Universidad Mayor de San Simón

Facultad de Ciencias y Tecnología

ALGORITMOS Y TECNÍCAS DE GENERACÍON DE DATOS DE PRUEBA PARA BASE DE DATOS BASADOS EN IDIOMS

Tesis presentada por Milver F. Flores Acevedo
para obtener el grado de ingenieria de sistemas
Tutorado por juan marcelo flores soliz

2016

Ingenieria de Sistemas

Agradecimientos

Este trabajo se la dedico al creador de todas las cosas quién supo guiarme por el buen camino, darme fuerzas para seguir adelante y no desmayar en los problemas que se presentaban, enseñándome a encarar las adversidades sin perder nunca la dignidad ni desfallecer en el intento.

A mi familia quienes por ellos soy lo que soy. Para mis padres por su apoyo, consejos, comprensión, amor, ayuda en los momentos difíciles, y por ayudarme con los recursos necesarios para estudiar. Me han dado todo lo que soy como persona, mis valores, mis principios, mi carácter, mi empeño, mi perseverancia, mi coraje para conseguir mis objetivos. A mis hermanas por estar siempre presentes, acompañándome para poderme realizar.

A mi tutor por la orientación y ayuda que me brindó para la realización de este trabajo, por su apoyo y amistad que me permitieron aprender mucho más que lo estudiado en el proyecto.

Índice general

Índice de figuras

Índice de cuadros

Resumen

En el desarrollo de software que implica el uso de base de datos, es bastante común requerir realizar pruebas para conocer el comportamiento del software conjuntamente con la base de datos, para lo cual es necesario tener la base de datos poblada con datos de prueba similar cuando el software este en un entorno de producción.

Si bien tenemos claro que es necesario tener la base de datos poblada, no resulta ser tan sencillo poblar con datos de prueba. Este trabajo presenta una propuesta de algunas técnicas y estrategias que ayudan el llenado de datos de prueba en una base de datos.

Capítulo 1

Introducción

Cuando se está desarrollando software que hacen uso de una bases de datos, que está en gran parte de las aplicaciones. Es común tener la necesidad de realizar pruebas sobre la base de datos, para verificar si los datos se están gestionando de la manera correcta o verificar la eficiencia de las consultas, para esto se insertan datos de prueba haciendo uso de comandos del Lenguaje de definición de datos (DDL) [4].Otra manera es hacerlo por una interfaz gráfica dependiendo del sistema gestor de base de datos, entre las mas usadas tenemos a:

- **Postgresql**: PgAdmin.
- **MySQL**: PhpMyAdmin, MySQL Workbench.
- **SQLServer**: .

Los que trabajan con base de datos llenaron datos de prueba de esta manera en algún momento o siguen con los mismos métodos.

Insertar datos de prueba en una base de datos de forma manual sea por interfaz gráfica o por una interfaz de texto como ser la consola, no suele ser tan agradable porque: conlleva mucho tiempo de trabajo; no siempre se tiene en mente lo que se quiere etc. Por lo que normalmente se insertan pocos datos de prueba para ver el comportamiento de las consultas que se realizan en el software.

Una aplicación en un entorno de producción de seguro no lleva pocos datos, al contrario llegan almacenar gigabytes de información.

El problema surge ahí, porque no es lo mismo hacer pruebas con cien datos que con cien mil datos o mas, si bien una consulta funciona de manera correcta con los pocos datos, es diferente el comportamiento con una población de datos mas grande, puede que las consultas no funcionen de forma correcta o esta son muy lentas. Un error común en principiantes es realizar un `SELECT * FROM tabla` que de seguro no tendría problemas con los pocos datos, sin embargo en el entorno de producción es catastrófico.

Sin duda es un problema a considerarse en el desarrollo de software, sobre todo cuando no se realizan pruebas a una base de datos, dejándonos incertidumbres sobre su comportamiento con cantidades de datos que podría tener cuando el software ya esté en un entorno producción.

1.1. Aplicaciones generadores de datos de prueba para base de datos

Para los que son nuevos en el mundo de base de datos es normal desconocer sobre herramientas que facilitan la tarea, y optan en llenar de forma manual una base de datos, que resulta ser tedioso y puede llegar a no gustar ser responsable de la parte de la base de datos, si a principios de la aparición de la base de datos no se contaba con herramientas de apoyo en esta área, hoy en día existen herramientas que automatizan procesos como el llenado de datos de prueba en cantidades grandes sobre una base de datos ya existente.

Entre las herramientas para la generación de datos de prueba para base de datos, se puede mencionar a Datanamic Data Generator MultiDB perteneciente a Datanamic [6], Generatedata para mysql de código abierto con licencia GNU [8], EMS Data Generator for MySQL de la línea de EMS y EMS Data Generator for PostgreSQL también de la línea de EMS [12] y MyDatagen [5], son algunas que podemos mencionar lo cual no significa que sean las únicas, con estas herramientas podemos realizar el poblado de datos para su posterior realización de pruebas.

De las herramientas el mas destacado es Datanamic, por la forma en como permite configurarlo. En la página oficial [6] estan disponibles para MySql, PostgreSQL, Oracle y otros. En el caso de Datanamic para PostgreSQL es una de las más destaca-

bles entre las mencionadas ,donde se puede ver opciones de generar datos lo primero es escoger una base de datos existente en otra base de datos y lo que hace el software es reconocer caracteristicas de la base de datos, con sus respectivos tipo de datos y por defecto da la opción de generar cien registros, con la libertad de configurar a gusto además da la opción de escoger la fuente de datos que se hará uso, a partir de una lista, otro es obtener datos como nombres por ejemplo a partir de un archivo, y poder seleccionar y escoger si será aleatoriamente o una forma secuencial, si será único o si se repetirá esto dependiendo de cómo se quiere llenar datos.

De las otras herramientas mencionadas tienen una similitud en el manejo y en cómo se realiza el llenado de los datos, a diferencia de Generatedata que es una herramienta libre y de código abierto escrita en JavaScript, PHP y MySQL. Que permite generar de una forma rápida grandes volúmenes de datos personalizados en una variedad de formatos, para su uso en pruebas de software, rellenar bases de datos, etc. Los desarrolladores pueden escribir sus propios tipos de datos para generar nuevos tipos de datos aleatorios e incluso personalizar los tipos de exportación, Para las personas interesadas en la generación de datos de localización geográfica, se pueden añadir nuevos complementos para proporcionar nombres de regiones (estados, provincias, territorios, etc.), nombres de ciudades y formatos de códigos postales para su país, todo esto porque es libre de código abierto, donde podemos observar que los datos a llenar a una base de datos los extrae de su propia base de datos que incluye Generatedata y en su modelo haciendo ingeniería reversa se puede visualizar que los datos maneja almacenado todos los datos posibles como el nombre de países. Las críticas que podría tener esta herramienta es porque no hace el llenado a una base de datos existente lo cual lo quita puntos a favor además que solo funciona para MySQL entre sus punto a favor es que es libre de código abierto. Sin embargo Datanamic viene para distintos motores de base de datos pero si hay uno que es multifuncional.

Con un análisis sobre el uso de las herramientas de cualquiera de las mencionadas, llenando datos de prueba en una base de datos, el tiempo del llenado es considerablemente inferior a lo que llevaría hacerlo manualmente, cuanto más datos mayor es el tiempo en llenarlo en la forma manual, sin embargo haciendo uso de alguna de estas herramientas que nos ayudan en realizar esta tarea por nosotros, el tiempo es muy similar que llenar pocos registros así que es muy conveniente llenar la mayor cantidad

de datos en la base de datos siempre que se disponga una fuente por ejemplo una lista de nombres, si queremos llenar nombres en una entidad.

Sería mucho mejor encontrar información sobre cómo llenar ya que hay poca información sobre estas herramientas con una documentación no muy clara de algunas como MyDatagen, PgDatagen que sin embargo Datanamic si cuenta con una documnetacion mas clara [7] de cómo se realiza el llenado.

Una de las deficiencias que se puede encontrar y que le quita puntos a su favor, es cuando se tiene relaciones compuestas no las reconoce como tal y esto llega a ser un problema.

Las herramientas comerciales como es el caso de Datanamic, MyDatagen y PgDatagen, no provee el acceso al código fuente y es un problema saber cómo es la logica de la generación de datos, pero es observable mediante la interfaz gráfica el como elige el generador de datos adecuado para cada columna, basado en las características de la columna, con más de cuarenta generadores de datos incorporados (específicos para países e idiomas), genera datos realistas con el uso inteligente del generador (para, por ejemplo, códigos postales) y una gran colección de listas con nombres, direcciones, ciudades, calles, etc. Obtiene datos al azar de una fuente externa y obtiene datos de fuentes existentes como las otras tablas, opción para deshabilitar los desencadenantes, vista previa en tiempo real de los datos que se van a generar, genera datos de prueba para una base de datos completa o para una selección de tablas, incluye una utilidad de línea de comandos para automatizar aún más el proceso de generación de datos, inserta datos directamente en la base de datos o genera un secuencia de comandos `SQL` con instrucciones de inserción, guarda tu plan de generación de datos de prueba a un archivo de proyecto, validacin extensa de configuración del generador de datos, ejecuta secuencia de comandos de datos previas posteriores a la generación, detección automática de los cambios en el esquema de base de datos, opción de generar valores únicos.

De todas estas características es interesante conocer acerca de cómo realiza la generación de datos, al tratar de ver como lo realiza, no se cuenta con información suficiente de parte de la aplicación, solo se puede visualizar. Sin embargo el objetivo es entender cómo hace la generación de datos según lo requerido, al observar las herramientas listadas, los datos que generan van dependiendo siempre del tipo de

dato, y poder tomar como parámetros la cantidad de datos si serán únicos entre otras, todas estas caracyteristicas nos da una idea de como hacer un generador de datos.

Al hacer uso de cualquiera de estas herramientas se puede encontrar con un problema que se consideraría que es una deficiencia, ninguna de las mencionadas trata de ayudar al usuario en el orden del llenado de las tablas, si bien internamente lo haga no muestra cuales son tablas que se deben configurar primero y cuales las siguientes así sucesivamente.

Capítulo 2

Algoritmos de ordenamiento y mecanismos del manejo referencial

Una base de datos relacional esta fuertemente ligada al concepto Entidad Relación [2](E-R "Entity Relationship"). Una entidad que representa gráficamente a un concepto del mundo real o abstracta, que da lugar a una tabla en la base de datos. Una relación entre dos o más entidades describe alguna interacción entre las mismas, el tipo de relación dará lugar a un comportamiento entre las entidades involucradas [10].

En un modelo Entidad Relación (E-R) para base de datos basados en ER Idioms[11], se tiene patrones de diseño más definidos, las relaciones que llegan a tener entre entidades y la forma en que se hacen da lugar prácticamente a siguientes siete patrones de diseño para un modelo ER.

1. Una entidad que no hace referencia a otra pero si es referenciada es una entidad de tipo *catalogo*. Actúa como un tipificador y generalmente almacenan pequeñas cantidades de datos, los datos que se almacenan se conocen a priori y la cantidades de datos son predecibles lo cual no significa que sean estables pueden llegar ha incrementarse, la entidad que hace referencia al *catalogo* llega a ser una entidad de tipo *catalogado*.

2. En algunos casos se tiene entidades que se encuentran sueltas por lo tanto no hacen referencia ni son referenciadas por ninguna otra, la cual es una entidad

de tipo *simple*.

3. En un modelo entidad relación donde una entidad hace referencia a más de una y que su existencia depende de las mismas, a todo este conjunto se le denomina *composición*, consiste en que puede componer de más de dos incluyéndose así mismo.

4. Cuando una entidad es dependiente de la existencia de otra al que detalla es de tipo *detalle* de la alguna entidad maestra, donde el detalle obedece a la maestra, a diferencia de un catalogador en un maestro no se puede determinar los posibles datos a priori ni mucho menos estimar la cantidad aproximada de datos que pueda tener y que generalmente almacena cantidades grandes de datos.

5. Hay veces que una entidad dependa de sí mismo de esta forma llega a referenciar así mismo llegando a ser una relación recursiva a la cual llega a ser una entidad de tipo *reflexivo simple*, cuando se implementa este tipo de relación es importante tener en cuenta que la relación no debe ser de obligatoriedad.

6. En ocasiones es necesario relacionar entidades del mismo tipo y guardar una historia de ellas, la forma de representar este concepto es que una entidad representa la forma que se relacionan la entidad que hace una doble referencia, lo cual lleva a ser una entidad tipo *reflexivo compuesto*.

7. Cuando se quiere hacer una especialización a una entidad en particular esta llega a ser la entidad hija de de la entidad generalizada este tipo de relación es conocida como *is a* en idioms.

2.1. Técnicas de ordenamiento

En una base de datos las tablas tienen una secuencia de prioridades en el llenado de datos, una manera de obtener esta secuencia es buscar todas las tablas que no tengan ningun `foreign key`, posteriormente las tablas que tienen como `foreign key` que referencian a las anteriores que llenamos y asi sucesivamente de manera secuencial, hasta acabar con el ultimo al que no referencia ninguna otra tabla, esto

llegar a ser confuso sobre todo si son muchas tablas al momento de hacer el llenado, para lo cual es necesario alguna técnica para el orden correcto del llenado.

La propuesta en este proyecto sobre la técnica para obtener el orden según la prioridad en que deben ser llenado las tablas, es identificar primero todas aquellas que son un entidad catalogador, simples y aquellas que no dependen de ninguna otra en las que puede que en algunos casos pueden estar las entidades maestras o las que son padres de una generalización, los siguientes a identificar son los catalogados, las que hacen referencia a las identificadas anteriormente pero que estas no deben depender de otras que aún no se llenó para evitar tener problemas de inconsistencia.

2.2. Algoritmo de ordenamiento

Para obtener la lista de tablas según el orden en que debe ser llenado una base de datos, es necesario aplicar el siguiente algoritmo:

1. Crear una matriz bidimensional.

2. Seleccionar las entidades de tipo catalogo, simples, maestras que no dependan de otra entidad y entidades que sean de tipo *reflexivo simple* pero que no hagan referencia a otra distinta de el, para luego almacenar este conjunto en una matriz lineal y que est a su ves agregarlo en la matriz bidimensional.

3. De la matriz bidimensional obtener la última matriz y por cada elemento buscar todas las entidades que le hagan referencia y agregarlos en un nuevo arreglo lineal, una vez recorrido todos los elementos añadir el arreglo lineal en el arreglo bidimensional.

4. Verificar que se ha recorrido todas las entidades, en caso de que se recorrió todo pasar al paso cinco, en caso contrario repetir el paso tres.

5. si se llego hasta aquí es porque ya tenemos un arreglo bidimensional ya con el orden pero esta aun no es la lista para lo que requerimos ya que se da el caso que se pueda repetir mas de una ves el nombre de una entidad en los distintos matrices que contiene la matriz bidimensional por lo tanto es necesario aplicar el algoritmo de ordenación primeros en ser llenados a la matriz bidimensional.

2.3. Aplicación del algoritmo

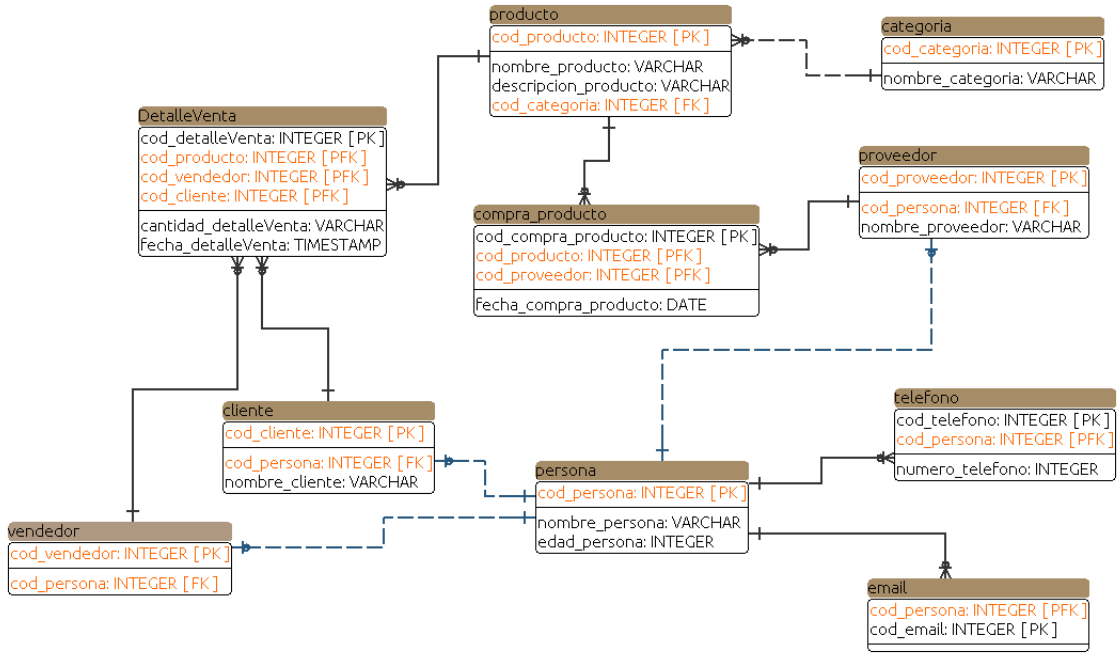


Figura 2.1: Modelo ER

En la Figura 2.1 Modelo ER, podemos observar las distintas entidades de las cuales es necesario identificar las entidades que no hagan referencia aplicando el paso dos del algoritmo podemos identificar una entidad catalogador *categoria* y una entidad que generaliza *persona*, a partir ellas buscamos los que le hacen referencia, *categoria* y *persona* la tomamos como raíz del árbol a generar, las entidades que hacen referencia serian *(producto, cliente, vendedor, proveedor, telefono, email)* abajo estarían *(detalleVenta y compra_producto)* y aplicando el algoritmo se llegaría a la siguiente Figura 2.2 que se encuentra en la página siguiente.

