i> para transformación del modelo ER al modelo no relacional.	Eduardo/Omar
13	Ajustar la interfaz del menú ER para mostrar mensaje de transformación al modelo NoSQL.	Omar
13	Manejar el estado del diagrama ER y redireccionar al menú no relacional al terminar la transformación.	Eduardo
13	Pruebas de caso de estudio para verificar la correcta transformación y coherencia de los datos.	Eduardo/Omar
14	Revisión de la redacción del reporte técnico para presentación de TT2	Eduardo

Tabla 4.1: Lista de producto

4.2. Análisis de factibilidad

Para todo proyecto es necesario realizar un estudio de factibilidad el cual consta de 3 partes. El objetivo de esta sección es mostrar la factibilidad técnica, económica y operativa para mostrar la sustentabilidad esquematizando los costos para su desarrollo.

4.2.1. Factibilidad técnica

Mediante esta fase del estudio se determinará si el equipo de desarrollo cuenta con los recursos técnicos necesarios para la realización del sistema propuesto. Esto se realiza considerando la disponibilidad de los recursos tanto de *hardware*, *software* y recurso humano.

4.2.1.1. Sistema operativo

Este es un elemento importante, ya que debe cumplir con las características de estabilidad, velocidad, seguridad y escalabilidad para soportar la instalación del sistema.

A continuación se presentan diferentes alternativas de sistemas operativos que cumplen con las características mencionadas y que son suficientes para albergar el sistema:

- Windows server 2019
- Red Hat Enterprise 8
- Ubuntu server 19.04

4.2.1.2. Lenguaje de desarrollo

El lenguaje de desarrollo debe de cumplir con las siguientes características:

- Soporte para conexión a base de datos.
- Facilidad para el desarrollo.
- En continua mejora.
- Fácil de administrar.
- Contar con algún *framework* web.

A continuación se presenta una lista de lenguajes de desarrollo que cumplen dichas características:

- Java

Equipo	Elemento	Capacidad
Laptop 1	Memoria RAM	8 GB
	Almacenamiento	500 GB HDD
	Procesador	Intel Core i5 6ta gen.
Laptop 2	Memoria RAM	8 GB
	Almacenamiento	256 GB SSD
	Procesador	Intel Core i5 8va gen.

Tabla 4.2: Equipo de cómputo

- Python
- C#
- Ruby

4.2.1.3. Sistema gestor de base de datos

Este es un factor muy importante, ya que determinará cómo se almacenará la información del sistema, por lo tanto debe cumplir con las siguientes características:

- Escalable.
- Seguro.
- Contar con soporte para grandes cantidades de información.
- Contar con soporte para conexión con distintos lenguajes de programación.

A continuación se presenta una lista de sistemas gestores de bases de datos que cumplen dichas características:

- MySQL
- MariaDB
- Oracle database
- MongoDB Atlas
- DynamoDB
- Apache Cassandra

Las características de los equipos de cómputo con los que se dispone actualmente para el desarrollo del sistema se muestran en la tabla [4.2](#).

Con los datos anteriormente mencionados, se concluye que la tecnología para el desarrollo del sistema existe y se cuenta con los recursos de *hardware* suficientes para iniciar con su implementación.

Parámetro	Cuenta	Factores de ponderación			Total
		baja	media	alta	
Entradas	6	3	4	6	36
Salidas	5	4	5	7	25
Tablas	1	3	4	6	4
Interfaces	4	7	10	15	40
Consultas	4	5	7	10	28
Conteo total					133

Tabla 4.3: Cálculo de las métricas por puntos de función

4.2.2. Factibilidad económica

De acuerdo con Pressman en *Ingeniería de software, un enfoque práctico*[48], en su sección de estimación, se utiliza una métrica por puntos de función como se muestra en la tabla 4.3.

Es por este medio que se calcula o se realiza una estimación del costo total del proyecto, incluyendo los salarios de los desarrolladores que llevarán a cabo la implementación del sistema, así como los gastos por pagos de servicios que sean necesarios.

4.2.2.1. Métricas orientadas a la función

Para este proyecto, se considera que todas las funciones identificadas son de complejidad media con excepción de las entradas que tienen la complejidad más alta del sistema.

Fi ($i = 1..14$) son factores de ajuste de valor basados en las respuestas de las preguntas de la tabla 4.4. Los valores pueden ir de 0 (no importante o aplicable) a 5 (absolutamente esencial).

4.2.2.2. Puntos de función

La fórmula para obtener los puntos de función con los factores de ajuste es la siguiente:

$$PF = \text{conteo total} * (0.65 + (0.01 * \sum Fi))$$

Se deben sustituir los valores del conteo total y los factores de ajuste.

$$PF = 133 * (0.65 + (0.01 * 54)) PF = 158.27 \approx 159 \quad (4.1)$$

De lo anterior, aproximadamente se obtienen **159** puntos de función. Una vez obtenidos utilizando la llamada “Ball-Park” o “Estimación Indicativa”, que es la técnica de macro-estimación que se utiliza en situaciones de falta de información sobre el proyecto. El autor del artículo *Applied Software Measurement*[49], Carper

Pregunta	Ponderación
¿Requiere el sistema métodos de seguridad y recuperación fiables?	3
¿Se requiere comunicación especializada?	5
¿Existen funciones de procesamiento distribuido?	2
¿Es crítico el rendimiento?	4
¿Se ejecutará el sistema en un entorno operativo existente y fuertemente utilizado?	4
¿Requiere el sistema una entrada de datos interactiva?	5
¿Requiere la entrada de datos interactiva que las transacciones de entrada se lleven a cabo sobre múltiples operaciones?	5
¿Se actualizan los archivos maestros de forma interactiva?	3
¿Son complejas las entradas, salidas, archivos o consultas?	4
¿Es complejo el procesamiento interno?	4
¿Se ha diseñado el código para ser reutilizable?	4
¿Están incluidas en el diseño la instalación y conversión?	3
¿Se ha diseñado el sistema para soportar múltiples instalaciones en diferentes organizaciones?	4
¿Se ha diseñado el sistema para facilitar los cambios y para ser fácilmente utilizable?	4
$\sum Fi =$	54

Tabla 4.4: Factores de ajuste

Jones, propone la siguiente ecuación para determinar el esfuerzo de desarrollo de un proyecto:

$$Esfuerzo = \left(\frac{PF}{150}\right) * PF^{0.4}$$

Donde:

PF : Puntos de función.