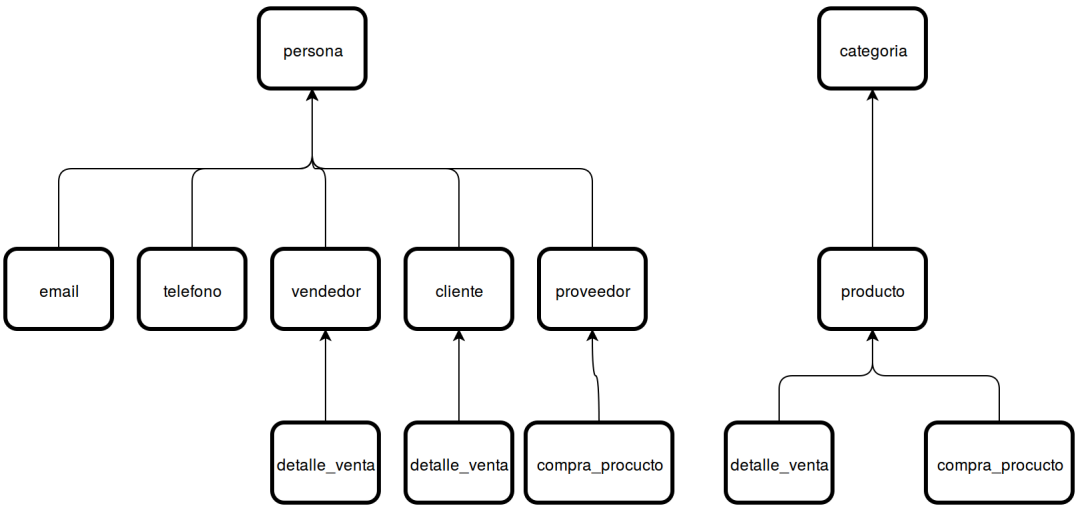


Figura 2.2: Modelo Ordenado

2.4. Algoritmo de ordenación *primeros en ser llenado*

Para obtener una lista ordenada de acuerdo al orden que se debe realizar el llenado, se inicia el recorrido de las matrices que trae la matriz bidimensional obtenida del algoritmo anterior, porque si recordamos la lista se ordenó según a que una entidad no dependiera de otra entidad, al recorrer podemos encontrarnos que una entidad se repite en distintos lugares, al inicio de la matriz bidimensional, puede también estar listada en medio o al final incluso se puede dar el caso que en uno de las matrices se repita más de una vez. Si vemos la Figura 2.2 la entidad *detalleVenta* se repite más de una vez en la misma fila.

Entonces cual lo tomamos como válido?. Para lo cual el recorrido se comienza con el último de las matrices que tiene la matriz bidimensional, una vez encontrada las demás que puedan encontrarse en posteriores no llegará a ser válida porque si bien hizo una referencia a un principio será valida el último. Esto se justifica porque si se encuentra al último es porque aún hace referencia a algún elemento de la matriz anterior al que pertenece, por lo tanto se debió esperar que se llegue hasta ese punto. para el caso de la Figura 2.2 listamos el último evitando que se repita. Para obtener una correcta lista ordenada según el orden en que deben ser llenados es necesario

seguir la siguiente secuencia de pasos:

1. Crear un matriz de tamaño de la cantidad de entidades de la base de datos seleccionada.

2. obtener la última matriz de la matriz bidimensional.

3. Recorrer la matriz y verificar por cada elemento no exista en la matriz del paso uno, en caso de que no exista añadir, en caso de que si no añadir y pasar a la siguiente elemento.

4. Eliminar la última matriz de la matriz bidimensional.

5. Si existen mas matrices en la matriz bidimensional volver a repetir el paso dos, caso contrario pasar al siguiente paso.

6. Invertir el orden de la matriz del paso uno.

7. Retornar la matriz creado en el paso uno que llegaría a ser el orden que se requiere.

2.5. Aplicación del algoritmo de *primeros en ser llenado*

Con el algoritmo llegamos a tener el siguiente orden en que debe ser llenado para el ejemplo dado en la figura 2.1.

Figura 2.3: Orden de llenado

El orden a seguir para este ejemplo es iniciar por el color verde terminando en el color rojo, el orden en cada fila no influye en el resultado permitiendo así la flexibilidad de tomar cualquier elemento para el inicio de cada fila o el conjunto de elementos del mismo color.

2.6. Mecanismo de *manejo de referencial* de una base de datos

Cuando un modelo entidad relación es llevado a un sistema gestor de base de datos, donde por cada entidad se crea una tabla y las referencias son representadas mediante las llaves primarias y llaves foráneas, En caso de un modelo entidad relación basado en ER Idioms tiene ciertas características que son.

2.6.1. Llaves primarias compuestas (`composite keys`)

Cuando se tiene más de un `primary key`, entre ellas las que son propias de la tabla y otras pertenecientes a las que referencia que vienen como primarias, llegan

a formar parte del `primary key` de la tabla, formando así `composite keys` para la misma. En conceptos de entidad relación en la figura 2.4 se puede observar las entidades que hacen referencia a otra.

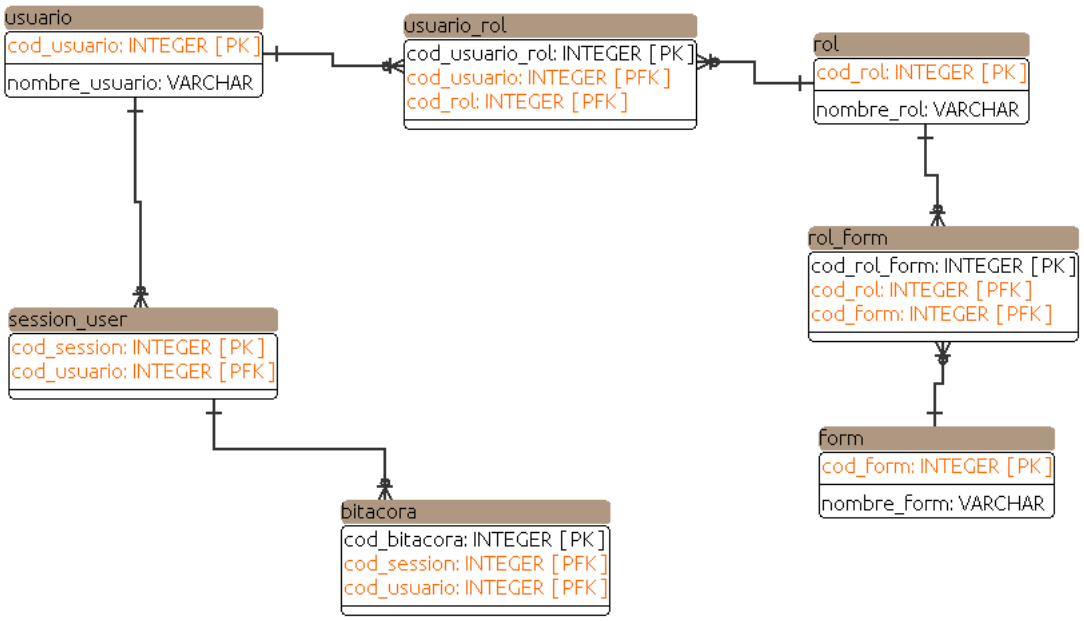


Figura 2.4: llaves compuestas

Donde se puede ver que la entidad *usuario_rol* hace referencia a las entidades *usuario* y *rol*, las tres entidades llegarían a formar una composición (*usuario_rol* compone de *usuario* y *rol*) donde la existencia de *usuario_rol* es dependiente de las que compone. Si este modelo lo llevamos a un gestor de base de datos un DBMS se llega a tener como en la figura 2.6.1.

Listing 2.1: SQL tabla usuario_rol

```
1 CREATE TABLE usuario_rol(
2   cod_usuario_rol serial NOT NULL,
3   cod_usuario INTEGER NOT NULL,
4   cod_rol INTEGER NOT NULL,
5   CONSTRAINT cod_usuario_rol PRIMARY KEY (cod_usuario_rol,
      cod_usuario,cod_rol),
6   CONSTRAINT rol_usuario_rol_fk FOREIGN KEY (cod_rol)
```

```
7       REFERENCES rol (cod_rol) MATCH SIMPLE
8       ON UPDATE NOT ACTION ON DELETE NOT ACTION,
9   CONSTRAINT usuario_usuario_rol_fk FOREIGN KEY (cod_usuario)
10      REFERENCES usuario (cod_usuario) MATCH SIMPLE
11      ON UPDATE NOT ACTION ON DELETE NOT ACTION
12 )
```

El `primary key` propia es independiente en cambio *cod_usuario* es perteneciente a la tabla *usuario* pero viene como `primary key` lo mismo sucede con el campo *cod_rol* que pertenece a la tabla *rol*, como ambas `foreing key` vienen como `primary key` la tabla *usuario_rol* llegaría a tener un `primary key` compuesta de tres *cod_usuario_rol, cod_usuario, cod_rol*. Cuando se quiere hacer una inserción de un registro a la tabla *usuario_rol* sin antes haber realizado una inserción a las tablas de *usuario* y *rol* cualquier DBMS no lo realiza la inserción por razones de primero debe existir datos en las tablas.

2.6.2. Mejor uso de Join

Al hacer uso de llaves compuestas(`composite keys`) hace que un identificador llegue más allá de lo que normalmente se acostumbra veamos en la figura 2.5.

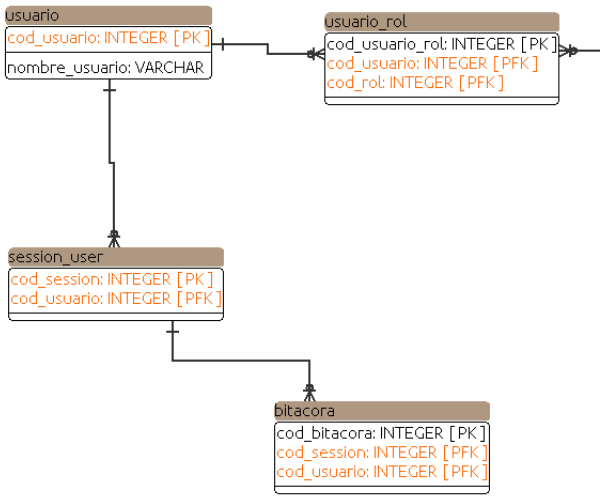


Figura 2.5: Modelo E-R con llaves compuestas

Donde en la entidad bitácora se tiene tres `primary key` llegando a ser una llave compuesta(`composite key`) y una de ellas es *cod_usuario* que si bien viene de *session_user* realmente su origen es en *usuario*, por lo tanto podemos hacer un join entre *bitacora* y *usuario* evitando hacerlo con *session_user* obteniendo asi una consulta mas eficiente.

2.7. Mecanismo de referenciación

Cuando generamos datos de prueba para una base de datos es importante tener en cuenta el manejo de las llaves compuestas(`composite keys`), no podemos generar al azar los `foreign key` porque se llegaría a tener problemas de inconsistencia.
Una técnica para evitar los problemas de inconsistencia es tener como fuente la tabla al que se referencia y los atributos tomarlo como base para la generación de n datos.

2.7.1. Referencia simple

Las referencias simples son cuando una tabla recibe solo un `primary key` o `foreign key` por parte de otra, observemos la figura 2.7.1.

Listing 2.2: Referencia simple

```
1  CREATE TABLE usuario_rol(
2    cod_usuario_rol serial NOT NULL,
3    cod_usuario INTEGER NOT NULL,
4    cod_rol INTEGER NOT NULL,
5    CONSTRAINT cod_usuario_rol PRIMARY KEY (cod_usuario_rol,
       cod_usuario,cod_rol),
6    CONSTRAINT rol_usuario_rol_fk FOREIGN KEY (cod_rol)
7        REFERENCES rol (cod_rol) MATCH SIMPLE
8        ON UPDATE NOT ACTION ON DELETE NOT ACTION,
9    CONSTRAINT usuario_usuario_rol_fk FOREIGN KEY (cod_usuario)
10       REFERENCES usuario (cod_usuario) MATCH SIMPLE
11       ON UPDATE NOT ACTION ON DELETE NOT ACTION
12 )
```

En la figura 2.7.1 se puede observar el código `SQL` (por sus siglas en inglés Structured Query Language) de la tabla *usuario_rol* de la figura 2.4. La tabla mencionada esta conformada de dos llaves que no son propias de la misma, *cod_usuario* forma parte de la tabla *usuario* y *cod_rol* apunta a la tabla *rol*, ambos son campos que apuntan

a su respectiva tabla de manera única por lo tanto se deja entender que se hace una referencia simple de llaves.

2.7.2. Referencia compuesta

Las referencias compuestas son cuando una tabla recibe más de un `primary key` o `foreign key` por parte de otra. Es importante tomar los atributos que apuntan a otra como un conjunto de atributos para manejar como base para la generación de los datos requeridos.

Listing 2.3: Referencia compuesta

```
1 CREATE TABLE bitacora(
2   cod_bitacora serial NOT NULL,
3   cod_session INTEGER NOT NULL,
4   cod_usuario INTEGER NOT NULL,
5   CONSTRAINT bitacora_pk PRIMARY KEY (cod_bitacora,cod_session,
     cod_usuario),
6   CONSTRAINT session_user_bitacora_fk FOREIGN KEY (cod_session,
     cod_usuario)
7       REFERENCES "session_user" (cod_session,cod_usuario) MATCH
     SIMPLE
8       ON UPDATE NOT ACTION ON DELETE NOT ACTION
9 )
```

En la figura 2.7.2 el campo *cod_session* y *cod_usuario* de la tabla *bitacora* hace referencia a *session_user* a los campos *cod_session* y *cod_usuario*, son dos campos que apuntan a la misma tabla que este caso *session_user* por lo tanto llegaría a ser una referencia compuesta.

Capítulo 3

Algoritmos de generación de datos de prueba

En el desarrollo de un software que hace uso de una base de datos es muy importante para los desarrolladores sobre todo para los que son encargados de la base de datos, trabajar con una base de datos con datos de prueba que se asemejen más a las reales tanto en cantidad como en tipo de datos, para eso es necesario tener una cantidad grande de datos de prueba cuanto más mejor, esto nos lleva a un enfoque de generar datos de prueba para base de datos.

Poseer con una cantidad suficiente de datos de prueba no es tan sencillo por lo que un desarrollador normalmente acostumbra hacer la inserción de forma manual lo cual toma un tiempo excesivo. Cuando se quiere hacer la inserción es necesario tener en cuenta los diferentes tipos de datos que la gran mayoría de los DBMS posee.

Cuadro 3.1: Tipos de datos

Tipo de dato	**Características**
`VARCHAR`(tamaño)	Almacena cadenas de caracteres de una longitud variable. La longitud máxima son 4000 caracteres
`CHAR`(tamaño)	Almacena caracteres con una longitud fija. Siendo 2000 caracteres el mximo
`NUMBER`(precisión, escala)	Almacena datos numéricos, tanto enteros como decimales, con o sin signo. Precisión, indica el número máximo de dígitos que va a tener el dato. Escala, indica el número de dígitos que puede haber a la derecha del punto decimal.
`LONG`	Almacena cadenas de caracteres de longitud variable. Puede almacenar hasta 2 gigas de información
`LONG RAW`	Almacena datos binarios. Se emplea para el almacenamiento de gráficos, sonidos, etc. Su tamao máximo es de 2 gigas
`DATE`	Almacena información de fechas y horas.De forma predeterminada almacena un dato con el siguiente formato:siglo/año/mes/dia/hora/minutos/segundos.Este formato se puede cambiar con otros parámetros.
`RAW`(tamaño)	Almacena datos binarios. Puede almacenar como mucho 2000 bytes.
`ROWID`	Se trata de un campo que representa una cadena hexadecimal que indica la dirección de una fila en su tabla
`NVARCHAR2`(tamaño)	Es similar al varchar2 pero el tamaño de un carácter depende de la elección del juego de caracteres. El tamaño máximo es 2000 bytes.
`NCHAR`(tamaño)	Similar al `CHAR` y con las mismas características que el `nvarchar2`
`CLOB`	Similar al `LONG` y se usa para objectos carácter
`BLOB`	Similar al `LONG RAW`. Este se usa para objetos binarios.

Los tipos de datos listadas anteriormente son genéricas en gran parte de lo gestores de base de datos, esta lista no es la misma en lo gestores de base de datos varia según el DBMS que incorporan sus tipos de datos personalizadas con características propias a los tipos de datos de la lista anterior.

3.1. Algoritmos de generación de datos

La generación de datos de prueba para base de datos no llega a ser tan sencilla por los diferentes tipos de datos y el límite en el tamaño, pero llega a ser una solución para pruebas que se quiera realizar a una base de datos determinada. Para realizar la generación de datos es necesario tener algoritmos generadores por cada tipo de datos que tome en cuenta las características de la misma y sean capaces de generar cantidades grandes tomando como base una pequeña cantidad de datos o a lo mejor sin contar ninguna base.

3.2. Tipos numéricos

Los tipos numéricos consisten en enteros de 2, 4 u 8 bytes y flotantes de 4 u 8, y un número de presición decimal a elección. Los números enteros es uno de los más sencillos a generar, con solo incrementar en una unidad al número inicial que nos pasan como parámetro llegamos en algún momento al límite que también se pasa como parámetro.

3.3. Tipos monetarios

Los tipos de datos monetarios en realidad se almacenan como un numero cualquiera y que estas tiene similitud con las decimales, el DBMS es quien se encarga de dar el formato necesario.

3.4. Tipos de caracteres

3.4.1. Generación de nombres

Para generar nombres es necesario tener una lista o también podemos generarlos haciendo combinaciones de las vocales y las consonantes a continuación haremos una descripción de cómo generar aplicando por cada una de ellas.

Generación de nombres a partir de vocales y consonantes

Un nombre esta compuesta generalmente por más de tres caracteres puede que tenga menos y a lo mucho llega a tener 10 a excepción de algunos. Y mucho depende del idioma o el país.

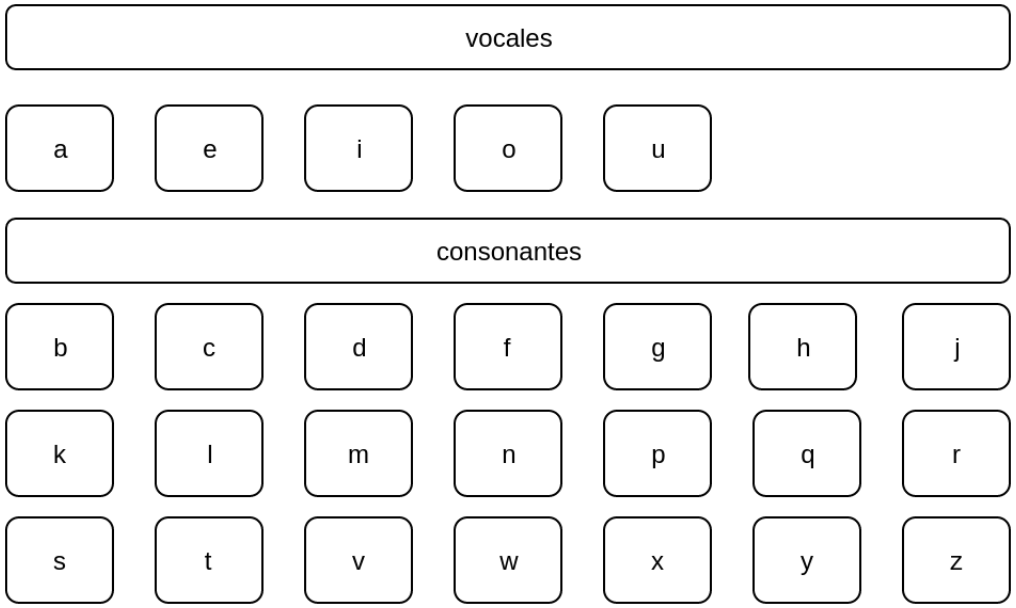


Figura 3.1: consonantes y vocales

Para generar un nombre es importante tener claro lo mencionado anteriormente, para ello aplicamos el siguiente algoritmo para generar nombres haciendo uso de las vocales y consonantes.

1. Crear una matriz e insertar las cinco vocales.

2. Crear una matriz e insertar las consonantes podemos omitir las que no deseemos usarlas.

3. Generar un número aleatorio entre 1 a 10 que representa la cantidad de caracteres del nombre y crear una variable al que se le asignará el nombre.

4. Declarar una variable bandera que indicará el turno a quien corresponde sea una vocal o consonante.

5. Preguntamos si la variable bandera indica constante si es verdadero pasamos al siguiente paso 6 caso contrario saltar al paso 7.

6. Generamos un número aleatorio entre 0 a un máximo de la cantidad de elementos del arreglo de consonantes menos uno, este llega a ser el índice del elemento

a obtener de las consonantes para luego realizar la concatenación a la variable que maneja el nombre, contradecir la variable bandera.

7. Generamos un número aleatorio entre 0 a un máximo de 4 por la cantidad de vocales, esta llega a ser la posición del elemento a obtener del arreglo de vocales y luego concatenamos a la variable que maneja el nombre, contradecir la variable bandera.

8. Preguntamos si el tamaño del nombre es igual al número generado en el paso 3, si es verdadero pasamos al paso 9 y si es falso volvemos al paso 5.

9. Finalizamos y retornamos el nombre generado.

Algorithm 1 Algoritmo de generación de palabras

Require: mínimo , máximo de caracteres.

```
1: cantidad ← random(minimo,maximo)
2: numero ← random(0,1)
3: cadena ← ""
4: while n < cantidad do
5:    if numero == 0 then
6:       cadena ← cadena + obtenerVocal
7:    else
8:       cadena ← cadena + ontenerConsonante
9:    end if
10:   numero ← random(0,1)
11: end while
12: return cadena
```

El Algoritmo 1 hace uso de vocales y consonantes para generar palabras.

Generación de nombres a partir de una lista de nombres y apellidos

La generación de nombre más apellido requiere de una lista de nombre y otro de apellidos, para generar se aplica el siguiente algoritmo:

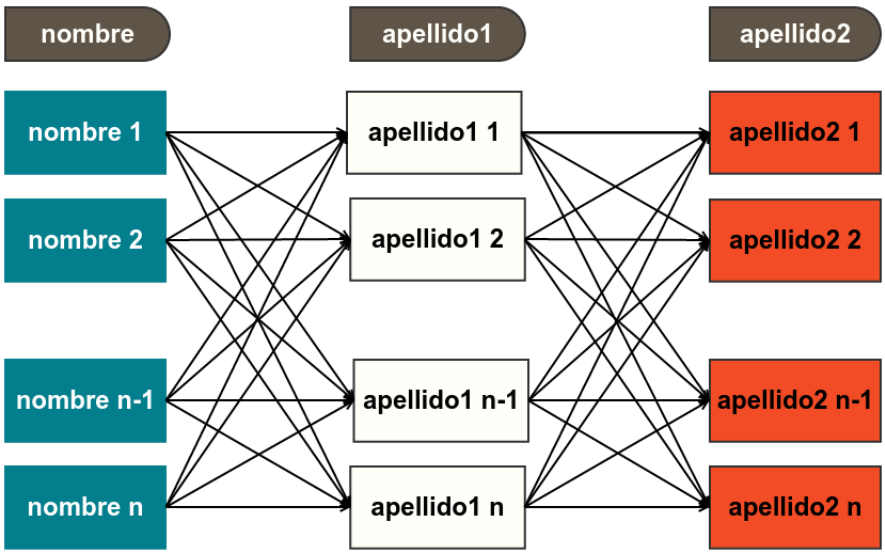


Figura 3.2: apartir de una lista de nombres

Algorithm 2 Algoritmo de generacion de nombresLista

Require: nombres [ ], apellidosPaternos [ ],apellidosMaternos [ ]

```
1: nombresCombinadas [ ]
2: ind
3: for iniNomb ← 0; iniNomb < tamanio(nombres);iniNomb++ do
4:    for iniApePat ← 0; iniApePat<tamanio(apellidosPaternos);iniApePat++ do
5:       for iniApePat ← 0; iniApePat<tamanio(apellidosPaternos);iniApePat++
         do
6:          nombresCombinados[ind] ← nombres [ iniNomb ] + apellidosPaternos [
            iniApePat ] + apellidosMaternos [ iniApePat ]
7:       end for
8:    end for
9: end for
10: return  true
```

3.5. Tipos de datos binarios(Binary Data Types)

Los sistemas gestores de base de datos(DBMS) permiten almacenar archivos en formatos como `bytea`, `blob` entre otros, variando estos segun el DBMS utilizado.
En este proyecto se pondra a consideracion trabajar con postgresql el cual permite almacenar archivos en formato `bytea`. El tipo de dato `bytea` permite almacenar objetos de gran tamaño, postgreSQL no conoce nada sobre el tipo de información que se almacena `bytea`, simplemente lo considera como una secuencia de bytes.
A momento de generar datos de prueba para base de datos, el tipo de dato `bytea` llega a ser un caso especial, debido a que no es posible crearlo como cualquier otro dato, como ser un numero de teléfono que llega a ser combinaciones de números bajo ciertas condiciones o el caso de un nombre que son combinaciones de vocales y consonantes, sin embargo `bytea` es posible generar haciendo uso de archivos existentes teniendo solo la ruta del archivo es suficiente para poder insertar en la base datos.

3.6. Tipos fecha/hora

3.6.1. Generación de fechas

Una fecha esta compuesta por tres partes:

- Año la parte del año para nuestros días comprende de cuatro dígitos desde 1000 hasta el año 9999 el rango no estrictamente establecido.
- Mes. la parte del mes se representa en número de dos dígitos que comprende en un rango ya establecida, con un inicio de 01 hasta 12 representando los doce meses del año.
- Día. la cantidad de días en un mes es variable mucho depende a que mes nos referimos, el rango comprende desde el día uno y con un final variable desde 28 a 31 días.

Para generar una cantidad de fechas es necesario tener una fecha inicial y final. a partir de ello hacemos la combinaciones para obtener todas las fechas entre el rango dado por parámetro veamos en la figura 3.3

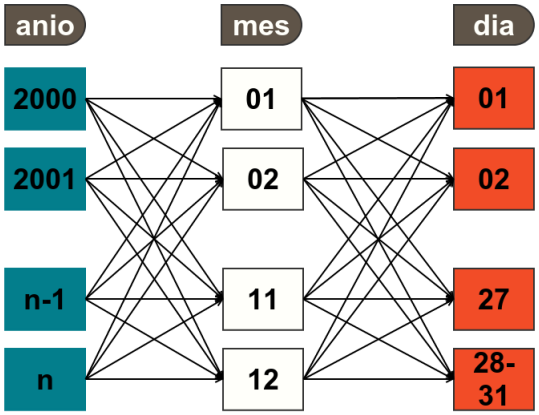


Figura 3.3: generación de fechas

Algoritmo de generación de fechas

Para realizar la aplicación del algoritmo de generación de fechas es necesario tener dos datos una fecha inicial y otra fecha limite final donde es importante que la fecha inicial debe ser una fecha anterior a la final, la cantidad de fechas a obtener dependería del tamaño de rango que existe entre la inicial y la final.
Veamos el algoritmo de generación de fechas en formato dd/mm/aa, tomando en cuenta que la cantidad de días es variable por cada mes además tomando en cuenta años bisiestos.

Algorithm 3 Algoritmo de generación de fechas

Require: fechaInicio fechaFinal

```
1: fechas[ ]
2: contador ← 0
3: if fechaInicio< fechaFinal then
4:    while fechaInicio < fechaFinal do
5:       fechas [ contador ] ← fechaInicio+ 1 dia
6:       contador ← contador+1
7:    end while
8: else
9:    return error
10: end if
11: return fechas
```

El algoritmo 3 es necesario que la fecha inicial sea menor a la fecha final.

3.6.2. Generación de dato tipo Time

La estructura del dato tipo tiempo es muy similar a las fechas con la diferencia de que estas tienen un rango ya establecidos sin ninguna variación, esta compuesta por:

1. Hora. las hora se representa en un número de dos dígitos comenzando desde las 00 horas hasta las 23.

2. Minuto. los minutos también se representa en un número de dos dígitos con un inicio en 00 hasta 59 minutos.

3. Segundo. el rango es idéntica a los de los minutos.

Para generar el tipo de dato time podemos hacer combinaciones de las tres partes que tiene este tipo de dato, realizando combinaciones podemos obtener la cantidad de datos que requerimos.
Veamos la figura siguiente de como hacer las combinaciones:

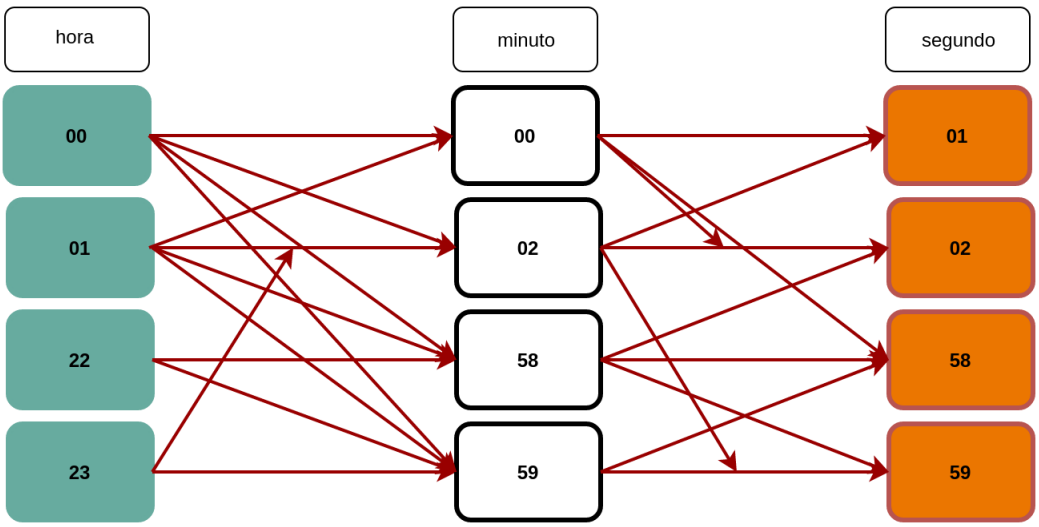


Figura 3.4: generación de Time

Algorithm 4 Algoritmo de generación de fecha hora

Require: fechaHoraInicio fechaHoraFinal

```
1: fechasHoras[ ]
2: contador ← 0
3: if fechaHoraInicio< fechaHoraFinal then
4:    while fechaHoraInicio < fechaHoraFinal do
5:       fechasHoras [ contador ] ← fechaHoraInicio+ 1 minuto
6:       contador ← contador+1
7:    end while
8: else
9:    return  error
10: end if
11: return  fechasHoras
```

El algoritmo 4 es necesario que la fecha inicial sea menor a la fecha final.

3.7. Tipos de direcciones de red

Las direcciones de red que almacena una base de datos son la IPv4, IPv6 y dirección mac.

3.7.1. Estructura de una dirección IPv4

Al igual que la dirección de una casa tiene dos partes (una calle y un código postal), una dirección IP también está formada por dos partes: el ID de host y el ID de red.

ID de red

La primera parte de una dirección IP es el ID de red, que identifica el segmento de red en el que está ubicado el equipo.

ID de host

La segunda parte de una dirección IP es el ID de host, que identifica un equipo, un router u otro dispositivo de un segmento.El ID de cada host debe ser exclusivo en el ID de red, al igual que la dirección de una casa es exclusiva dentro de la zona del código postal.
La IPv4 tiene un un formato (xxx.xxx.xxx.xxx) en que debemos basarnos para generar los limites van establecidos donde (0 < xxx 255) donde se debe tomar en cuenta que no se usa hasta 255.

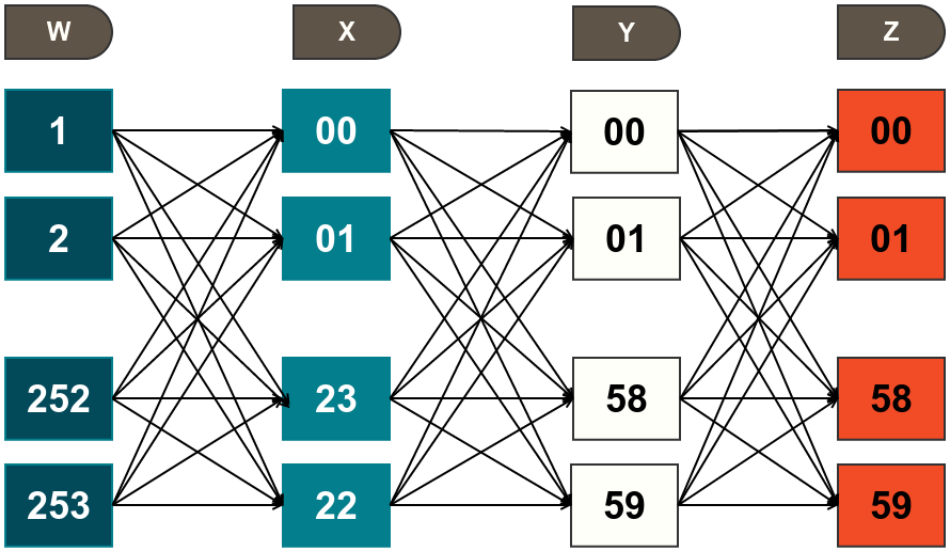


Figura 3.5: ipv4

Algorithm 5 Algoritmo de generación de IPv4

Require: inicio final

```
1: direcciones[ ]
2: contador ← 0
3: if inicio< fiinal then
4:     while inicio < final do
5:         direcciones [ contador ] ← inicio+ 1
6:         contador ← contador+1
7:     end while
8: else
9:     return error
10: end if
11: return direcciones
```

Capítulo 4

Metadatos

En este capítulo se hará muestra de las técnicas necesarias para obtener la estructura de una base de datos, una base de datos existente por lo general lleva tablas que de alguna manera se relacionan entre ellas. El detallar la estructura de una base de datos comprende:

- listar todas las tablas de una base de datos.
- listar las tablas que hacen referencia y a que tablas.
- listar las llaves primarias de una tabla.
- listar las llaves foráneas de una tabla.
- detallar una tablas como ser el tipo de dato, tamaño, si puede ser nulo etc...

Es muy importante realizar lo listado anteriormente para obtener la estructura de una base de datos, y usar estos resultados para construir una interfaz gráfica de configuración de generación de datos de acuerdo a las características de la columna de una tabla. Obtener de la estructura de una base de datos puede variar dependiendo del DBMS(Database Managment System) con la que trabajemos, entre los DBMS tenemos a varios entre las mas usadas están MySQL, PostgreSQL, Oracle, SQLServer y otras. En este proyecto se hace la eleccion de trabajar con postgresSQl las razones para esta elección son las siguientes.

- Soporta llaves primarias compuestas(lo cual nos permite aplicar patrones de diseño ER Idioms).

- Es un DBMS de licencia BSD libre.

Esto no significa que en las otras no se puedan aplicar este proyecto de lo contrario son aplicables a DBMS relacionales claro que existen diferencias de como manejan los datos internamente cada una de ellas por ejemplo mencionar que postgreSQL los almacena en metadatos todas las tablas que creamos, por lo tanto podemos deducir que para obtener la estructura de una base de datos es necesario trabajar con los metadatos de postgreSQL.

4.1. Metadatos en PostgreSQL

PostgreSQL tiene una arquitectura que involucra muchos estilos, en su nivel mas alto es un esquema clásico cliente-servidor, mientras que el acceso a metadatos es un esquema en capas.

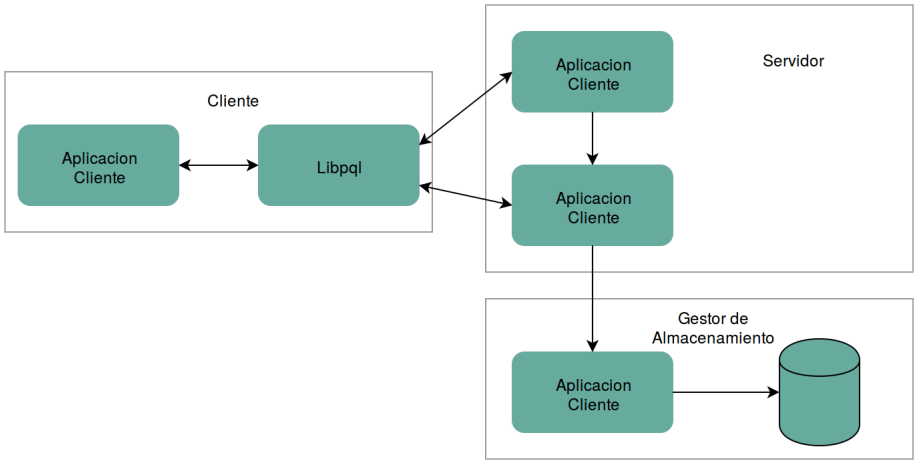


Figura 4.1: Arquitectura PostgreSQL

- Libpq es el responsable de manipular las comunicaciones entre la aplicación cliente y el postmaster(Servicio del PostgreSQL en el servidor).
- El servidor esta compuesto por dos grandes subsistemas, "Postmaster"que es el responsable de aceptar las comunicaciones con el cliente, autentificar y dar acceso. "Postgre"se encarga de la administración de los consultas(querys) y comandos enviados por el cliente. PostgreSQL trabaja bajo el concepto de

"process per user", eso significa un solo proceso cliente por conexión. Tanto el Postmaster como el Postgre deben estar junto en el mismo servidor siempre.