Al sustituir los valores, el resultado es:

$$Esfuerzo = \left(\frac{159}{150}\right) * 159^{0.4} Esfuerzo = 8.05 meses$$

De estos 8.05 meses, considerando un total de 40 horas a la semana de trabajo y 4.34 semanas por mes, el total de horas para el desarrollo y conclusión del proyecto se obtiene de esta manera:

$$\text{Tiempo de desarrollo} = 40 * 4.34 * 8.05 \quad \text{Tiempo de desarrollo} = 1397.48 \approx 1398 \text{ horas}$$

Por ejemplo, una sola persona trabajando en el desarrollo del proyecto debería invertir 1398 horas con una jornada de 8 horas diarias de lunes a viernes hasta su finalización, por lo que si un equipo de desarrollo es de 2 personas con un horario de lunes a viernes de 4 horas diarias, el proyecto concluiría en 8.05 meses.

4.2.3. Costos de desarrollo

De acuerdo con *Software Guru*[50], en una publicación que recopila los datos de salarios en el área de desarrollo de *software* para febrero de 2020, un desarrollador con 0 a 2 años de experiencia, como es el caso de un estudiante, tiene un salario de \$ 15 000 mensuales en una jornada completa, considerando que este proyecto contempla jornadas de medio tiempo (4 horas) de lunes a viernes, se reduce la cifra antes mencionada. Teniendo en cuenta estos datos y un periodo de 9 meses, que es el tiempo aproximado de duración del proyecto, el costo total por salarios para el equipo de desarrollo está desglosado en la tabla 4.5.

Concepto	Costo Aprox. Semanal	Costo Aprox. Mensual	Monto total
Salario	1850	7500	67500

Tabla 4.5: Costos del personal

Esto da como resultado un total final de \$ 135 000 por los salarios de los 2 integrantes del equipo de desarrollo. Se tomaron en cuenta 9 meses para todos los gastos, un mes extra a lo obtenido en estimación para utilizarse en el caso de estudio del sistema una vez concluido.

Los gastos por pagos de licencia de *software* quedan excluidos, ya que las tecnologías seleccionadas son libres o gratuitas, lo cual no supone un costo para su uso. De igual manera, esto se encuentra simplificado en la tabla 4.6.

Software	Licencia	Costo
Visual Studio Code	MIT	0
Gunicorn(flask Server)	MIT	0
MongoDB Atlas	Apache v2	0

Tabla 4.6: Costos por licencias de software

Concepto	Costo Mensual	Monto total
Luz	250	2250
Internet	349	3141
Heroku hosting	0 (free plan)	0
Netlify hosting	0 (free plan)	0

Tabla 4.7: Costos por servicios

Otros gastos necesarios son los pagos por servicios requeridos listados en la tabla 4.7.

Habiendo realizado la suma de todas las cantidades antes mencionadas, el total final es:

$$\text{salarios} = 135\,000.00 \text{ servicios} = (2250 + 3141) * 9 = 48\,519.00$$

$$\text{Gastos totales} = 135\,000.00 + 48\,519.00 = 183\,519.00 \text{ pesos mexicanos.}$$

Factibilidad operativa

La factibilidad operativa permite predecir de cierta forma si es posible poner en marcha el sistema propuesto, aprovechando todos los beneficios que se ofrecen a todos los usuarios involucrados en ello. La herramienta va dirigida a los estudiantes que se encuentren en un primer acercamiento a los modelos de bases de datos entidad-relación o relacional desde un enfoque conceptual, buscando principalmente mostrarles una aproximación a los modelos no relacionales de bases de datos. El sistema propuesto cuenta con una interfaz intuitiva para que el usuario final, los estudiantes, puedan visualizar, crear y editar un diagrama ER y las opciones que esta les brinde de manera comprensible.

Teniendo en cuenta los motivos anteriormente explicados, el sistema propuesto tiene una alta probabilidad de aceptación por parte de los usuarios finales al encontrarse en un entorno en el que se trabaja con *software* continuamente, además del beneficio que aporta al plan de estudios actual al ofrecer una forma práctica de ver aplicado los conceptos adquiridos en el curso de base de datos, el cual solo contempla un alcance hasta la normalización de bases de datos relacionales y tener una introducción a los modelos no relacionales (noSQL). Un estudiante que ha cursado dicha asignatura se dará cuenta que el tiempo disponible durante el curso es limitado por la cantidad de módulos que pretende cubrir y en muchas ocasiones los docentes deben prescindir de ciertos temas para completar el temario.

Con la implantación de la aplicación web que se está proponiendo, los

estudiantes que cursen la asignatura de de base de datos tendrán la oportunidad de conocer una opción más en cuanto a tecnologías de almacenamiento de datos para implementar en sus propios sistemas. De igual manera, puede impulsarlos a generar propuestas para la apertura de una asignatura optativa si el interes por estos modelos de datos resulta interesante para ellos.

Teniendo en cuenta los puntos mencionados anteriormente, se concluye que el sistema propuesto tendrá un uso en la institución y un potencial beneficio para los estudiantes y los involucrados en ello.

Capítulo 5

Algoritmos

5.1. Validación estructural diagrama entidad-relación

De acuerdo con Dullea[7], un modelo ER está compuesto por entidades, las relaciones entre entidades y restricciones en esas relaciones.

Las entidades pueden estar encadenadas en una serie de entidades y relaciones alternas, o pueden participar singularmente con una o más relaciones.

La conectividad de entidades y relaciones se denomina ruta. Las rutas son los bloques de construcción de nuestro estudio en el análisis de validez estructural.

Las rutas definen visualmente la asociación semántica y estructural que cada entidad tiene simultáneamente con todas las demás entidades o consigo misma dentro de la ruta.

Los términos, validez estructural y semántica, se definen de la siguiente manera.

Definition 5.1.1. Un diagrama de entidad-relación es estructuralmente válido solo cuando la consideración simultánea de todas las restricciones estructurales impuestas en el modelo no implica una inconsistencia lógica en ninguno de los posibles estados. Un diagrama de entidad-relación es semánticamente válido solo cuando cada relación representa exactamente el concepto del modelador del dominio del problema. Un diagrama de entidad-relación es válido cuando es válido semántica y estructuralmente.

En el modelado de datos, la validez se puede clasificar en dos tipos: validez semántica y estructural. Un ERD semánticamente válido muestra la representación correcta del dominio de aplicación que se está modelando.

El diagrama debe comunicar exactamente el concepto previsto del entorno tal como lo ve el modelador. Dado que la validez semántica depende de la aplicación, no podemos definir criterios de validez generalizados.

Por lo tanto, no se considerará la validez semántica.