- El gestor de almacenamiento(Storage Manager) es responsable de la administración general del almacenamiento de los datos, controla todos los trabajos del back end incluido la administración del buffer, archivos, bloqueos y control de consistencia de la información.

4.1.1. Almacenamiento y organización de datos

Los datos siempre se va guardar en "disco"(esto puede no ser literalmente un Hard Drive). Esto genera un intenso trabajo de I/O, cuando leemos la data la sacamos de "disco"para pasarla a la RAM, cuando escribimos la bajamos de la RAM al "disco".

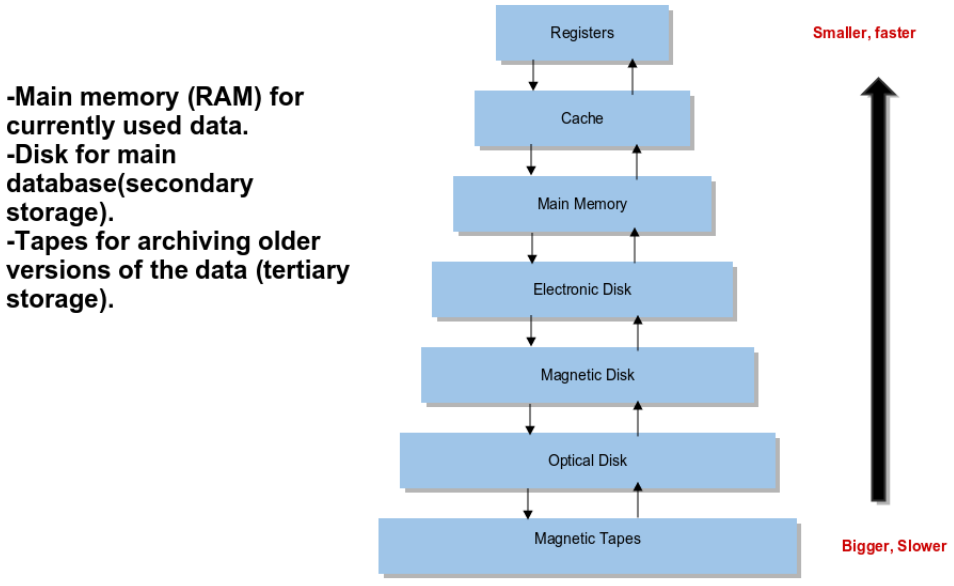


Figura 4.2: Almacenamiento y Organizacion de datos

PostgreSQL posee un "Storage Manager"(MySql tiene 5 o más por ejemplo), esta compuesto por varios módulos que proveen administración de las transacciones y acceso a los objetos de la base de datos.
Los módulos se programaron bajo tres lineamientos bien claros:

- Manejar transacciones sin necesidad de escribir código complejo de recuperación en caso de caídas.
- Mantener versiones históricas de la data bajo el concepto de "graba una vez, lee muchas veces".
- Tomar las ventajas que ofrece el hardware especializado como multiprocesadores, memoria no volátil, etc.

Los índices

Cada tipo de búsqueda tienen un tipo de índice adecuado para trabajarla, básicamente un índice es un "archivo"donde está parte de un dato y estructura de una tabla con "search key"de búsqueda.

4.1.2. Como se procesa un consulta(Query)

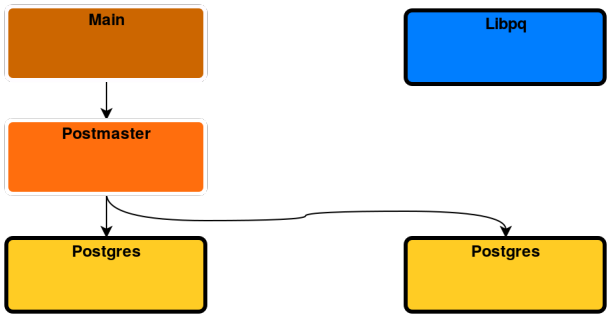


Figura 4.3: Como se procesa un query

Luego pasa por:

```
1 FindExec: found "/var/local/postgres/ ./bin/postgres" using argv[0]
2 DEBUG: connection: host=[local] user=postgres database=test
3 DEBUG: InitPostgtres
4 DEBUG: StartTransactionCommand
5 DEBUG: query: SELECT firstname
6      FROM friend
7      WHERE age = 33;
```

```
8
9  [query is proceesd]
10
11 DEBUG: ProcessQuey
12 DEBUG: CommitTransactionCommand
13 DEBUD: proc_exit(0)
14 DEBUG: shmen_exit(0)
15 DEBUG: exit(0)
```

- Un identificador de reglas de que lo escrito sea sintácticamente entendible, que los dígitos y los números sean reconocibles.
- Luego se descompone "palabra" a "palabra" el query, para pasar a la estructura que le corresponde según el query, en este caso a la estructura de un `SELECT`, esto se ve asi:

```
1  simple_select: SELECT opt_distinc target_list
2      into_clause from_clause where_clause
3      group_clause having_clause
4      [
5       SelectStmt *n = makeNode(SelectStmt);
6       n->distincClause = $2;
7       n->targetList = $3;
8       n->istemp = (bool)((Value *| lfirst($4)))->val.ival;
9       n->into = (char*) lnext($4);
10      n->fromClause = $5;
11      n->whereClause = $6;
12     n->groupClause = $7;
13     n->havingClause = $8;
14     $$ = (Node)n;
15     ]
```

```
1  typedef struct SelectStmt
2  {
3   NodeTag type;
4   List *distincClause;
5
6   char *into;
7   bool istemp;
8   List *targetList;
9   List *fromClause;
10  Node *whereClause;
11  List *groupClause;
```

```
12  Node *havingClause;
13
14  List *sortClause;
15  char *portalname;
16  bool binary;
17  Node *limitOffset;
18  Node *limitCount;
19  List *forUpdate;
20
21  SetOperartion op;
22  bool all;
23  struct SelectStmt *larg;
24  struct SelectStmt *rarg;
25 } SelectStmt;
```

Métodos para relacionar tablas

Nested loop join Consume mas recursos de memoria pero la cantidad de búsquedas a realizar a realizar es menor.

Merge join Requiere que a data este ordenada para ubicar las relaciones, el costo esta justamente en mantener la data ordenada.

Hash join Aparentemente sería la forma mas rápida de acceder a un dato gracias a la creación de tablas indexadas, pero limitada a una búsqueda de igualdad. Las combinaciones hash requerirá una combinación de igualdad predicado (un predicado comparación de los valores de una tabla con los valores de la otra tabla utilizando el operador igual ‘ =′),las combinaciones hash también puede ser evaluado por un predicado anti-join (un predicado seleccionar valores de una tabla cuando no hay valores relacionados se encuentran en el otro). Dependiendo de los tamaños de las tablas, diferentes algoritmos se pueden aplicar.
El “Executor”toma el plan de ejecución que el “planer”le entrega e inicia el procesamiento, ejecuta un “plan tree”. Este “plan tree”tiene varios nodos de ejecución que se van ejecutando uno a uno y de cada uno de ellos se obtiene un set de datos(tuplas). Los nodos tienen subnodos y otros a su vez otros subnodos, tantos como sea necesario.

4.2. Obtener la estructura de una base de datos

En este caso requerimos obtener una estructura de información con detalles por cada columna de una tabla semejante a esta:
`column_name` => `cedula`,
`datatype` => `integer`,
`key` => `PRI`,
`is_nullable` => `NO`,
`max_length` => `8`,
`column_default` =>
Podemos observar el detalle de un atributo de una tablas por lo tanto podemos hacerlo para cada una y donde:
`datatype`: es un tipo de dato interno de postgresql.
key: `UNI` = `unique`, el campo es un índice único.
key: `PRI` = `primary key`, el campo es un índice primario,
key: `FK` = `foreign key`, el campo es un índice de una llave foránea.
max_length: Si el campo es integer, muestra la precisión del entero (2,4,8), si es un varchar, la longitud en caracteres (ej. 75).
column_default: muestra el tipo de valor por defecto de la tabla; si la tabla es serial, veremos la llamada al nextval de la secuencia: ej. nextval(personas_cliente_id_seq'::regclass). Lo que nos permite determinar que campo de nuestra tabla es serial (auto-incremental).

4.2.1. Obtener el detalle de una tabla

Para entender cada tabla del **pg_catalog** debe ser interrogada con el **oid** de la tabla, que lo sacamos de **pg_class**. Los campos y sus atributos, los sacamos de la tabla **pg_attribute**. El tipo de datos lo sacamos de la tabla **pg_type** los constraints de la tabla los obtenemos de la tabla **pg_constraint** y el valor por defecto, lo sacamos de la tabla **pg_attrdef**. La sentencia construida para sacar esa información de una sola vez de todas las tablas es esta:

Usando OIDs

Listing 4.1: Query para obtener detalle tabla con OIDs

```
SELECT pgca.attname as column_name,
       t.typname as data_type,
CASE
     WHEN cc.contype='p' THEN 'PRI'
     WHEN cc.contype='u' THEN 'UNI'
     WHEN cc.contype='f' THEN 'FK'
     ELSE ''
END AS key,
CASE
     WHEN pgca.attnotnull=false THEN 'YES'
ELSE    'NO'
END AS is_nullable,
CASE
     WHEN pgca.attlen=-1 THEN(pgca.atttypmod-4)
     ELSE pgca.attlen
END as max_length,
d.adsrc as column_default
FROM  pg_catalog.pg_attribute pgca
     LEFT JOIN pg_catalog.pg_type t ON
       t.oid=pgca.atttypid
     LEFT JOIN pg_catalog.pg_class c ON
       c.oid=pgca.attrelid
     LEFT JOIN pg_catalog.pg_constraint cc ON
       cc.conrelid=c.oid AND
       cc.conkey[1]=pgca.attnum
     LEFT JOIN pg_catalog.pg_attrdef d ON
       d.adrelid=c.oid AND
       pgca.attnum=d.adnum
WHERE c.relname='TABLA' AND
   pgca.attnum>0 AND
   t.oid = pgca.atttypid.
```

Donde <TABLA> representa el nombre de la tabla a la que queremos interrogar para obtener sus metadatos. Si el modelo de la Figura 2.1 llevamos al gestor de base de datos en este caso PostgreSQL y hacemos uso del código de Figura 4.2.1 obtendremos algo similar a la siguiente imagen.

Data Output | Explain | Messages | History

	column_name name	data_type name	key text	is_nullable text	max_length integer	column_default text
1	cod detalleventa	int4	PRI	NO	4	nextval('detalleventa cod detalleventa seq'::regclass)
2	cod cliente	int4	FK	NO	4	
3	cantidad detalleventa	varchar		NO	-5	
4	cod vendedor	int4	FK	NO	4	
5	cod producto	int4	FK	NO	4	
6	fecha detalleventa	date		NO	4	

Figura 4.4: Detalle tabla OIDs

Donde:

- *column_name* muestra en nombre del columna de la tabla.
- *data_type* indica que tipo que almacena esta columna.
- *key* indica si es una llave sea primaria, foránea o sea único.
- *is_nullable* nos indica si este campo puede ser nulo.
- *max_length* indica el tamaño de memoria de información máxima.
- *colum_default* en este campo nos indica si es autoincremental en PostgreSQL(serial, bigserial y smallserial) que normalmente se suele usar en llaves primarias las cuales no son obligatorias insertar ya que el DBMS se encarga de realizarlo por nosotros.

Son algunos campos almacenadas en el metadato de las muchas que se puede obtener y depende de lo que necesitemos saber, las listadas son básicas sobre la información detallada de una determina tabla de una base de datos. El query de la Figura 4.4 para obtener los metadatos de una tabla no siempre tiene que ser de esa manera podemos hacerlo también de la siguientes forma:

Usando Information schema

PostgreSQL a partir de la versión 8.0 introdujo el `INFORMATION_SCHEMA`. Las vistas definidas en el `INFORMATION_SCHEMA` le dan acceso a la información almacenada en las tablas del sistema PostgreSQL. El `INFORMATION_SCHEMA` se define como parte del estandar `SQL` y encontrarás un `INFORMATION_SCHEMA` en sistemas de bases de datos

más comerciales (y algunos de código abierto).Por ejemplo, para ver una lista de las tablas definidas en la base de datos actual, puede ejecutar el comando:

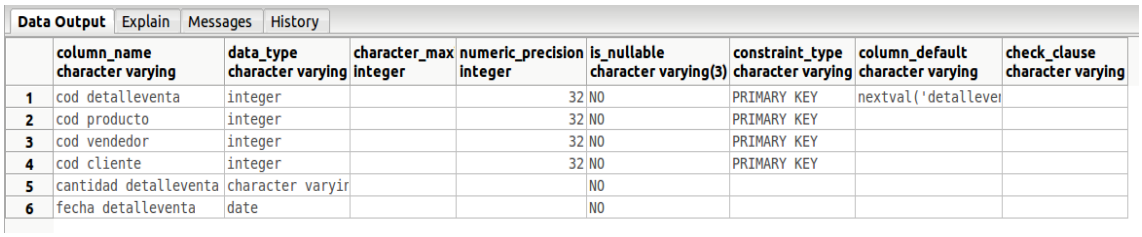
Listing 4.2: Query para detalle obtener el detalle de una tabla information scheme

```
SELECT tc.column_name,
       data_type,
       character_maximum_length,
       numeric_precision,
       is_nullable,
       tcs.constraint_type,
       column_default,
       check_clause
FROM information_schema.columns AS tc
     LEFT OUTER JOIN
        information_schema.constraint_column_usage AS cc
      ON tc.table_name = cc.table_name AND
        tc.column_name = cc.column_name
     LEFT OUTER JOIN
        information_schema.table_constraints AS tcs
      ON tcs.constraint_name = cc.constraint_name
     LEFT OUTER JOIN
        information_schema.check_constraints AS cccs
      ON cccs.constraint_name = tcs.constraint_name
WHERE tc.table_name = 'NOMBRE DE LA TABLA' AND
      tc.table_schema = 'public' AND
     (tcs.constraint_type='PRIMARY KEY' OR
      tcs.constraint_type='CHECK' OR
      tcs.constraint_type ISNULL)
ORDER BY ordinal_position;
```

En esta consulta `SQL` podemos ver las diferencias con la anterior, en esta ya no hacemos uso de los `OIDS` las tablas que son parte del metadato que usamos son:

```
SELECT * FROM information_schema.columns
SELECT * FROM information_schema.constraint_column_usage
SELECT * FROM information_schema.table_constraints
SELECT * FROM information_schema.check_constraints
```

Como resultado tenemos la siguiente información:



Data Output | Explain | Messages | History

	column_name character varying	data_type character varying	character_max integer	numeric_precision integer	is_nullable character varying(3)	constraint_type character varying	column_default character varying	check_clause character varying
1	cod detalleventa	integer		32	NO	PRIMARY KEY	nextval('detallever	
2	cod producto	integer		32	NO	PRIMARY KEY		
3	cod vendedor	integer		32	NO	PRIMARY KEY		
4	cod cliente	integer		32	NO	PRIMARY KEY		
5	cantidad detalleventa	character varyir			NO			
6	fecha detalleventa	date			NO			

Figura 4.5: information schema

Donde:

- *column_name* muestra en nombre del columna de la tabla.
- *data_type* indica el tipo de datos que esta permitido insertar, si comparamos con resultados de la figura 4.4 aquí nos devuelve el tipo de dato(integer en lugar de int4) como definimos en el modelo de la Figura 2.1 Modelo ER.
- *constraint_type* indica si es una llave sea primaria, foránea o sea único.
- *is_nullable* nos indica si este campo puede ser nulo.
- *character_max_length* indica el tamaño de memoria de información máxima.
- *colum_default* en este campo nos indica si es autoincrementable en PostgreSQL(serial, bigserial y smallserial) que normalmente se suele usar en llaves primarias las cuales no son obligatorias insertar ya que el DBMS se encarga de realizarlo por nosotros.
- *check_clause* en esta columna nos muestra si es campo tiene restricciones sobre la inserción de datos ejemplo: podemos decidir si queremos registrar edades entre 18 a 60 años.
- *numeric precision* nos indica el tamaño del tipo, aparte de pertenecer a un cierto tipo en los DBMS suelen tener tipos de datos mas precisos.

4.2.2. Obteniendo las relaciones entre las tablas

La estructura de las relaciones entre tablas en una base de datos es el resultado de su modelo ER, donde las relaciones en PostgreSQL son representas por *constraints*

en un sistema gestor de base de datos.

Listing 4.3: Query para obtener el detalle de referencias

```
SELECT (SELECT relname
        FROM pg_catalog.pg_class c
         LEFT JOIN pg_catalog.pg_namespace n ON
         n.oid = c.relnamespace
        WHERE c.oid=r.conrelid) as tablas,
        conname,
        pg_catalog.pg_get_constraintdef(oid, true) as ref
FROM pg_catalog.pg_constraint r
WHERE r.conrelid
   IN(SELECT c.oid
      FROM pg_catalog.pg_class c LEFT JOIN
           pg_catalog.pg_namespace n ON
           n.oid = c.relnamespace
      WHERE c.relname !~ 'pg_' AND
            c.relkind = 'r' AND
            pg_catalog.pg_table_is_visible(c.oid)) AND
r.contype = 'f'
```

De alguna manera necesitamos saber mediante un script las relaciones entre tablas de una base de datos, que tablas se relación con otra y exactamente que columnas están involucradas, al decir las columnas involucradas nos referimos exactamente a las columnas de una determinada tabla que son las llaves foráneas y que estas existen en la tabla que es referenciada, Es lo que precisamente el sript nos da como resultado mostrado en la Figura 4.6.

Data Output | Explain | Messages | History

	tablas name	conname name	referencias text
1	detalleventa	producto detalleventa fk	FOREIGN KEY (cod producto) REFERENCES producto(cod producto)
2	detalleventa	cliente detalleventa fk	FOREIGN KEY (cod cliente) REFERENCES cliente(cod cliente)
3	detalleventa	vendedor detalleventa fk	FOREIGN KEY (cod vendedor) REFERENCES vendedor(cod vendedor)
4	vendedor	persona vendedor fk	FOREIGN KEY (cod persona) REFERENCES persona(cod persona)
5	producto	categoria producto fk	FOREIGN KEY (cod categoria) REFERENCES categoria(cod categoria)
6	cliente	persona cliente fk	FOREIGN KEY (cod persona) REFERENCES persona(cod persona) ON UPDATE RESTRICT
7	proveedor	persona proveedor fk	FOREIGN KEY (cod persona) REFERENCES persona(cod persona)
8	telefono	persona telefono fk	FOREIGN KEY (cod persona) REFERENCES persona(cod persona)
9	compra producto	producto compra producto fk	FOREIGN KEY (cod producto) REFERENCES producto(cod producto)
10	compra producto	proveedor compra producto fk	FOREIGN KEY (cod proveedor) REFERENCES proveedor(cod proveedor)
11	email	persona email fk	FOREIGN KEY (cod persona) REFERENCES persona(cod persona)

Figura 4.6: Referencias entre tablas

En la Figura 4.6 tenemos el resultado de las tablas que llegan a referenciar a otra y las tablas que son referenciadas donde:

- **tablas** Nos muestra la lista de tablas que hacen referencia, puede encontrarse que el nombre de una tabla llegue a repetirse en mas de una ocasión no es un problema, detallaremos después de ver la explicación de la tercera columna.
- **conname** Muestra el `CONSTRAINT` de la relación, que seria algo como en nombre de la relación entre las tablas involucradas.
- **referencias** En esta columna de la Figura 4.6 nos trae toda la información necesaria para ser usado. Analicemos una de ellas.

```
"FOREIGN KEY (cod_producto) REFERENCES producto(cod_producto)"
```

  La cadena de texto posee información relevante donde *(cod_producto)* es el campo que hace referencia como indica `REFERENCES` a la tabla *producto* y al campo *(cod_producto)*. Aunque el resultado es en modo texto existen formas de solucionarlo para tener la información separada a lo que necesitamos, una manera es realizar un parseo al texto que las distintos lenguajes de programación ya tienen funciones implementadas para estas tareas.

En la columna *tablas* llegan a repetirse en nombre de una tablas en mas de una vez esto es debido a que nos lista por cada relación que llegue a tener una tabla con otras

4.2.3. Obteniendo las tablas independientes

Para obtener las tablas que son independientes de otras es necesario usar el script anterior la cual nos entrega un conjunto solo de las tablas que se relacionan de alguna manera entre ellas y tener otro conjunto de todas las tablas de la base de datos, como se tiene estos dos conjuntos de tablas hacemos una operación de resta entre los conjuntos. La lista de todas las tablas menos la lista de las tablas que se relacionan, como resultado se tiene las tablas que son independientes y que llegarían a ser los primeros en ser llenados.

Listing 4.4: Query para obtener tablas independientes

```
SELECT tablename
FROM pg_tables
WHERE schemaname = 'public' AND
```

```
4       tablename NOT IN
5      (SELECT (SELECT relname
6               FROM pg_catalog.pg_class c LEFT JOIN
7                    pg_catalog.pg_namespace n ON
8                    n.oid = c.relnamespace
9               WHERE
10                   c.oid=r.conrelid) as nombre
11      FROM pg_catalog.pg_constraint r
12      WHERE r.conrelid IN
13            (SELECT c.oid
14             FROM pg_catalog.pg_class c LEFT JOIN
15                  pg_catalog.pg_namespace n ON
16                  n.oid = c.relnamespace
17             WHERE c.relname !~ 'pg_' AND
18                   c.relkind = 'r' AND
19                   pg_catalog.pg_table_is_visible(c.oid))
20      AND r.contype = 'f')
```

El código SQL para obtener esa información es la que vemos en la Figura 4.2.3, si ejecutamos esta consulta nos daría el siguiente resultado.

Data Output | Explain | Messages | History

	tablename name
1	persona
2	categoria

Figura 4.7: Tablas independientes

Si analizamos la Figura 2.1 Modelo ER del Capitulo **??** podemos observar que las entidades que son independientes que no hacen referencia a otra son las mismas que nos da como resultado en la Figura 4.7.

4.3. Ordenando los metadatos

Si ya contamos con la información de los metadatos de una base de datos es necesario por una parte tener claro en el detalle de una tabla, que las llaves foráneas pueden ser un conjunto de columnas que hagan referencia a una determinada tabla.
Al momento de hacer las inserciones de datos se debe tener cuidado con este caso si insertamos valores a columnas que hacen referencia estas deben existir en la tabla

referenciada veamos un ejemplo:

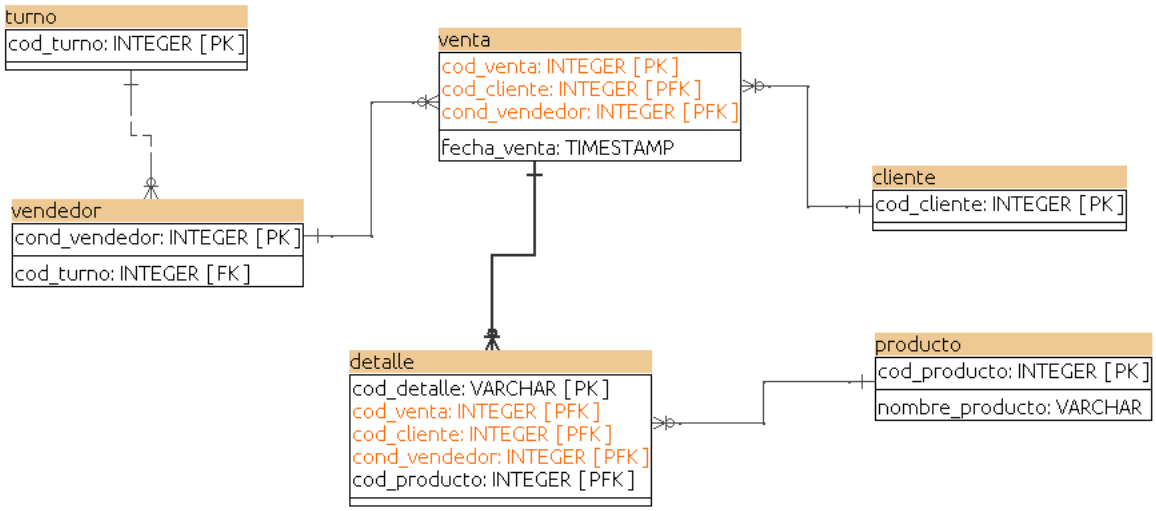


Figura 4.8: Modelo ER compuesto

	cod_venta [PK] serial	cod_cliente [PK] integer	cond_vendedor [PK] integer	fecha_venta timestamp without tim
1	1	1	1	2014-09-09 00:00:00
2	2	1	2	2014-01-25 00:00:00
3	3	2	1	2014-11-11 00:00:00
4	4	2	3	2014-12-22 00:00:00
5	5	3	3	2014-11-21 00:00:00
*				

Figura 4.9: tabla venta

	cod_detalle [PK] character	cod_venta [PK] integer	cod_cliente [PK] integer	cond_vendedor [PK] integer	cod_producto [PK] integer
1	1	1	1	1	1
2	2	1	1	1	2
3	3	1	1	1	3
4	4	2	1	2	1
*					

Figura 4.10: tabla detalle inserción Correcta

	cod_detalle [PK] character	cod_venta [PK] integer	cod_cliente [PK] integer	cond_vendedor [PK] integer	cod_producto [PK] integer
1	1	1	1	1	1
2	2	1	1	1	2
3	3	1	1	1	3
4	4	2	1	2	1
*		2	1	1	1

Figura 4.11: tabla detalle inserción Incorrecta

Como podemos observar en la Figura 4.8 la entidad venta es una composición de *vendedor* y *cliente* y que esta a la vez llega ser maestra de la entidad *detalle*, por lo tanto la entidad *detalle* tiene una llave compuesta. Si el modelo lo llevamos a un sistema gestor de base de datos en este caso PostgreSQL y llenamos con uno cuantos datos de prueba como vemos en la Figura 4.9 que esta compuesta de llaves compuestas.

Al realizar la inserción en *detalle* debemos tener cuidado en no cometer el error de la ultima inserción que se quiere hacer en la Figura 4.11, esta llega a ser incorrecta debido a que no existe una fila de (*cod_venta,cod_cliente,cod_vendedor*) en la tabla *venta* con valores de (*2,1,1*), llegando a no cumplir la integridad referencial además son datos inconsistentes. La ultima inserción de la Figura 4.10 es correcta porque si vemos la tabla Figura 4.9 podemos encontrar una fila también conocida como tupla que (*cod_venta,cod_cliente,cod_vendedor*) tengan los valores (*2,1,2*) cumpliendo así la integridad referencial y consistencia de datos.

La otra parte es el ordenar la lista de tablas según la prioridad que deben ser llenados, esta claro que las tablas independientes son los primeros de ahi en adelante aun no esta claro, por lo tanto es necesario desarrollar mecanismos para obtener una lista de tablas según el orden en que se requiere.

Para evitar errores de estos dos casos es necesario desarrollar mecanismos que ayuden de alguna manera a solucionar el problema, en el Capitulo **??**. ya desarrollamos mecanismos de como obtener una lista ordenada haremos aplicación de dicho algoritmo y como manejar las llaves compuestas para evitar los problemas de la Figura 4.11, haremos uso de esas técnicas.

4.3.1. Ordenando las tablas

La Figura 4.2.3 nos da como resultado las tablas que deben ser llenados primero, las siguientes son algunas de la lista que nos entrega el query de la Figura 4.2.2, que prácticamente están desordenados.
Si el modelo entidad relación de la Figura 4.8 lo llevamos a postgreSQL y aplicamos el query de la Figura 4.2.2 obtenemos el siguiente resultado:

Output pane

Data Output | Explain | Messages | History

	tablas name	conname name	referencias text
1	detalle	venta detalle fk	FOREIGN KEY (cod cliente, cod venta, cond vendedor) REFERENCES venta(cod cliente, cod venta, cond vendedor)
2	detalle	producto detalle fk	FOREIGN KEY (cod producto) REFERENCES producto(cod producto)
3	vendedor	fecha factura vendedor fk	FOREIGN KEY (cod turno) REFERENCES turno(cod turno)
4	venta	cliente factura fk	FOREIGN KEY (cod cliente) REFERENCES cliente(cod cliente) ON UPDATE CASCADE ON DELETE CASCADE
5	venta	vendedor factura fk	FOREIGN KEY (cond vendedor) REFERENCES vendedor(cond vendedor)

Figura 4.12: Detalle de relaciones entre tablas

En la Figura 4.12 en la primera columna se tiene en nombre de tablas que hacen referencia a las entidades de la base de datos, pero en la tercera columna no se tiene separada en nombre de la tabla que es referenciada por que es necesario separarlos de alguna manera, analicemos el formato de texto que nos da como resultado para la primera linea perteneciente a *detalle*:

```
1 "FOREIGN KEY (cod_cliente, cod_venta, cond_vendedor) REFERENCES
     venta(cod_cliente, cod_venta, cond_vendedor)"
```

Lo que haremos es separar en cinco partes la cadena de texto:

1. `FOREIGN KEY`
2. `cod_cliente, cod_venta, cond_vendedor`
3. `REFERENCES venta`
4. `cod_cliente, cod_venta, cond_vendedor`
5. ...

Las manera de implementar puede variar de acuerdo a la tecnología sea java, php, python etc... sin embargo muchas de estas tecnologías ya vienen implementadas estas funciones para hacer estas tareas, por ejemplo en java podemos llevar la cadena de texto a un arreglo de textos simplemente definimos delimitadores que este caso serian "(,)" obteniendo así un resultado similar a la que listamos. A partir de esa lista escogemos el de la posición 2 iniciando a contar desde 0 que llega ser *REFERENCES venta* en este caso en particular, esta cadena lo volvemos a separar en dos:

1. `REFERENCES`
2. `venta`

Como ya tenemos el nombre de la tabla retornamos este valor como la tabla que es referenciada para *detalle*. Realizamos esto para cada una de la lista de la Figura 4.12.

Cabe aclarar que en la segunda columna en la segunda separación de texto que hicimos puede que en algunos casos sobre todo cuando se hace uso de scheme(esquemas) en postgreSQL venga concatenada en nombre del scheme antes del nombre de la tabla concatenada con un punto seguido con el nombre de la tabla ej.
public.venta.
Lo cual no debería preocuparnos por el simple hecho de que nos ayuda a a identificar en que scheme se encuentra la tabla.Como resultado de las operaciones que se hizo se obtiene el siguiente resultado.

Cuadro 4.1: tablas que referencian a otra

tablas que referencian	**tablas referenciadas**
detalle	venta
detalle	producto
vendedor	turno
venta	cliente
venta	vendedor

En la primera columna se tiene las tablas que hacen referencia y en la segunda columna las que son referenciados. Para hacer uso del algoritmo de ordenación 2.2 del Capitulo 2, ya contamos con datos hasta el paso dos por lo tanto pasamos al paso tres donde realizamos la búsqueda para todas aquellas entidades que le hacen referencia a los que son independientes, que para el modelo ER de la Figura 4.8 llegaría a ser lo siguiente:

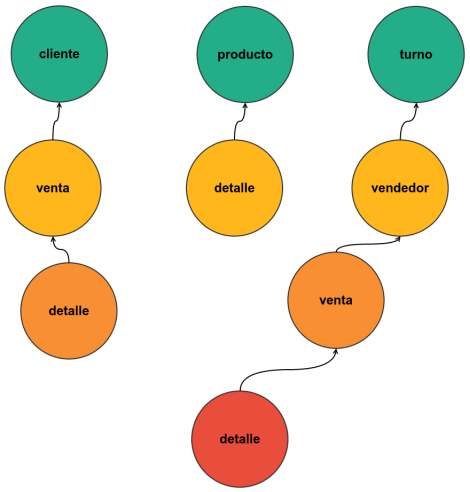


Figura 4.13: secuencia

Donde podemos observar que en nombre de las tablas se llegan a repetir en varios lugares aplicamos el 2.4 de Capitulo 2 , como resultado llegamos a tener el siguiente orden:

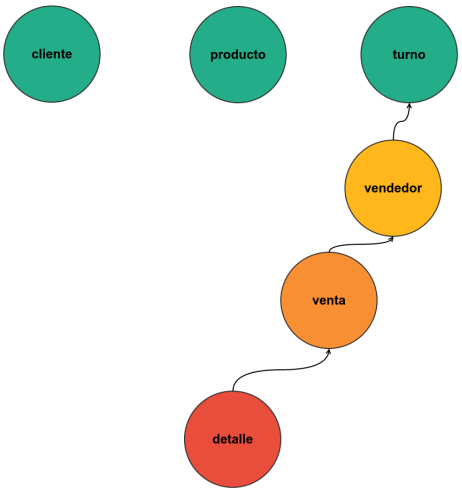


Figura 4.14: Orden correcto

4.3.2. Uniendo `foreign keys`

Una vez que ya tenemos la información detallada por cada una de las tablas que hacen referencia y de las cuales las columnas que estén involucradas llegan a ser

llaves extranjeras, que si bien pueden ser parte de la llave primaria compuesta de la tabla o simplemente ser una llave foránea, al momento de insertar puede llegar a dar lo mismo, por lo tanto no vamos a centrarnos en ese detalle.
La información que nos provee el query de la Figura **??** nos da una detallada información sobre una tabla en particular pero para lo que necesitamos generar datos de prueba es importante tener mecanismos que eviten cometer errores en las llaves que no son propias de una tabla.Para lo cual haremos uso de **??**. En el query de la Figura 4.12 tenemos una información valiosa en la tercera columna:

```
1 "detalle";"venta_detalle_fk";"FOREIGN KEY (cod_cliente, cod_venta,
     cond_vendedor) REFERENCES venta(cod_cliente, cod_venta,
     cond_vendedor)"
2 "detalle";"producto_detalle_fk";"FOREIGN KEY (cod_producto)
     REFERENCES producto(cod_producto)"
```

La tabla *detalle* con las columnas *(cod_cliente, cod_venta, cond_vendedor)* hace referencia a la tabla *venta* a las columnas *(cod_cliente, cod_venta, cond_vendedor)*, además la columna *(cod_producto)* hace referencia a la tabla *producto* a la columna *(cod_producto)*, son las llaves que no son propias de la tabla *detalle*, veamos y analicemos la información detallada que nos provee el query de la Figura **??** sobre esta tabla

Output pane

Data Output | Explain | Messages | History

	column_name character varying	data_type character varying	character_m integer	numeric_precision integer	is_nullable character varying(3)	constraint_type character varying	column_default character varying	check_clause character varying
1	cod detalle	character varying			NO	PRIMARY KEY	nextval('detalle coc	
2	cod venta	integer		32	NO	PRIMARY KEY		
3	cod cliente	integer		32	NO	PRIMARY KEY		
4	cond vendedor	integer		32	NO	PRIMARY KEY		
5	cod producto	integer		32	NO	PRIMARY KEY		

Figura 4.15: Detalle de la tabla *detalle*

Fijémonos en el detalle que nos da como información la Figura 4.15, las llaves que no son propias no están agrupadas de acuerdo a la tabla que referencián como sucede en la Figura 4.12 donde si lo agrupa pero solo nos provee esos campos que son llaves que apuntan a otra tabla, a diferencia de la Figura 4.15 si nos da la información de todas las columnas.

El formato detallado de la tabla que necesitamos es unir esas dos informaciones que tenemos obtener algo similar al siguiente cuadro.