La validez estructural de un ERD se refiere a si un ERD dado contiene o no construcciones que son contradictorias entre sí. Un ERD con al menos una restricción

de cardinalidad inconsistente es estructuralmente inválido.

Un ERD representa la semántica de la aplicación en términos de restricciones de cardinalidad máxima y mínima. La fuerza impulsora detrás de la colocación de la restricción de cardinalidad es la semántica del modelo.

Cada conjunto de restricciones de cardinalidad en una sola relación debe ser coherente con todas las restricciones restantes en el modelo y en todos los estados posibles

Una relación recursiva, es decir, la asociación entre grupos de roles dentro de una sola entidad, se evalúa como un objeto conceptual independiente.

Es semánticamente inválido cuando el concepto no refleja las reglas de negocio definidas por la comunidad de usuarios.

Una relación recursiva es estructuralmente inválida cuando las restricciones de participación y cardinalidad no respaldan la existencia de instancias de datos como lo requiere el usuario y hace que todo el diagrama sea inválido.

En general, un diagrama estructuralmente inválido refleja reglas de negocio semánticamente inconsistentes. Para que un modelo sea válido, todas las rutas del modelo también deben ser válidas.

Relaciones recursivas

Una relación recursiva se define como una asociación entre instancias a medida que asumen roles dentro de la misma entidad. Los roles juegan un papel importante en el examen de la validez estructural, especialmente para las relaciones recursivas.

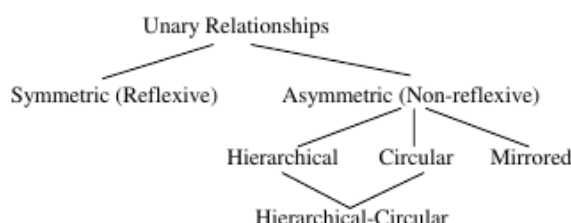


Figura 5.1: Taxonomía de relaciones recursivas

Un rol es la acción o función que desempeñan las instancias de una entidad en una relación. En una relación recursiva, un conjunto de instancias puede asumir un solo rol o múltiples roles dentro de la misma relación.

Examinar estos roles permite clasificar todas las relaciones recursivas en asociaciones simétricas o asimétricas, mientras que clasificamos aún más los tipos de relaciones asimétricas en asociaciones jerárquicas, circulares y duplicadas.

La clasificación completa de las relaciones recursivas que consideramos en este artículo se muestra en la figura 5.1 como sigue.

Una relación recursiva es simétrica o reflexiva cuando todas las instancias que participan en la relación toman un solo papel y el significado semántico de la

Tipo de relación	Dirección de la relación		Restricción de participación	Restricciones de cardinalidad	Ejemplo	
					Relación	Roles
Simétrica (reflexiva)	Bidireccional		Opcional-opcional Obligatoria-obligatoria	1-1 M-N	Cónyuge de	Persona
Asimétrica (no reflexiva)	Jerárquica	Unidireccional	Opcional-opcional	1-M 1-1	Supervisa Es supervisado por	Gerente- empleado
			Opcional-opcional Opcional-obligatoria Obligatoria-obligatoria	M-N	Supervisa Es supervisado por	Gerente Empleado
			Opcional-opcional Obligatoria-obligatoria	1-1	Apoya Es apoyado por	Servicio técnico
	Jerárquica Circular	Unidireccional	Opcional-opcional Opcional-obligatoria	1-M	Apoya Es apoyado por	Responsable
			Opcional-opcional Opcional-obligatoria Obligatoria-obligatoria	M-N	Supervisa	Gerente- empleado
	Reflejada	Unidireccional	Opcional-opcional	1-1	Gestiona Se gestiona	CEO

Tabla 5.1: Tipos de relación recursiva válidos según las restricciones de cardinalidad

relación es exactamente el mismo para todas las instancias que participan en la relación independientemente de la dirección en la que se ve. Estos tipos de relación se denominan bidireccionales.

Una relación recursiva es asimétrica o no reflexiva cuando hay una asociación entre dos grupos de roles diferentes dentro de la misma entidad y el significado semántico de la relación es diferente dependiendo de la dirección en la que se ven las asociaciones entre los grupos de roles. Estos tipos de relación se denominan unidireccionales.

Una relación recursiva es jerárquica cuando un grupo de instancias dentro de la misma entidad se clasifican en calificaciones, órdenes o clases, una encima de otra. Implica un comienzo (o arriba) y un final (o abajo) para el esquema de clasificación de instancias.

Una relación recursiva es circular cuando una relación recursiva asimétrica tiene al menos una instancia que no cumple con la jerarquía de clasificación. La relación es unidireccional, ya que se puede ver desde dos direcciones con un significado semántico diferente.

Otra pregunta que surge en el modelado de las relaciones recursivas es si una instancia puede asociarse consigo mismo.

Este evento es imposible en relaciones superiores al primer grado, pero podría ocurrir en casos especiales de una relación recursiva y llamamos a este evento especial una relación reflejada.

Una relación reflejada existe cuando la semántica de una relación permite que una instancia de una entidad se asocie a sí misma a través de la relación.

La tabla 5.1 resume cada relación recursiva por sus propiedades direccionales, la combinación de restricciones de cardinalidad mínima y máxima, y ejemplos. En nuestros diagramas a lo largo de este En el documento, utilizamos la notación uno (1) y muchos (M) para obtener la máxima cardinalidad, una sola línea para indicar la participación opcional y una línea doble para mostrar la participación obligatoria. Las palabras “obligatorio” y “opcional” se utilizan en la tabla para

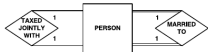
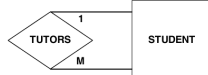
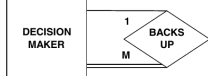
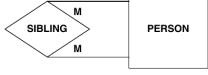

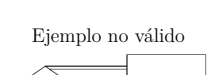

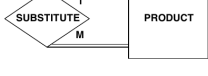
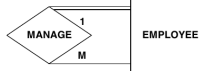
Reglas de validación para relaciones recursivas	Ejemplo válido
Solo las relaciones recursivas 1:1 con restricciones de cardinalidad mínimas obligatorias-obligatorias u opcionales son estructuralmente válidas. Válido para relaciones simétricas y completamente circular.	 
Para las relaciones recursivas 1:M o M:1, la cardinalidad mínima opcional-opcional es estructuralmente válida. Válido solo para relaciones asimétricas.	
Para 1:M las relaciones recursivas del tipo jerárquico-circular, la cardinalidad mínima opcional-obligatoria son estructuralmente válidas. Válido solo para relaciones jerárquico-circulares.	
Todas las relaciones recursivas con cardinalidad máxima de muchos a muchos son estructuralmente válidas independientemente de las restricciones mínimas de cardinalidad. Válido para relaciones simétricas, jerárquicas y jerárquicas-circulares.	
Todas las relaciones recursivas con cardinalidad mínima opcional-opcional son estructuralmente válidas. Válido para relaciones simétricas y asimétricas.	
Colorarios de validez para relaciones recursivas	Ejemplo no válido
Todas las relaciones recursivas 1: 1 con restricciones de cardinalidad mínima obligatoria-opcional u opcional-obligatoria son estructuralmente inválidas.	
Todas las relaciones recursivas 1:M o M:1 con restricciones de cardinalidad mínimas obligatorias-obligatorias son estructuralmente inválidas.	
Todas las relaciones recursivas 1:M o M:1 con restricción de participación obligatoria en "uno" y una restricción de participación opcional en las "muchas" restricciones son estructuralmente inválidas.	

Tabla 5.2: Resumen de reglas de validez para relaciones recursivas con ejemplos

indicar la cardinalidad mínima obligatoria (o total) y opcional (o parcial), respectivamente. Además, la notación $|E|$ representa el número de instancias en la entidad E .

A continuación se pone un resumen de las reglas válidas para relaciones unarias y binarias:

Generales

1. No puede haber elementos sin conectar.
2. Tampoco puede haber enlaces sin conectar.
3. Solo relación de participación binarias ?????

Entidad

1. Una entidad es válida si tiene atributos, porque no tiene propósito una entidad sin atributos.
2. La clave primaria puede ser simple o compuesta.
3. La clave primaria no es una clave foranea.

Reglas de validez para relaciones binarias.	Ejemplo válido
Una ruta cíclica que contiene todas las relaciones binarias siempre es estructuralmente válida.	
Una ruta cíclica que contiene todas las relaciones binarias y una o más relaciones opcional-opcional siempre es estructuralmente válida.	
Una ruta cíclica que contiene todas las relaciones binarias y una o más relaciones de muchos a uno con participación opcional del lado "uno" siempre es estructuralmente válida.	
Una ruta cíclica que contiene todas las relaciones binarias y una o más relaciones de muchos a muchos es siempre estructuralmente válida.	
Las rutas cíclicas que contienen al menos un conjunto de relaciones opuestas siempre son válidas.	
Una ruta cíclica que contiene todas las relaciones binarias uno a uno que son todas obligatorias-obligatorias o al menos una restricción de cardinalidad mínima opcional-opcional siempre es estructuralmente válida.	
Validez corolarios para relaciones binarias	Ejemplo no válido
Las rutas cíclicas que no contienen relaciones opuestas ni relaciones autoajustables son estructuralmente inválidas y se denominan relaciones circulares.	
La presencia de una relación "uno a uno obligatorio-obligatorio" no tiene ningún efecto sobre la validez estructural (o invalidez) de una ruta cíclica que contiene otros tipos de relación. (Este corolario se aplica a todas las reglas anteriores).	

Tabla 5.3: Resumen de reglas de validez para relaciones binarias con ejemplos

-
4. La clave primaria debe ser un atributo clave asociado a la entidad. (restricción del proyecto, para facilidad)
 5. Dos entidades solo pueden estar conectadas entre sí mediante una relación.
 6. Todas las entidades deben tener un atributo clave

Entidad débil

1. Un atributo solo puede estar asociado a un solo atributo o a una sola entidad.
2. Una entidad débil no puede existir si no tiene una relación con otra entidad.

Atributo

1. Un atributo solo puede estar asociado a un solo atributo o a una sola entidad.
2. Un atributo puede ser compuesto.
3. Un atributo puede ser multivalor.
4. Un atributo puede ser derivado.
5. Un atributo debe tener un nombre.
6. Un atributo no puede usar una relación para asociarse a otro elemento.
7. Un atributo compuesto solo puede estar asociado a una entidad.
8. Un atributo derivado solo puede estar asociado a una entidad.

Relación

1. Una relación solo puede ser entre entidades.
2. El grado de participación máximo es dos. (restricción del proyecto)
3. Una relación puede ser unaria (recursiva).
4. No están permitidas relaciones ternarias o de grado n (restricción del proyecto)

5.2. Modelo entidad-relación a relacional

Elmasri[14] propone un algoritmo en siete pasos para convertir las estructuras de un modelo ER básico con tipos de entidades fuertes y débiles, relaciones binarias con distintas restricciones estructurales, relaciones n-ary y atributos (simples, compuestos y multivalor) en relaciones.

Paso 1: Mapeado de los tipos de entidad regulares

Por cada entidad (fuerte) regular E del esquema ER cree una relación R que incluya todos los atributos simples de E .

Incluya únicamente los atributos simples que conforman un atributo compuesto; seleccione uno de los atributos clave de E como clave principal para R .

Si la clave elegida de E es compuesta, el conjunto de los atributos simples que la forman constituirán la clave principal de R .

Si durante el diseño conceptual se identificaron varias claves para E , la información que describe los atributos que forman cada clave adicional conserva su orden para especificar las claves (únicas) secundarias de la relación R .

El conocimiento sobre las claves también es necesario para la indexación y otros tipos de análisis.

Paso 2: Mapeado de los tipos de entidad débiles

Por cada tipo de entidad débil W del esquema ER con el tipo de entidad propietario E , cree una relación R e incluya todos los atributos simples (o componentes simples de los atributos compuestos) de W como atributos de R .

Además, incluya como atributos de la *foreign key* de R , los atributos de la o las relaciones que correspondan al o los tipos de entidad propietarios; esto se encarga de identificar el tipo de relación de W .

La clave principal de R es la combinación de las claves principales del o de los propietarios y la clave parcial del tipo de entidad débil W si la hubiera.

Si hay un tipo de entidad débil E_2 cuyo propietario también es un tipo de entidad débil E_1 , E_1 debe asignarse antes que E_2 para determinar primero su clave principal.

Paso 3: Mapeado de los tipos de relación 1:1 binaria

Por cada tipo de relación 1:1 binaria R del esquema ER, identifique las relaciones S y T que corresponden a los tipos de entidad que participan en R . Hay tres metodologías posibles: (1) la metodología de la *foreign key*, (2) la metodología de la relación mezclada y (3) la metodología de referencia cruzada o relación de relación. La primera metodología es la más útil y la que debe seguirse salvo que se den ciertas condiciones especiales, como las que explicamos a continuación:

- 1 Metodología de la *foreign key*:** seleccione una de las relaciones (por ejemplo, S) e incluya como *foreign key* en S la clave principal de T . Lo mejor es elegir un tipo de entidad con participación total en R en el papel de S . Incluya todos los atributos simples (o los componentes simples de los atributos compuestos) del tipo de relación 1:1 R como atributos de S .
- 2 Metodología de la relación mezclada:** una asignación alternativa de un tipo de relación 1:1 es posible al mezclar los dos tipos de entidad y la relación

en una sola relación. Esto puede ser apropiado cuando las dos participaciones son totales.