Cuadro 4.2: tabla de referencias para la tabla detalle

nombre columna	**tipo de dato**	**es primaria?**	**serial**	**tabla a la que referencia**	**columnas a la que referencia**
cod_detalle	INTEGER	PRIMARY KEY	si	NULL	null
cod_producto		FORANEA		producto	cod_producto
cod_venta, cod_cliente, cod_vendedor		FORANEA		venta	cod_venta, cod_cliente, cod_vendedor

Si recordamos el query de la figura 4.2.2 da como resultado la lista de tablas que referencián en este caso no necesitamos la información de todos, requerimos específicamente para una tabla determinada. Para lo cual haremos alguna modificación al query de la Figura 4.2.2 quedando de la siguiente forma:

Listing 4.5: Query para detalle referencias para una tabla

```
SELECT(SELECT relname
       FROM pg_catalog.pg_class c
            LEFT JOIN
                 pg_catalog.pg_namespace n ON
                 n.oid=c.relnamespace
       WHERE
            c.oid=r.conrelid) as nombre,
       conname,
       pg_catalog.pg_get_constraintdef(oid,true)AS ref
FROM
        pg_catalog.pg_constraint r
WHERE r.conrelid IN
        (SELECT
                c.oid
         FROM pg_catalog.pg_class c
             LEFT JOIN
                pg_catalog.pg_namespace n ON
                n.oid = c.relnamespace
         WHERE
               c.relname !~ 'pg_' AND
               c.relkind='r' AND
               pg_catalog.pg_table_is_visible(c.oid))AND
      r.contype = 'f' AND
        (SELECT relname
         FROM pg_catalog.pg_class c
             LEFT JOIN
```

```
27                pg_catalog.pg_namespace n ON
28                n.oid = c.relnamespace
29          WHERE
30                c.oid=r.conrelid)='nombreTabla';";
```

A diferencia del query de la figura 4.2.2 en este query especificamos exactamente para que tabla queremos saber a quienes referencia agregando al final las siguientes lineas de código sql:

Listing 4.6: Parte que determina para una tabla

```
1             AND
2              (SELECT relname
3               FROM pg_catalog.pg_class c
4                    LEFT JOIN
5                    pg_catalog.pg_namespace n ON
6                    n.oid = c.relnamespace
7               WHERE
8                     c.oid=r.conrelid)='nombreTabla';";
```

Con la adición del código extra hacemos que no filtre solo para la tabla que requerimos bastara con solo cambiar el *nombreTabla*. El resultado de este query nos daría solo los registros donde una determinada tabla hace referencia, veamos para el caso de la tabla *detalle* :

Output pane

Data Output | Explain | Messages | History

	nombre name	conname name	referencias text
1	detalle	producto detalle fk	FOREIGN KEY (cod producto) REFERENCES producto(cod producto)
2	detalle	venta detalle fk	FOREIGN KEY (cod cliente, cod venta, cond vendedor) REFERENCES venta(cod cliente, cod venta, cond vendedor)

Figura 4.16: Detalle de referencias de la tabla detalle

Los resultados no llegan a ser tan buenos debido a que nos devuelve en texto todo los datos de las columnas que referencián y la tabla que es referenciada con sus respectivos columnas. Haremos uso de las mismas técnicas que aplicamos al momento de realizar el ordenamiento de las tablas según el orden que deben ser llenados que al final necesitamos tener un resultado similar a la siguiente tabla.

Cuadro 4.3: tabla referencias formateada

tabla que referencia	**columnas que referencian**	**tabla referenciada**	**columnas referenciadas**
detalle	cod_producto	producto	cod_producto
detalle	cod_cliente, cod_venta, cond_vendedor	venta	cod_cliente, cod_venta, cond_vendedor

De la Figura 4.16 tomamos la columna 3 y la fila 2 como ejemplo escogemos la fila 2 por razones didácticas, es donde se encuentra la información a en modo texto:

```
1 "FOREIGN KEY (cod_cliente, cod_venta, cond_vendedor) REFERENCES
     venta(cod_cliente, cod_venta, cond_vendedor)"
```

La cadena de texto lo separamos en 5 partes:
Lista en 5 partes

1. `FOREIGN KEY`
2. `cod_cliente, cod_venta, cond_vendedor`
3. `REFERENCES venta`
4. x`REFERENCES venta`
5. ...

Para obtener esta lista separada llevamos a un arreglo la cadena de texto teniendo como separadores a "(", ")" . La mayoría de los lenguajes de programación ya nos proveen funciones que realicen esta tarea de llevar una cadena de texto a un arreglo con solo indicando los caracteres separadores.
En las la mayoría de los lenguajes de programación el conteo de las posiciones se inicia en 0 por tanto vamos basarnos en esa regla, del arreglo solo necesitamos el de la posición 1 es donde se encuentra las columnas que hacen referencia en conjunto a la tabla de la posición 2 de arreglo sin antes aclarar que este elemento de la posición debe ser separado en dos partes:
Lista en 2 partes

1. `REFERENCES`

2. venta.

De esta lista solo nos es útil el de la posición 1 es donde encontramos en nombre de la tabla al que se hace referencia.
Si volvemos a la lista que separamos en 5 partes el otro elemento útil es de la posición 3, es donde encontramos las columnas referenciadas.
Realizamos este procedimiento por cada relación que haga la tabla obteniendo así un resultado similar a la tabla del cuadro 4.3. Con los resultados obtenidos de la Figura 4.15 y el de la Figura tabla formateada del cuadro 4.3 realizamos la union de estos dos resultados para tener una tabla como se ve en el cuadro 4.2.
Para obtener un resultado del cuadro 4.2 es necesario agregar campos al resultado que nos provee la Figura 4.15, agregamos cuatro campos adicionales:

- *es_foranea* En esta columna podemos agregar si es foránea o no para luego ser evaluado como tal.
- *referencian* En esta columna agregamos los nombres de las columnas que hacen referencia a otra tabla.
- *tabla* En esta columna agregamos el nombre de la tabla al que referencia.
- *referenciados* En esta columna agregamos los nombres de las columnas que son referenciados.

Quedando una tabla de la siguiente manera

Cuadro 4.4: tabla con columnas aumentadas para la tabla *detalle*

column_name	data_type	constraint_type	.....	es_foranea	columnas referencian	tabla referenciada	columnas referenciadas
cod_detalle	character varying	PRIMARY_KEY	.....				
cod_venta	INTEGER	PRIMARY_KEY					
cod_cliente	INTEGER	PRIMARY_KEY					
cod_vendedor	INTEGER	PRIMARY_KEY					
cod_producto	INTEGER	PRIMARY_KEY					

Si rocordamos la cadena de texto

```
1 "FOREIGN KEY (cod_cliente, cod_venta, cond_vendedor) REFERENCES
     venta(cod_cliente, cod_venta, cond_vendedor)
2 FOREIGN KEY (cod_producto) REFERENCES producto(cod_producto)"
```

que lo llevamos en un arreglo de 5 elementos, el elemento de la posición 1 que llega a ser

```
1 "cod_cliente, cod_venta, cond_vendedor"
```

Es donde encontramos las columnas que hacen referencia por lo tanto a esta cadena necesitamos también llevarlo a un arreglo lineal donde el carácter separador llega a ser el "," quedando como resultado.

Cuadro 4.5: columnas de la tabla detalle que referencian a venta

nombre columna	cod_cliente	cod_venta	cond_vendedor
posicion	0	1	2

Cuadro 4.6: columna de la tabla detalle que referencia a producto

nombre columna	cod_producto
posición	0

Los datos del cuadro 4.5 y 4.6 son las que hacen referencia a otra tabla para unir las llaves foráneas y llegar a un resultado como en el cuadro 4.2 para lo cual realizaremos el siguiente procedimiento.

1. Realizar la unión en un arreglo único los elementos del cuadro 4.5 y 4.6 llamemosle *referencian* y nos declaramos dos variables denominemosle *pos* y *posTabla* declarada con valor inicial de 0 para controlar la posición del nuevo arreglo creado.

2. Obtener el elemento de la la posición *pos* y comparar con el elemento de la posición *posTabla* del cuadro de 4.4 y comparamos si son iguales.

3. Si llegan a ser iguales es porque este campo del cuadro 4.4 es un campo que hace referencia a otra entidad por lo tanto agregamos el valor de *true* en su columna *es_foranea* e incrementar el valor de pos en una unidad y volver al paso 2 asignar un valor de 0 a la variable *posTabla*, de lo contrario pasar al siguiente paso.

4. Es caso de que no sean iguales esta claro que este atributo no hace referencia a ninguna otra tabla y lo agregamos con un valor de *false* y volvemos al paso 2 e incrementar el valor de la variable *posTabla*.

Llegado a un resultado como en el siguiente cuadro:
ESTE CUADRO FALTA REEMPLAZAR POR UNO QUE SERIA COMO QUEDARÍA

Cuadro 4.7: tabla de muestra de atributos foráneas

nombre columna	cod_producto
posición	0

En el cuadro 4.7 ya se tiene claro que atributos que no son propias y que dependen de la existencia de registros en la tabla a la que referencia, así como esta no es como queremos el siguiente procedimiento a realizar es eliminar estos atributos y reemplazarlos por los que tenemos en el cuadro 4.3, para realizar este reemplazo necesitamos tener una tabla con los mismo atributos *(tabla_name , data_type, check_clause ... ,referencian,tabla,referenciados)*. Eliminando los atribustos que tengan el valor de *true* en la columna *es_foranea* llegamos a tener como resultado como el siguiente cuadro
ESTE CUADRO FALTA REEMPLAZAR POR UNO QUE SERIA COMO REALMENTE SE QUIERE

Cuadro 4.8: tabla sin los atributos foráneas

nombre columna	cod_producto
posición	0

Para obtener la otra tabla con solo de llaves foráneas aumentamos los campos que llevan a ser similar a la tabla del cuadro 4.8 realizamos las siguientes operaciones:

1. Crear un arreglo llamemosle *clon* con las mismas dimensiones que del cuadro 4.4 y declaramos una variable llamemosle *indice* que hara el control de la pocisión

2. Obtenemos la fila de la posición *indice* del cuadro 4.7 y verificamos el valor de la columna *es_foranea* en caso que sea *true* pasamos a 3 y si no hacemos un salto al 4.

3. A esta fila no lo hacemos la copia en *clon* porque este atributo no es propia de la tabla e incrementamos en una unidad a *índice*.

4. Realizamos la copia en *clon* e incrementamos en una unidad a *indice*.

5. Si no hay mas elementos que comparar pasamos al siguiente de lo contrario volvemos a 2.

6. Al realizar los pasos anteriores el resultado obtenido será todas los atributos que son propias de la tabla, al lo cual debemos completar con las restantes que son dependientes de otras tablas pero en una forma diferente,para lo cual tomamos el de la posición 1 2 y 3 del arreglo separado en 5 partes que como resultado final se tiene en el cuadro 4.8.

Capítulo 5

Crear proyecto de configuración

Cuando realizamos el llenado de datos de prueba sobre una base de datos, hay un inicio y un final donde no siempre iniciamos y acabamos sin alguna interrupción, por muchas razones fallas eléctricas, cansancio entre otras para lo cual es importante que toda la información obtenida en el capitulo anterior sea persistente y que podamos tener esa información sin volver a ejecutar los algoritmos además sin la necesitada de volver a conectarnos a la base de datos. Para lo cual almacenamos en archivos de texto plano con algún formato, entre los formatos mas conocidos tenemos a XML (Extensible Markup Language o Lenguaje de Marcas Extensible) y JSON (JavaScript Object Notation - Notación de Objetos de JavaScript).

5.1. JSON (JavaScript Object Notation - Notación de Objetos de JavaScript)

Es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, Standard ECMA-262 3rd Edition - Diciembre 1999. JSON es un formato de texto que es completamente independiente del lenguaje pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos. JSON est constituido por dos

estructuras:

- Una colección de pares de nombre/valor. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
- Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras. En JSON, se presentan de estas formas:
Un objeto es un conjunto desordenado de pares nombre/valor. Un objeto comienza con (llave de apertura) y termine con (llave de cierre). Cada nombre es seguido por : (dos puntos) y los pares nombre/valor estn separados por , (coma).

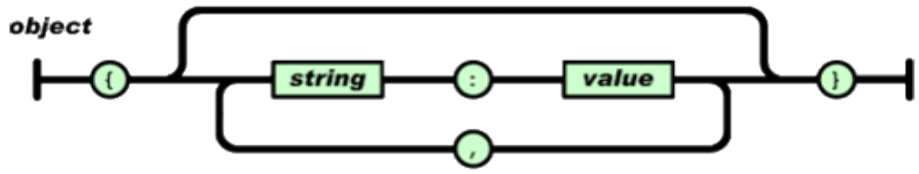


Figura 5.1: Object JSON

Un arreglo es una colección de valores. Un arreglo comienza con [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por , (coma).

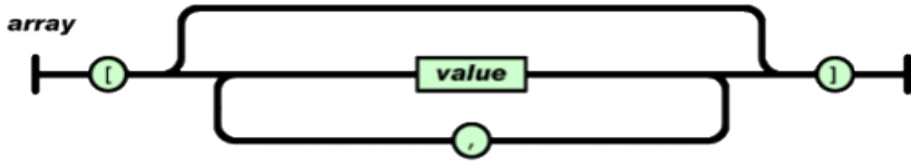


Figura 5.2: Array JSON

Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un arreglo. Estas estructuras pueden anidar.

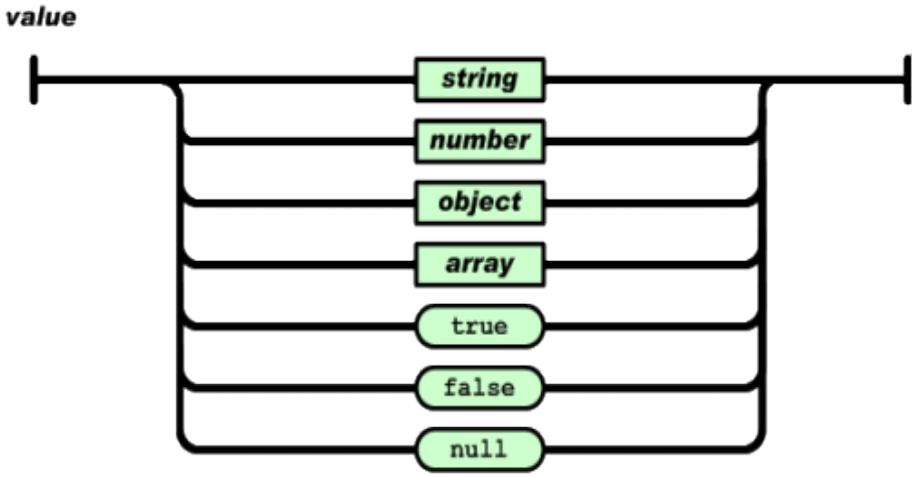


Figura 5.3: Value JSON

Una cadena de caracteres es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter está representado por una cadena de caracteres de un único carácter. Una cadena de caracteres es parecida a una cadena de caracteres C o Java.

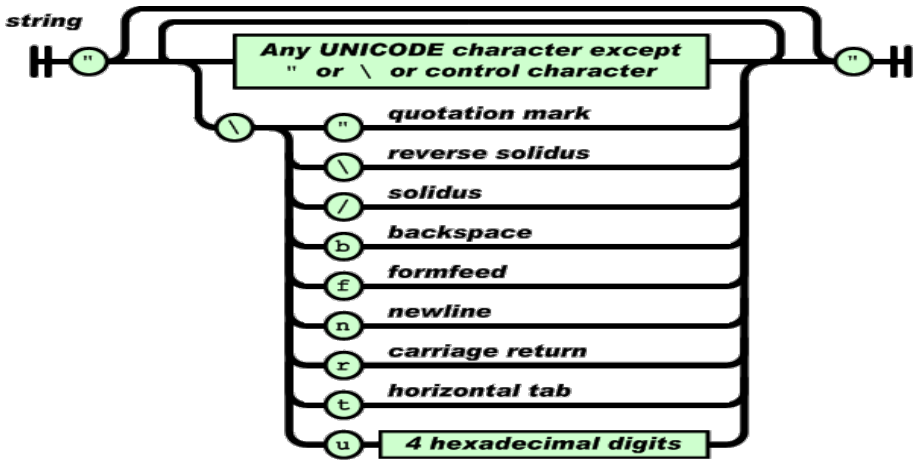


Figura 5.4: String JSON

Un número es similar a un número C o Java, excepto que no se usan los formatos octales y hexadecimales.

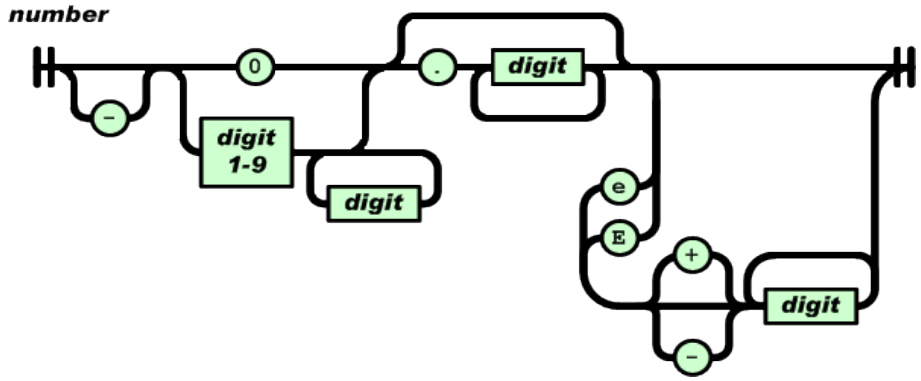


Figura 5.5: Number JSON

Los espacios en blanco pueden insertarse entre cualquier par de símbolos.

5.2. Persistencia de la información de metadatos

La información obtenida en el anterior capitulo es necesario que sean persistentes para la reanudación en el proceso de configuración para el objetivo, la información necesaria a persistir son las siguientes:

- Datos de la conexión a la base de datos como ser el nombre de la base de datos, usuario, contraseña, puerto y el host.
- La lista de las tablas de la base de datos elegida según al orden en que estos deben ser llenados que obtuvimos en el capitulo anterior.
- El detalle por cada una de las tablas (nombre de la columna, el tipo de dato, si acepta que sea nulo, si es una llave etc...).

Una alternativa para dar solución a este requisito es almacenar toda esta información en archivos sea en el formato JSON o XML, en este proyecto se optará por Json las razones son las siguientes:

- Soporta dos tipos de estructuras, una de ellas son objetos que contienen una colección de pares llave-valor y el otro tipo se trata de arrays de valores. Esto proporciona una gran sencillez en las estructuras.

- No tiene espacios de nombres, cada objeto es un conjunto de claves independientes de cualquier otro objeto.

- JSON no necesita ser extensible por que es flexible por sí solo. Puede representar cualquier estructura de datos pudiendo añadir nuevos campos con total facilidad.

- Es mucho mas simple que XML, el cual proporciona pesadas tecnologías que le avalan (Scheme, XSLT, XPath).

- Si comparamos el tamaño de un archivo JSON con uno XML y que contenga la misma información el primero llega a ser mucho mas pequeño.

5.2.1. Creando la estructura de un proyecto

Cuando sea crea un proyecto java ,php , python etc... normalmente se tiene un estructura de directorios y archivos, donde ciertos archivos guardan configuraciones sobre recursos que se hacen uso, la version del proyecto entre otros. Para este proyecto realizaremos algo similar para lo cual se va tener como base la siguiente estructura de directorios y archivos.

Donde se guarda la información de los archivos de configuración, veamos en detalle cada directorio y su contenido:

- **conexión** En este directorio se tiene un archivo *conexion.xml* la cual contiene la información de los datos de conexión. Va separada en un directorio por razones de que puede existir un nombre de una tabla igual al archivo por lo que es necesario que no se confunda.