3 Metodología de referencia cruzada o relación de relación: consiste en configurar una tercera relación R con el propósito de crear una referencia cruzada de las claves principales de las relaciones S y T que representan los tipos de entidad. Como veremos, esta metodología es necesaria para las relaciones $M : N$ binarias. La relación R se denomina relación de relación (y, en algunas ocasiones, tabla de búsqueda), porque cada tupla de R representa una instancia de relación que relaciona una tupla de S con otra de T .

Paso 4: Mapeado de tipos de relaciones 1:N binarias

Por cada relación $1 : N$ binaria regular R , identifique la relación S que representa el tipo de entidad participante en el lado N del tipo de relación.

Incluya como *foreign key* en S la clave principal de la relación T que representa el otro tipo de entidad participante en R ; hacemos esto porque cada instancia de entidad en el lado N está relacionada, a lo sumo, con una instancia de entidad del lado 1 del tipo de relación.

Incluya cualesquiera atributos simples (o componentes simples de los atributos compuestos) del tipo de relación $1 : N$ como atributos de S .

De nuevo, una metodología alternativa es la opción de relación de relación (referencia cruzada), como en el caso de las relaciones 1:1 binarias.

Creamos una relación R separada cuyos atributos son las claves de S y T , y cuya clave principal es la misma que la clave de S . Esta opción puede utilizarse si pocas tuplas de S participan en la relación para evitar excesivos valores NULL en la *foreign key*.

Paso 5: Mapeado de tipos de relaciones M:N binarias

Por cada tipo de relación $M : N$ binaria R cree una nueva relación S para representar a R .

Incluya como atributos de la *foreign key* en S las claves principales de las relaciones que representan los tipos de entidad participantes; su combinación formará la clave principal de S .

Incluya también cualesquiera atributos simples del tipo de relación $M : N$ (o los componentes simples de los atributos compuestos) como atributos de S .

No podemos representar un tipo de relación $M : N$ con un atributo de *foreign key* en una de las relaciones participantes (como hicimos para los tipos de relación $1 : 1$ o $1 : N$) debido a la razón de cardinalidad $M : N$; debemos crear una relación de relación S separada.

Siempre podemos asignar las relaciones $1 : 1$ o $1 : N$ de un modo similar a las relaciones $M : N$ utilizando la metodología de la referencia cruzada (relación de relación), como explicamos anteriormente.

Esta alternativa es particularmente útil cuando existen pocas instancias de relación, a fin de evitar valores NULL en las *foreign keys*.

En este caso, la clave principal de la relación de relación sólo será una de las *foreign keys* que hace referencia a las relaciones de entidad participantes.

Para una relación $1 : N$, la clave principal de la relación de relación será la *foreign key* que hace referencia a la relación de la entidad en el lado N .

En una relación $1 : 1$, cada *foreign key* se puede utilizar como la clave principal de la relación de relación, siempre y cuando no haya entradas NULL en la relación.

Paso 6: Mapeado de atributos multivalor

Por cada atributo multivalor A , cree una nueva relación R .

Esta relación incluirá un atributo correspondiente a A , más el atributo clave principal K (como *foreign key* en R) de la relación que representa el tipo de entidad o tipo de relación que tiene A como un atributo.

La clave principal de R es la combinación de A y K . Si el atributo multivalor es compuesto, incluimos sus componentes simples.

Paso 7: Mapeado de los tipos de relación n-ary

Por cada tipo de relación n-ary R , donde $n > 2$, cree una nueva relación S para representar R .

Incluya como atributos de la *foreign key* en S las claves principales de las relaciones que representan los tipos de entidad participantes.

Incluya también cualesquiera atributos simples del tipo de relación n-ary (o los componentes simples de los atributos compuestos) como atributos de S .

Normalmente, la clave principal de S es una combinación de todas las *foreign keys* que hacen referencia a las relaciones que representan los tipos de entidad participantes.

No obstante, si las restricciones de cardinalidad en cualquiera de los tipos de entidad E que participan en R es 1, entonces la clave principal de S no debe incluir el atributo de la *foreign key* que hace referencia a la relación E' correspondiente a E .

Correspondencia entre los modelos ER y relacional

Tabla 5.4: Correspondencia entre los modelos ER y relacional.

Modelo entidad-relación	Modelo relacional
Tipo de entidad	Relación de entidad
Tipo de relación $1 : 1$ o $1 : N$	<i>foreign key</i> (o relación de relación)
Tipo de relación $M : N$	Relación de relación y dos <i>foreign keys</i>

Tabla 5.4 – continued from previous page

Modelo entidad-relación	Modelo relacional
Tipo de relación n-ary	Relación de relación y <i>n foreign keys</i>
Atributo simple	Atributo
Atributo compuesto	Conjunto de atributos simples
Atributo multivalor	Relación y <i>foreign key</i>
Conjunto de valores	Dominio
Atributo clave	Clave principal (o secundaria)

5.3. Obtención de esquema SQL desde modelo relacional

5.4. Modelo entidad-relación a Generic Data Metamodel

El algoritmo necesita ser basado en queries como el de Chebotko.

input: modelo entidad-relacion

output: ¿modelo orientado a documentos o GDM?

¿Qué definición del modelo orientado a documentos se usará? ¿El del GDM o el de De Lima?

¿Cómo convertir las relaciones? Hay reglas para convertir las relaciones en bloques del documento orientado a objetos.

5.5. Generic Data Metamodel a modelo lógico NoSQL

Un *DocumentType* es la estructura principal que se guardará en la colección.

Sea Q el conjunto de todas las queries q_n en una instancia GDM:

$$Q = \{q_1, q_2, \dots, q_n\} \text{ donde } n = 1, 2, \dots$$

Sea E_f el conjunto de entidades formado al consultar el único elemento from de cada q_n ($q_n.from$ de Q):

$$E_f = \{e_{f1}, e_{f2}, \dots, e_{fn}\} \text{ donde } n = 1, 2, \dots$$

Sea C el conjunto de colecciones c_n donde cada c_n corresponde a un e_{fn} de E_f (es decir que por cada e_{fn} se crea un c_n en la que cada c_n contiene un único elemento *DocumentType*, llamado “documento raíz”):

$$C = \{c_1, c_2, \dots, c_n\} \text{ donde } n = 1, 2, \dots$$

Finalmente, por cada c_n se generan los contenidos de cada documento raíz:

Como cada colección c_n es creada a partir de una única entidad e_{fn} , se necesita buscar del conjunto Q todas las q_n que contengan esa e_{fn} como entidad de búsqueda principal (es decir, que e_{fn} esté en el elemento *from* de la q_n).

Dicho de otra manera, sea el conjunto Q_E el subconjunto propio de Q en el que cada q_{en} contiene en su elemento *from* la e_{fn} .

$$Q_E = \{q_{e1}, q_{e2}, \dots, q_{en}\} \text{ donde } Q_E \subseteq Q | q_{en}.from == e_{fn}$$

Por cada q_{en} se consulta todos sus elementos *including*, a cada elemento *including* le llamamos r_n para formar el conjunto R de referencias.

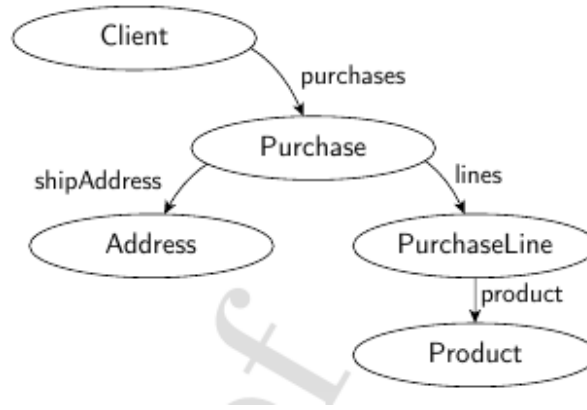


Figura 5.2: Access Tree - Modelo lógico orientado a documentos

```

query Q4_clientPurchasesNearChristmas:
select client.clientId, client.name,
       client.nationality,
       purchases.purchaseId, purchases.year,
       purchases.month, purchases.day,
       lines.quantity, lines.unitPrice,
       product.name, address.postalCode
from Client as client
including client.purchases as purchases,
         client.purchases.lines as lines,
         client.purchases.lines.product as product,
         client.purchases.shipAddress as address
where client.clientId = "71" and purchases.year = "72"
       and purchases.month >= 10
order by purchases.month, product.price

```

Figura 5.3: Access query Q4

Creamos un árbol de acceso con el conjunto R y con ese árbol de acceso generamos los documentos de cada documento raíz de cada c_n que tiene una única e_{fn} .

La figura 5.2 es un ejemplo del árbol de acceso de la consulta q_4 de la figura 5.3.

Añadimos todos los atributos simples de la e_{fn} .

Cada r_n lo añadimos al documento raíz de acuerdo a 1) la cardinalidad de la referencia y 2) la entidad objetivo de la referencia

Si la entidad destino está dentro del árbol de consultas implica que datos de esa entidad serán requeridos en la consulta, por lo que el nuevo *field* deberá ser incluido como un subdocumento o una colección de documentos, dependiendo de su cardinalidad.

En contraparte, si la entidad destino no está en el árbol de consultas, se podría quitar esa referencia, pero se agregará para mejorar la escalabilidad de la base de datos para futuras consultas. Esta referencia tendría el valor del identificador de la entidad referenciada cuando la cardinalidad es 1. Si la cardinalidad es n , será un arreglo de identificadores.

Cuando una referencia es transformada a un subdocumento se vuelve a generar su árbol de acceso en una llamada recursiva para generar sus contenidos.

Por último, el autor menciona dos optimizaciones si se quiere reducir el nivel de denormalización, las que se pueden consultar en su investigación[8].

En resumen, el algoritmo quedaría de la forma:

Algorithm 1: Transformación del modelo conceptual GDM al modelo lógico orientado a documentos

Input : una instancia del modelo GDM, gdm
Output: un modelo lógico orientado a documentos, ddm

```
1  $mainEntities \leftarrow gdm.queries.collect((q)|q.from);$   
2 foreach  $me \in mainEntities$  do  
3    $collection \leftarrow newCollection();$   
4    $collection.name \leftarrow me.name;$   
5    $accessTree \leftarrow allQueryPaths(me, gdm.queries);$   
6    $collection \leftarrow populateDocumentType(collection.root, accessTree);$   
7    $ddm.collections.add(collection);$   
8 end
```

Donde la función `populateDocumentType()` es otro algoritmo de la forma:

5.6. Modelo lógico NoSQL a modelo físico en MongoDB

Algorithm 2: Generar el contenido de un *DocumentType* dado un árbol de acceso

Input : Un “document type”, *dt*
Output: un nodo del arbol de acceso

```
1 nodeAttributes  $\leftarrow$  node.entity.features.select(f|f.isTypeOf(Attribute))
2 nodeReferences  $\leftarrow$ 
   node.entity.features.select(f|f.isTypeOf(Reference))
3 foreach attr  $\in$  nodeAttributes do
4   | pf  $\leftarrow$  newPrimitiveField()
5   | pf.name  $\leftarrow$  attr.name
6   | pf.type  $\leftarrow$  attr.type
7   | dt.fields.add(pf)
8 end
9 foreach ref  $\in$  nodeReferences do
10  | targetNode  $\leftarrow$  node.arcs.find(a|a.name = ref.name).target
11  | if exists(targetNode) then
12  |   | baseType  $\leftarrow$  newDocumentType()
13  |   | populateDocumentType(baseType, targetNode)
14  | else
15  |   | baseType  $\leftarrow$  newPrimitiveField()
16  |   | baseType.type  $\leftarrow$  findIdType(ref.entity)
17  | end
18  | baseType.name  $\leftarrow$  ref.name
19  | if ref.cardinality == 1 then
20  |   | dt.field.add(baseType)
21  | else
22  |   | arrayField  $\leftarrow$  newArrayField()
23  |   | arrayField.type  $\leftarrow$  baseType
24  |   | dt.fields.add(arrayField)
25  | end
26 end
```

Apéndice A

Apéndice

A.1. Unified Modeling Language

De acuerdo con Wikipedia[51], un diagrama de clases en Lenguaje Unificado de Modelado (UML) es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos.

Miembros

UML proporciona mecanismos para representar los miembros de la clase, como atributos y métodos, así como información adicional sobre ellos.