- **control** Si observamos el contenido de los directorios *control,dates,tables* son similares la diferencia esta en el contenido de los archivos.

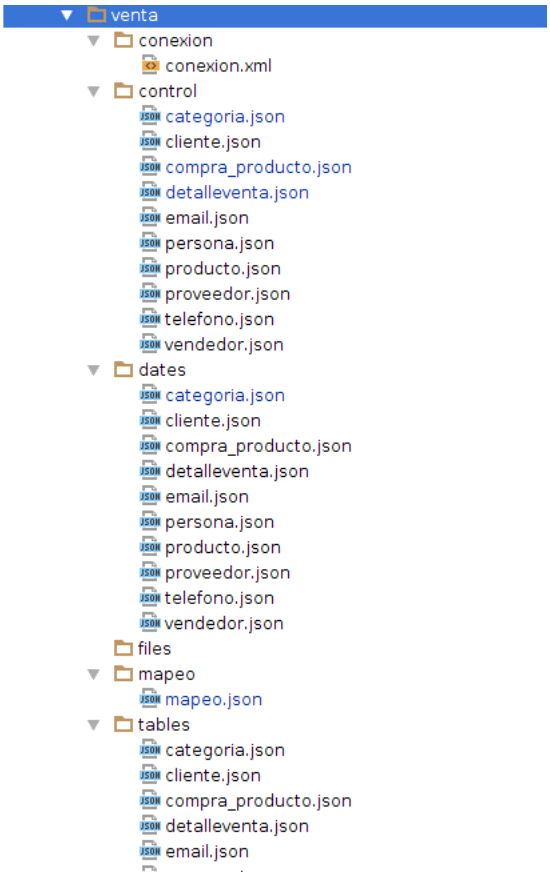


Figura 5.6: Estructura

En este directorio almacenamos archivos de control de las columnas por cada tabla y que tienen el mismo nombre que en la base de datos veamos el contenido para la tabla *compra_producto* de la Figura 2.1.

Listing 5.1: Ejemplo archivo control

```
[
  {
    "column_name": "cod_producto",
    "is_nullable": "NO",
    "rellenado": false
  },
  {
    "column_name": "cod_proveedor",
    "is_nullable": "NO",
    "rellenado": false
  },
  {
    "column_name": "cod_compra_producto",
    "is_nullable": "NO",
    "rellenado": false
  },
  {
    "column_name": "fecha_compra_producto",
    "is_nullable": "NO",
    "rellenado": false
  }
]
```

.

Encontramos información en formato JSON clave - valor y donde *column_name* indica el nombre de la columna, *is_nullable* nos indica si este campo puede ser nulo y por ultimo *rellenado* llega a ser la mas importante porque es aquí donde controlamos si ya fue configurada esta columna.

- **dates** los archivos de este directorio almacenan información generada por cada columna, a excepción de tipos de datos como bytea o blob, para este tipo de datos es recomendable almacenar el nombre del archivo. El formato del archivo que almacena la información generada es el siguiente:

Listing 5.2: Ejemplo archivo control

```
[
  {
    "nombre_categoria": "bebida",
    "cod_categoria": "1"
  },
  {
    "nombre_categoria": "comidarapida",
    "cod_categoria": 2
  },
  {
    "nombre_categoria": "enlatados",
    "cod_categoria": 3
  },
  {
    "nombre_categoria": "especial",
    "cod_categoria": 4
  },
  {
    "nombre_categoria": "ensaladas",
    "cod_categoria": 5
  }
]
```

Si observamos la Figura 2.1 la tabla *categoria* tiene dos atributos *nombre_categoria* y *cod_categoria*, si vemos el contenido del archivo *categoria.json* de directorio *dates* existen varios registros con los valores asignados, la cantidad puede variar dependiendo de la cantidad de datos que se quiere generar.

- **files** En este directorio se almacenan los archivos de tipo bytea y que al momento de hacer la insercion las usamos por su nombre.

- **mapeo** Solo existe un archivo con un contenido de la lista de las tablas según el orden en que estas deben ser configurados además tiene dos atributos mas *nivel* la cual nos indica cual es el orden en que le corresponde y por ultimo la *cantidad* si encontramos un valor igual a cero es porque esta tabla no tiene columna alguna configurada esto nos deja entender que si se da un valor, es para la tabla en general. Veamos para la el caso de la Figura 2.1.

Listing 5.3: Ejemplo archivo control

```
[
  {
    "tablename": "categoria",
    "nivel": 0,
    "cantidad": "5"
  },
  {
    "tablename": "persona",
    "nivel": 0,
    "cantidad": 0
  },
  {
    "tablename": "producto",
    "nivel": 1,
    "cantidad": 0
  },
  .
  .
  .
  {
    "tablename": "compra_producto",
    "nivel": 2,
    "cantidad": 0
  },
  {
    "tablename": "detalleventa",
    "nivel": 2,
    "cantidad": 0
  }
]
```

- **tables** En el directorio tables es donde almacenamos la información detallada por cada una de las tablas, además cada archivo representa a una tabla de la base de datos y que llevan el mismo nombre. Veamos el contenido del archivo *categoria.json* que representa a la tabla *categoria*:

Listing 5.4: Ejemplo archivo control

```
[
  {
    "column_name": "cod_categoria",
    "data_type": "integer",
    "character_maximum_length": null,
    "es_foranea": "false",
    "referencian": null,
    "tabla": null,
    "referenciados": null,
```

```
10      "numeric_precision": "32",
11      "is_nullable": "NO",
12      "constraint_type": "PRIMARY KEY",
13      "column_default": "nextval('categoria_cod_categoria_seq'::
        regclass)",
14      "check_clause": null
15    },
16    {
17      "column_name": "nombre_categoria",
18      "data_type": "character varying",
19      "character_maximum_length": null,
20      "es_foranea": "false",
21      "referencian": null,
22      "tabla": null,
23      "referenciados": null,
24      "numeric_precision": null,
25      "is_nullable": "NO",
26      "constraint_type": null,
27      "column_default": null,
28      "check_clause": null
29    }
30  ]
```

Es una estructura de directorios que no necesariamente se tienen que llamar así, sin embargo el nombre de los archivos es aconsejable que lleven el mismo nombre que en la base de datos para que resulte mas amigable.

5.3. configuración de columnas

Una vez que se tiene el proyecto creado, la configuración que se hace es por cada columna de la tabla para lo cual necesitamos saber que tipo de dato acepta cada columna o si hace referencia a otra tabla. Las columnas de una tabla puede ser de diferente tipo de dato(*integer, varchar, boolean etc....* Independientemente del tipo de dato de una columna se puede agruparlos tomando en cuenta ciertas características como ser:

- Que el tipo de dato sea texto, fechas hora, direcciones de red al momento de insertar a la base de datos estos son tomados como si fuesen un texto(*'valor'*).
- El tipo de dato sea un numero en las que estan (*integer,serial,smallserial,bigserial, beint ...smallint*) están son insertados como un

numero (*valor*).

- Que sea una llave primaria sin importar el tipo de dato están deben ser únicas al momento de generarlos.
- Cuando sea una llave foránea no se debe generar por que estas deben existir en la tabla que hace referencia para lo cual se toma los valores generados en la tabla referenciada.
- El tipo de dato sea bytea es un caso especial que no se trata como una cadena ni como un numero.

5.3.1. Configuración para llaves foráneas

Las llaves foráneas o primarias que no son propias de la tabla no se necesitan generarlos con un algoritmo, lo que se hace es trabajar con los datos generados en la columna de la tabla al que se hace referencia, En este proyecto la técnica empleada para el manejo de estas fue agruparlos todas las que hacen referencia en conjunto a alguna tabla como si fuese una sola columna veamos un ejemplo.

.

En la Figura 5.7 la *tabla1* cuenta con una llave primaria (*T1C1*) al igual que la *tabla2*, y si bajamos un nivel mas abajo a la *tabla3* esta hace referencia a *tabla1* y *tabla2* y que las columnas de las tablas referenciadas mandan como llaves primarias, formando asi una llave compuesta para *tabla3*.Por otro lado la *tabla4* hace referencia a *tabla2* y que tambie tiene una llave compuesta. Si bajamos a la *tabla5* esta compuesta de 4 columnas de las cuales 3 no son propias, si nos fijamos vienen juntadas como si fuera una columna es lo que hicimos al momento de guardar el detalle de una tabla, En la *tabla7* se hace referencia a la *tabla5* y *tabla6* y las columnas que no son propias son agrupadas de acuerdo a que tabla se haga referencia es el caso de *T7C2,T7C3,T7C4,T7C5* que en conjunto hacen referencia a la *tabla5* y *T7C6,T7C7,T7C8* hacen referencia a *tabla6*.
Al momento de hacer la configuración llegamos a tener un problema, de la *tabla5* el campo *T5C2,T5C3,T5C4* no lo encontramos si lo buscamos como una columna en

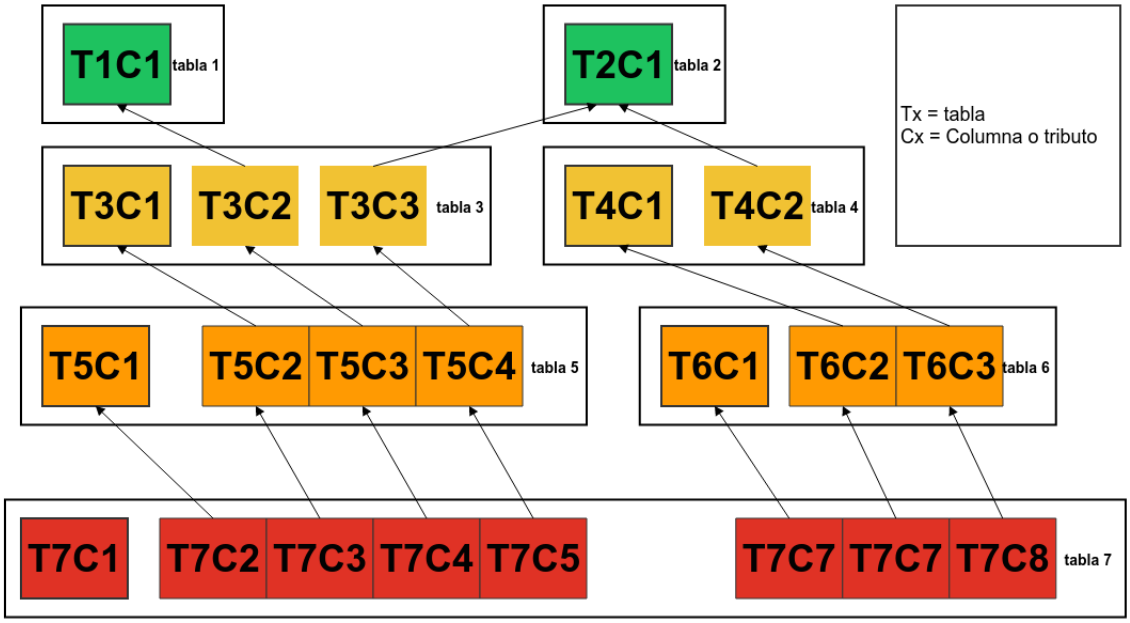


Figura 5.7: foraneasline

la *tabla3* como es posible si tenemos esas columnas? Es cierto que existen pero están separadas.

.

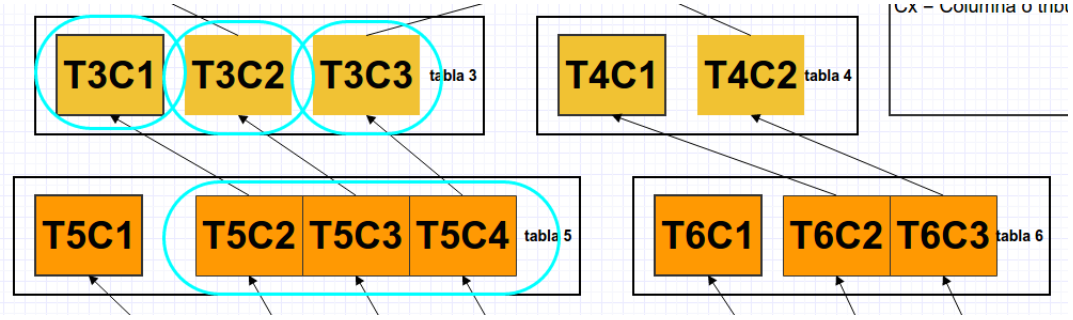


Figura 5.8: Problema llaves foraneas

Como se observa en la Figura 5.8 se da el mismo problema para la *tabla7* la columna *T7C2, T7C3, T7C4, T7C5* no se encuentra en una solo columna en la *tabla5* pero hay algo interesante que se puede deducir veamos la siguiente figura:

.

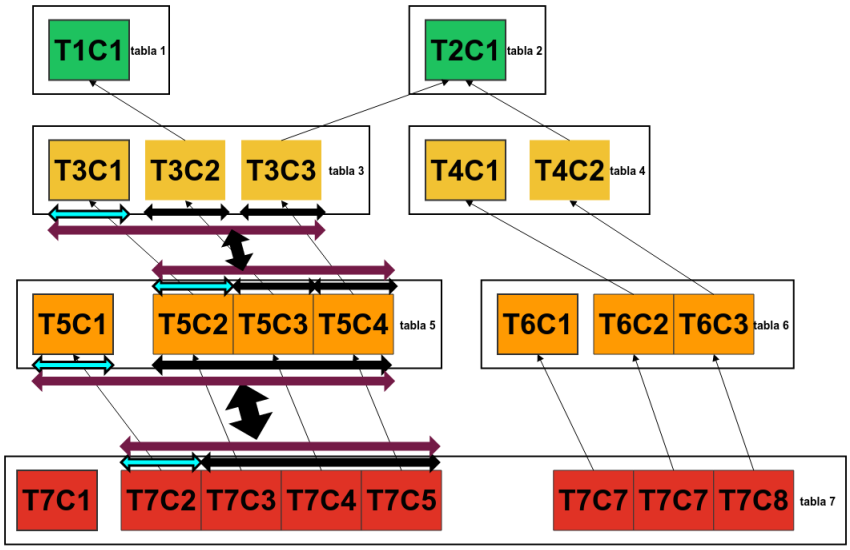


Figura 5.9: Ejemplo foraneas unidas

En la *tabla5* las columnas *T5C2,T5C3,T5C4* en conjunto hacen referencia como una sola a la *tabla3* donde *T5C2* apunta a *T3C1* y que esta es una llave primaria propia de la *tabla3*, si vamos a la *tabla7* la columna *T7C2* hace referencia a una propia de la *tabla5*, Podemos determinar que el comportamiento de las llaves primarias que no son propias siguen este modelo.

Problemas

El problema surge al momento de realizar la validación que por ejemplo cuando por alguna razón tratamos de configurar un atributo que hace referencia y que previamente no se configurón los atributos referenciados en la tabla referenciada no se llega a encontrar como se ve en la figura 5.9. Las columnas de la *tabla3* se van encontrar todas al igual que en la *tabla4*, pero en la *tabla5* no se llega a encontrar la columna *T5C2T5C3T5C4* lo mismo sucede con *T6C2T6C3*. Una vez hecha la validación el siguiente paso es configurar, para lo cual se necesita los datos de la tabla referenciada, y nos fijamos en la Figura 5.9 nos encontramos en el mismo problema de la validación de columnas no encontradas.

Soluciónes

Una solución obvia al problema de la validación presentado es verificar que todas las columnas de la tabla referenciada se encuentren configuradas para no tener problemas al obtener los datos pero no es tan cierto se puede configurar con solo tener configuradas las columnas referenciadas podemos optar por cualquiera son cuestiones de validación.
En cuanto al problema de la configuración una vez pasada la anterior la solución no llega a ser tan sencilla necesitamos aplicar algun mecanismo(os) de obtener los datos, La solución que damos no estrictamente así depende del modelo y del manejo de llaves.
.

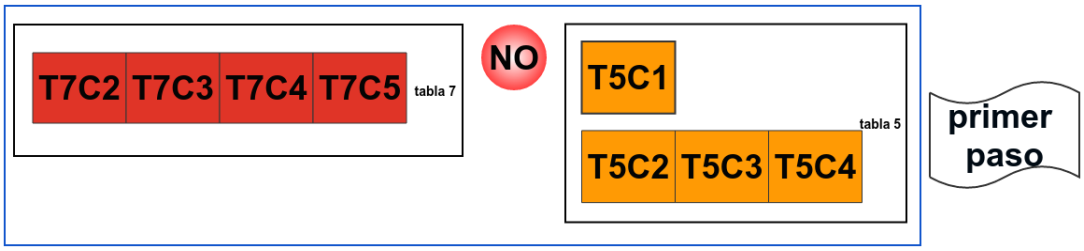


Figura 5.10: intento 1

En el intento 1 no llegamos a la solución ya que no encontramos la columna en la *tabla5*.
.

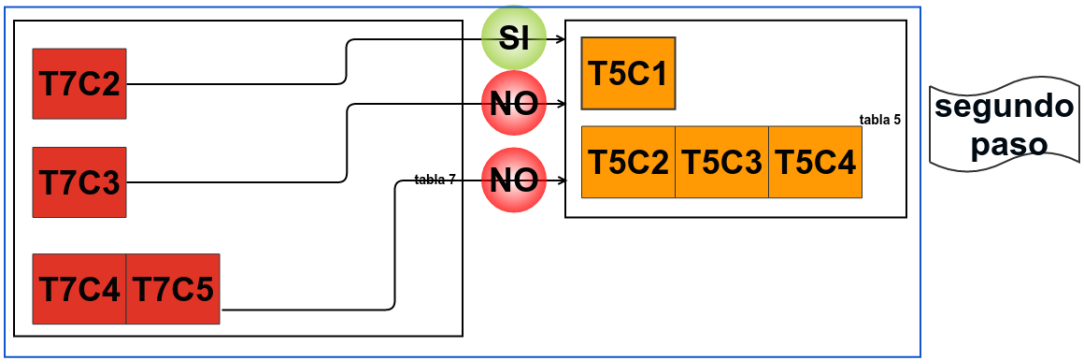


Figura 5.11: intento 2

Iniciamos con el primer elemento buscando en la tabla referenciada en caso de que exista pasamos al siguiente elemento caso contrario listamos en una lista de los elementos no encontrados.

.