Visibilidad

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), se coloca uno de los siguientes signos delante de ese miembro:

1. + Público
2. - Privado
3. # Protegido
4. / Derivado (se puede combinar con otro)
5. Paquete

Ámbitos

UML especifica dos tipos de ámbitos para los miembros: instancias y clasificadores y estos últimos se representan con nombres subrayados.

-
1. Los miembros clasificadores se denotan comúnmente como “estáticos” en muchos lenguajes de programación. Su ámbito es la propia clase.
 - a) Los valores de los atributos son los mismos en todas las instancias.
 - b) La invocación de métodos no afecta al estado de las instancias.
 2. Los miembros instancias tienen como ámbito una instancia específica.
 - a) Los valores de los atributos pueden variar entre instancias.
 - b) La invocación de métodos puede afectar al estado de las instancias (es decir, cambiar el valor de sus atributos).

Para indicar que un miembro posee un ámbito de clasificador, hay que subrayar su nombre. De lo contrario, se asume por defecto que tendrá ámbito de instancia.

Relaciones

Una relación es un término general que abarca los tipos específicos de conexiones lógicas que se pueden encontrar en los diagramas de clases y objetos. UML presenta las siguientes relaciones:

Enlace

Un enlace es la relación más básica entre objetos.

Asociación

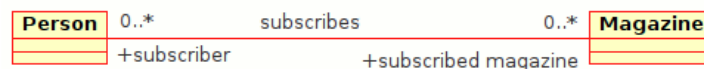


Figura A.1: Asociación

Una asociación representa a una familia de enlaces. Una asociación binaria (entre dos clases) normalmente se representa con una línea continua. Una misma asociación puede relacionar cualquier número de clases. Una asociación que relacione tres clases se llama asociación ternaria.

A una asociación se le puede asignar un nombre, y en sus extremos se puede hacer indicaciones, como el rol que desempeña la asociación, los nombres de las clases relacionadas, su multiplicidad, su visibilidad, y otras propiedades.

Hay cuatro tipos diferentes de asociación: bidireccional, unidireccional, agregación (en la que se incluye la composición) y reflexiva. Las asociaciones unidireccional y bidireccional son las más comunes.

Por ejemplo, una clase vuelo se asocia con una clase avión de forma bidireccional. La asociación representa la relación estática que comparten los objetos de ambas clases.

Agregación

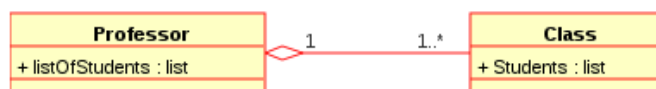


Figura A.2: Agregación

La agregación o agrupación es una variante de la relación de asociación “tiene un”: la agregación es más específica que la asociación. Se trata de una asociación que representa una relación de tipo parte-todo o parte-de.

Como se puede ver en la imagen del ejemplo (en inglés), un Profesor ‘tiene una’ clase a la que enseña.

Al ser un tipo de asociación, una agregación puede tener un nombre y las mismas indicaciones en los extremos de la línea. Sin embargo, una agregación no puede incluir más de dos clases; debe ser una asociación binaria.

Una agregación se puede dar cuando una clase es una colección o un contenedor de otras clases, pero a su vez, el tiempo de vida de las clases contenidas no tienen una dependencia fuerte del tiempo de vida de la clase contenedora (de el todo). Es decir, el contenido de la clase contenedora no se destruye automáticamente cuando desaparece dicha clase.

En UML, se representa gráficamente con un rombo hueco junto a la clase contenedora con una línea que lo conecta a la clase contenida. Todo este conjunto es, semánticamente, un objeto extendido que es tratado como una única unidad en muchas operaciones, aunque físicamente está hecho de varios objetos más pequeños.



Figura A.3: Asociación rombo sin rellenar y composición rombo negro

Composición

La representación en UML de una relación de composición es mostrada con una figura de diamante rellenado del lado de la clase contenedora, es decir al final de la línea que conecta la clase contenido con la clase contenedor.

Diferencias entre Composición y Agregación

Relación de Composición

-
1. Cuando intentamos representar un todo y sus partes. Ejemplo, un motor es una parte de un coche.
 2. Cuando se elimina el contenedor, el contenido también es eliminado. Ejemplo, si eliminamos una universidad eliminamos igualmente sus departamentos.

Relación de Agrupación

1. Cuando representamos las relaciones en un software o base de datos. Ejemplo, el modelo de motor MTR01 es parte del coche MC01. Como tal, el motor MTR01 puede hacer parte de cualquier otro modelo de coche, es decir si eliminamos el coche MC01 no es necesario eliminar el motor pues podemos usarlo en otro modelo.
2. Cuando el contenedor es eliminado, el contenido usualmente no es destruido. Ejemplo, un profesor tiene estudiantes, cuando el profesor muere los estudiantes no mueren con él o ella.

Bibliografía

- [1] Google, *Google Trends - NoSQL*, 2020. dirección: <https://trends.google.es/trends/explore?date=all&q=NoSQL>.
- [2] E. F. Codd, «A relational model of data for large shared data banks», *Communications of the ACM*, vol. 13, DOI: [10.1145/362384.362685](https://doi.org/10.1145/362384.362685).
- [3] C. Li, «Transforming relational database into HBase: A case study», en *2010 IEEE International Conference on Software Engineering and Service Sciences*, jul. de 2010, págs. 683-687. DOI: [10.1109/ICSESS.2010.5552465](https://doi.org/10.1109/ICSESS.2010.5552465).
- [4] A. Chebotko, A. Kashlev y S. Lu, «A Big Data Modeling Methodology for Apache Cassandra», en *2015 IEEE International Congress on Big Data*, jun. de 2015, págs. 238-245. DOI: [10.1109/BigDataCongress.2015.41](https://doi.org/10.1109/BigDataCongress.2015.41).
- [5] M. J. Mior, K. Salem, A. Abounaga y R. Liu, «NoSE: Schema Design for NoSQL Applications», *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, n.º 10, págs. 2275-2289, oct. de 2017, issn: 2326-3865. DOI: [10.1109/TKDE.2017.2722412](https://doi.org/10.1109/TKDE.2017.2722412).
- [6] D. Martínez-Mosquera, R. Navarrete y S. Lujan-Mora, «Modeling and Management Big Data in Databases—A Systematic Literature Review», *Sustainability*, 2020. DOI: <https://doi.org/10.3390/su12020634>.
- [7] J. Dullea, I.-Y. Song e I. Lamprou, «An analysis of structural validity in entity-relationship modeling», *Data & Knowledge Engineering*, vol. 47, n.º 2, págs. 167-205, 2003, Publisher: Elsevier.
- [8] A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla y P. Sánchez, «Mortadelo: Automatic generation of NoSQL stores from platform-independent data models», *Future Generation Computer Systems*, vol. 105, págs. 455-474, 2020, Publisher: Elsevier.
- [9] C. de Lima y R. dos Santos Mello, «A workload-driven logical design approach for NoSQL document databases», en *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, 2015, págs. 1-10.
- [10] M. J. Mior, K. Salem, A. Abounaga y R. Liu, «NoSE: Schema design for NoSQL applications», *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, n.º 10, págs. 2275-2289, 2017, Publisher: IEEE.
- [11] datafluent, *Kashlev Data Modeler*, 2020. dirección: <https://www.datafluent.org/>.

-
- [12] Amazon, *NoSQL Workbench for Amazon DynamoDB*, 2020. dirección: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/workbench.html>.
- [13] HacKolade, *HacKolade*, 2020. dirección: <https://hackolade.com/>.
- [14] Ramez Elmasri, *Fundamentos de SIstemas de Bases de Datos*. Pearson, ISBN: 84-7829-085-0.
- [15] P. Chen, *The entity-relationship model: Toward a unified View of data*. dirección: <https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.523.6679>.
- [16] Cristina Marta Bender, Claudia Deco, Juan Sebastián González Sanabria, María Hallo y Julio César Ponce Gallegos, *Tópicos Avanzados de Bases de Datos*, 1.^a ed. Iniciativa Latinoamericana de Libros de Texto Abiertos.
- [17] C. Coronel y S. Morris, *Database Systems: Design, Implementation, and Management*, 13.^a ed. Cengage, ISBN: 978-1-337-62790-0.
- [18] P. J. Sadalage y M. Fowler, *Nosql Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 1.^a ed. Pearson Education, ISBN: 978-0-321-82662-6.
- [19] *HTTP - Hypertext Transfer Protocol*. dirección: <https://www.w3.org/Protocols/>.
- [20] *HTML*. dirección: <https://developer.mozilla.org/es/docs/Web/HTML>.
- [21] H. W. Lie y B. Bos, *Cascading style sheets: Designing for the web, Portable Documents*. Addison-Wesley Professional, 2005.
- [22] *What is CSS?* Dirección: <https://www.w3.org/standards/webdesign/htmlcss#whatcss>.
- [23] *Documentation Sass*. dirección: <https://sass-lang.com/documentation>.
- [24] *Most Loved, Dreaded, and Wanted Web Frameworks - stack overflow*. dirección: <https://insights.stackoverflow.com/survey/2019#technology-most-loved-dreaded-and-wanted-web-frameworks>.
- [25] *JavaScript*. dirección: <https://developer.mozilla.org/es/docs/Web/JavaScript>.
- [26] T. Publishing, *TypeScript Programming Language*, 1.^a ed. Independently published, nov. de 2019, ISBN: 1-70883-980-1.
- [27] *stack overflow - Developer Survey Results 2019 - Most Popular Technologies*. dirección: <https://insights.stackoverflow.com/survey/2019#most-popular-technologies>.
- [28] Wired, *Wired - Get Started with Web Frameworks*, 2020. dirección: https://www.wired.com/2010/02/get_started_with_web_frameworks/.
- [29] React, *React - A JavaScript library for building user interfaces*, 2020. dirección: <https://reactjs.org/>.
- [30] Angular, *Angular - One framework, mobile & desktop*, 2020. dirección: <https://angular.io/>.
- [31] *¿Qué es Vue.js?* Dirección: <https://es.vuejs.org/v2/guide/>.

-
- [32] *What is NuxtJS?* Dirección: <https://nuxtjs.org/guide>.
- [33] Wikipedia contributors, *CSS framework* — *Wikipedia, The Free Encyclopedia*. 2020. dirección: https://en.wikipedia.org/w/index.php?title=CSS_framework&oldid=952876853.
- [34] *Introduction Material Design*. dirección: <https://material.io/design/introduction#goals>.
- [35] *Vuetify Component API Overview*. dirección: <https://vuetifyjs.com/en/components/api-explorer/>.
- [36] *Documentation Bootstrap*. dirección: <https://getbootstrap.com/docs/4.5/getting-started/introduction/>.
- [37] MongoDB, *MongoDB - MongoDB Documentation*, 2020. dirección: <https://docs.mongodb.com/manual/>.
- [38] MySQL, *MySQL - MySQL Documentation*, 2020. dirección: <https://dev.mysql.com/doc/>.
- [39] *GoJs*. dirección: <https://gojs.net/latest/index.html>.
- [40] *Fabric Documentation*, 2020. dirección: <http://fabricjs.com/fabric-intro-part-1>.
- [41] Wikipedia contributors, *D3.js* — *Wikipedia, The Free Encyclopedia*. 2020. dirección: <https://en.wikipedia.org/w/index.php?title=D3.js&oldid=956132961>.
- [42] M. Lutz, *Learning Python*, 5.^a ed. O'Really, jun. de 2013, ISBN: 978-1-4493-5573-9.
- [43] Wikipedia, *Django (framework)* — *Wikipedia, La enciclopedia libre*. 2020. dirección: [https://es.wikipedia.org/w/index.php?title=Django_\(framework\)&oldid=125055379](https://es.wikipedia.org/w/index.php?title=Django_(framework)&oldid=125055379).
- [44] U. C. los Ángeles, *Metodología de desarrollo de software*, 2020. dirección: <https://www.uladech.edu.pe/images/stories/universidad/documentos/2018/metodologia-desarrollo-software-v001.pdf>.
- [45] T. S. Guide, *The Definitive Guide to Scrum: The Rules of the Game*, 2020. dirección: <https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf#zoom=100>.
- [46] S. México, *SCRUM México - Escribiendo Historias de Usuario*, 2020. dirección: <https://www.scrum.mx/informate/historias-de-usuario>.
- [47] M. T. Gallego, *Metodología SCRUM*, 2020. dirección: <http://openaccess.uoc.edu/webapps/o2/bitstream/10609/17885/1/mtrigasTFC0612memoria.pdf>.
- [48] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
- [49] A. Abran, *Applied Software Measurement: Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress*. Shaker, 2006.
- [50] P. Galván, *Estudio de salarios SG 2020*, 2020. dirección: <https://sg.com.mx/estudios/salarios/2020>.

-
- [51] Wikipedia, *Diagrama de clases* — *Wikipedia, La enciclopedia libre*. 2020.
dirección: https://es.wikipedia.org/w/index.php?title=Diagrama_de_clases&oldid=126211624.