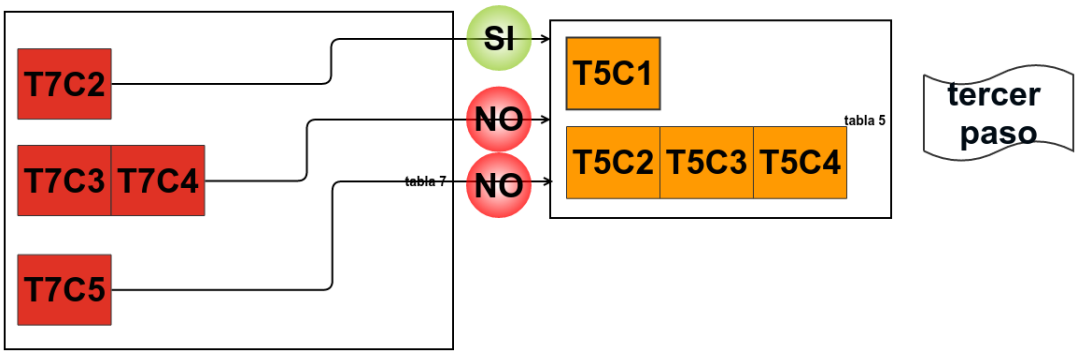


Figura 5.12: intento 3

Pasamos con el siguiente elemento uniendo con las no encontradas y volvemos a buscar en la tabla referenciada como aun no encontramos pasamos al siguiente.

.

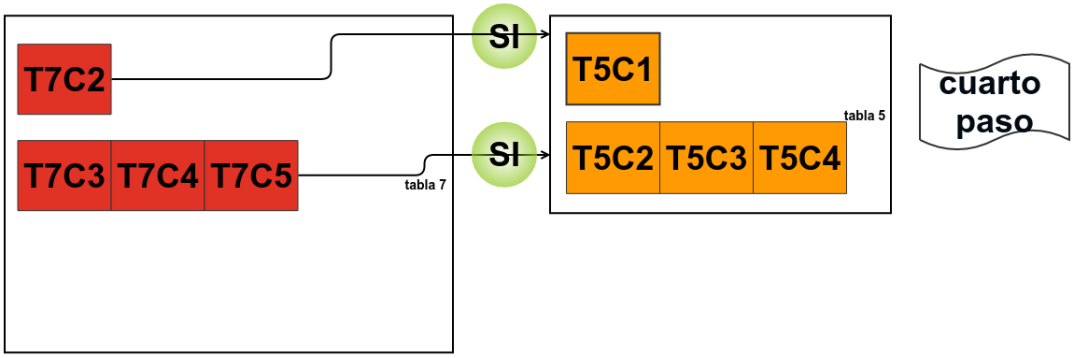


Figura 5.13: intento 4

En este paso volvemos a juntar las no encontradas y el elemento a obtener y buscamos en la tabla referenciada y como encontramos y no hay mas elementos que buscar se llega a la solución.

Capítulo 6

Poblando datos en la base de datos y probando el comportamiento

Una vez que se tiene configurada en forma completa un proyecto, el siguiente proceso a realizar es el poblado de datos y posteriormente realizar algunas consultas de prueba para ver el comportamiento con una población de datos mayor.Para llevar adelante este objetivo es necesario tener la siguiente información.

- Recuperar datos de conexión para la base de datos y realizamos la conexión.
- Obtenemos los datos de archivo *mapeo.json* donde se encuentra en orden de las tablas.
- Por cada tabla creamos la estructura SQL de inserción `INSERT INTO tabla (col1,col2...coln)VALUES (val1,val2...valn)`.

  Existe casos muy importantes al crear la estructura de datos, la cantidad de columnas que tiene una tabla, el tipo de dato de cada columna, al momento de hacer la inserción es importante tomar en cuenta estos datos , casualmente que ciertos tipos de datos se insertan de una manera y otras de cierta manera.

6.1. Poblado de datos a la base de datos

Los distintos tipos de datos que nos provee los DBMS, algunos con una cantidad mayor de tipos de dato y otras con una cantidad mas reducida, es importante analizar

como lo insertamos según al tipo de dato, además se debe tomar encuenta la cantidad de columnas que tiene una tabla.

6.1.1. Tipos de datos tratados como texto

Los tipos de datos tratados como cadenas de texto son:

- Las fechas y horas `(DATE,DATETIME,TIME)`.
- Las cadenas de texto `(VARCHAR,CHARACTER VARYIN,TEXT)`.
- Las direcciones de red `(MACADDRESS,INET)`.

Estos tipos de datos van entre comillas simples(`INTO tabla(col)VALUES('col')`).

6.1.2. Tipos de datos tratados como números

Los tipos de datos son tratado como un numero entero sea decimal flotante son los tipos de dato como:

- Los tipos enteros `(INTEGER, BEGINT, SMALLINT, SERIAL, BIGSERIAL)`.
- Los tipos decimales `(FLOAT, DECIMAL MONEY)`.

Los tipos de datos numéricos a diferencia del anterior no van entre comillas(`INTO tabla(col)VALUES(val)`). Existe otro tipo de dato mas que podemos incorporarlo es el tipo `BOOLEANO` si bien no es un numero esta no necesita ir dentro las comillas.

6.1.3. Tipo de dato bytea

El tipo de dato bytea es un tipo especial ya que nececita una conversion previa a la inserción, las distintas tecnologias ya se php, java , python , ruby etc... proveen metodos para realizar esta conversión, por lo cual no es un tema de preocupación. Si recordamos al momento de generar los datos el tipo de dato bytea no lo guardamos en el archivo generado solo el nombre del archivo, con la que formamos un codigo SQL de insercion de la siguente manera (`INTO tabla(col)VALUES(conversionprevia(nombre archivo))`).

6.1.4. Cantidad de columnas por tabla

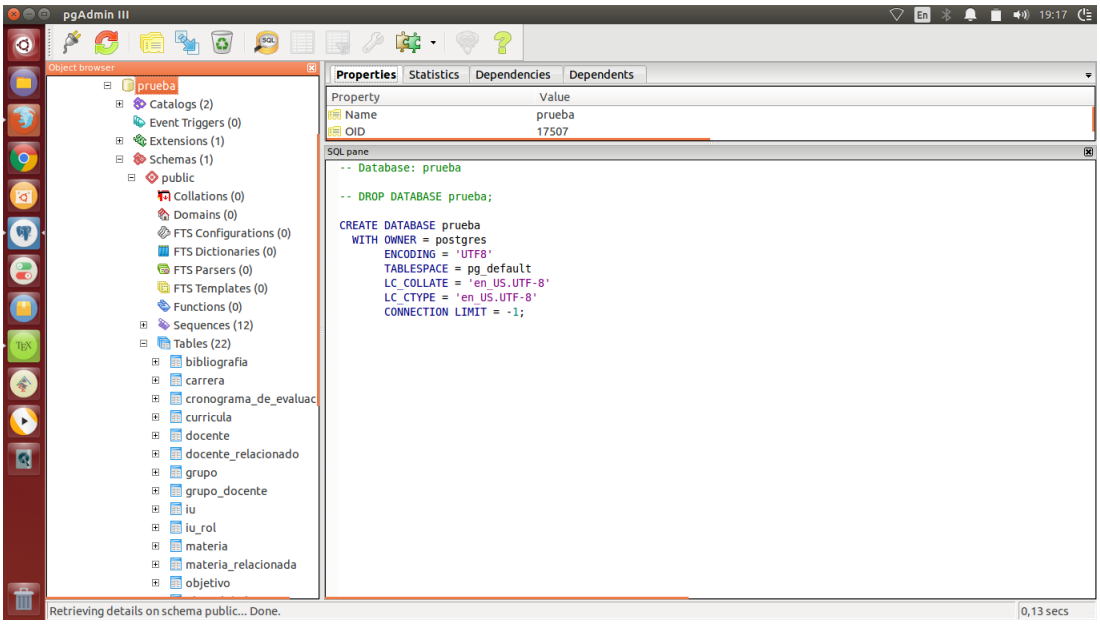
La cantidad de columnas de una tabla es variable, pudiendo tener una o mas columnas para formar la estructura del comando `INSERT` necesitamos obtener la información del directorio *tables*. Donde cada tabla es representada por un archivo con el mismo nombre de la tabla y que además esta contiene toda la información detallada de la tabla, entre ellas esta los nombre de las columnas. Con esta informacion formamos la parte necesaria del comando `INSERT INTO tabla(col1, col2,...coln)VALUES()`. En la parte de los valores obtenemos informacion del directorio *dates del proyecto* y referencia [3]

Capítulo 7

uso del prototipo

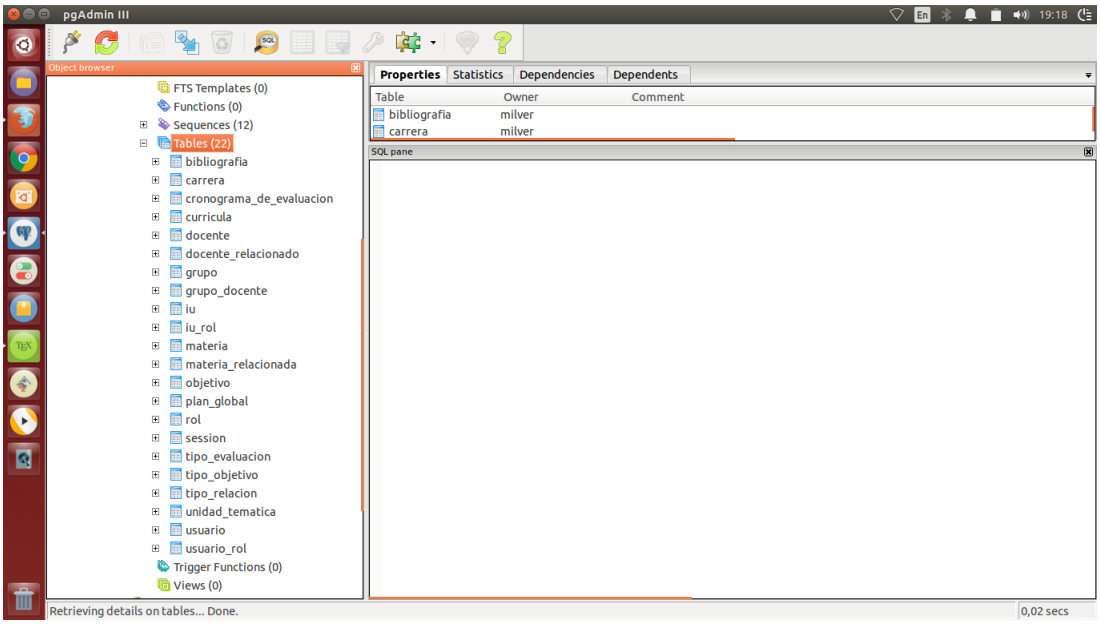
En este capítulo hacemos uso del prototipo desarrollado para el poblado de datos de prueba para una base de datos, para lo cual es necesario que se tenga cantidad de tablas. Como ejemplo tomamos una base de datos denominada prueba como observamos en la Figura 7.1

Figura 7.1: base de datos prueba



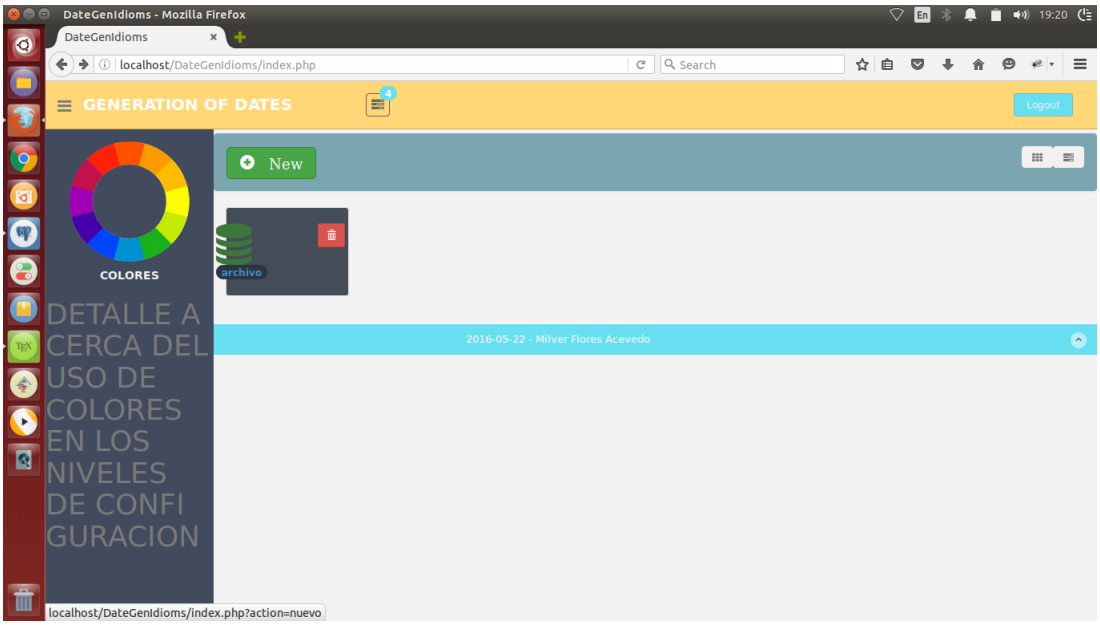
Esta base de datos se tiene veinte y dos tablas como se observa en la Figura 7.2.

Figura 7.2: base de datos prueba



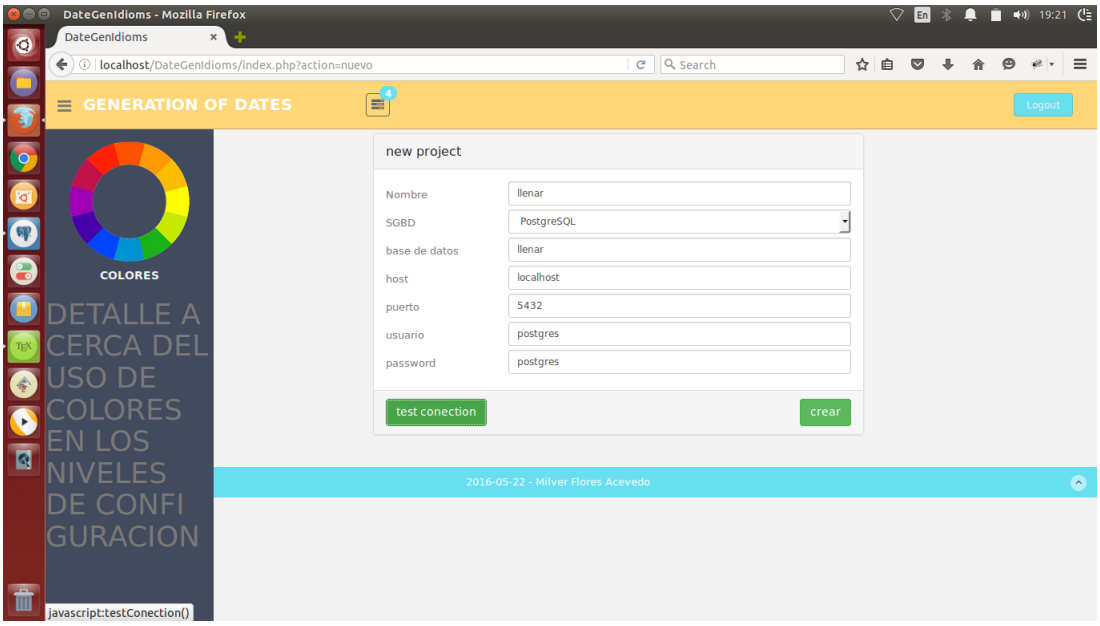
En el prototipo se tiene la opción de crear como un proyecto en el boton nuevo y la lista de proyectos como se ve en la Figura 7.3.

Figura 7.3: lista de proyectos



Al hacer clic nos llevara a un formulario como se ve en la Figura 7.4

Figura 7.4: formulario para crear un nuevo proyecto



Para crear un proyecto es necesario llenar los datos en el formulario que son:

Nombre este campo es a eleccion con la restricción que no se puede tener dos proyectos con el mismo nombre.

sgbd el sistema gestor de base de datos que en este trabajo elegimos trabajar con PostgreSQL.

base de datos en este campo es necesario el nombre exacto de la base de datos por que sera de la cual obtendremos su estructura.

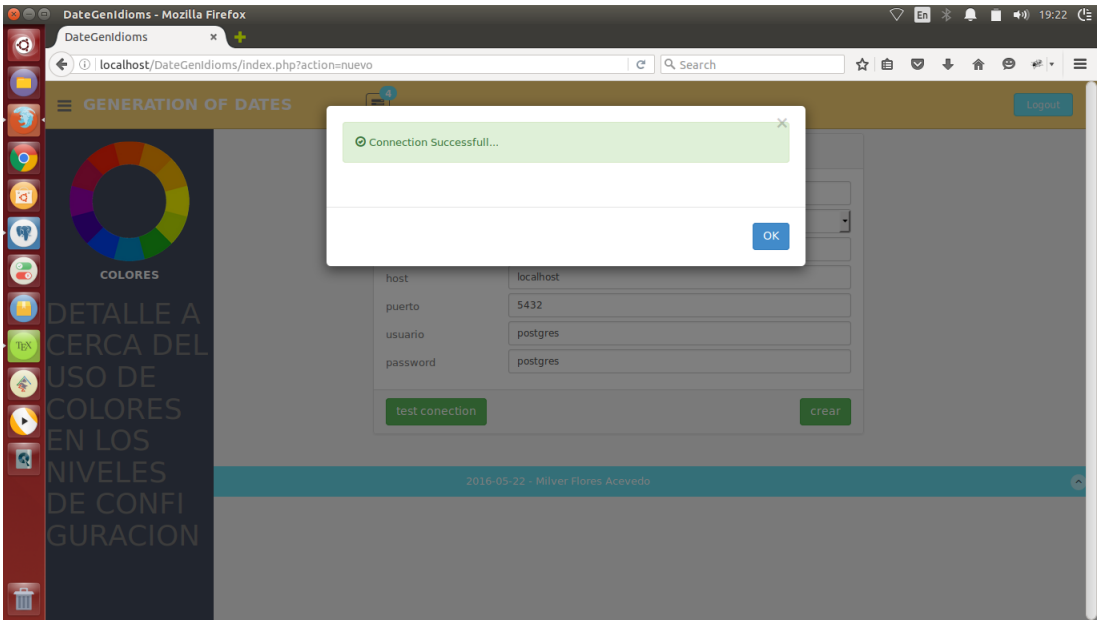
host la url donde se encuentra alojada la base de datos. En nuestro caso localhost.

puerto normalmente el puerto que usa PostgreSQL es 5432.

usuario con el usuario que se conectara con previligios de acceso a metadatos.

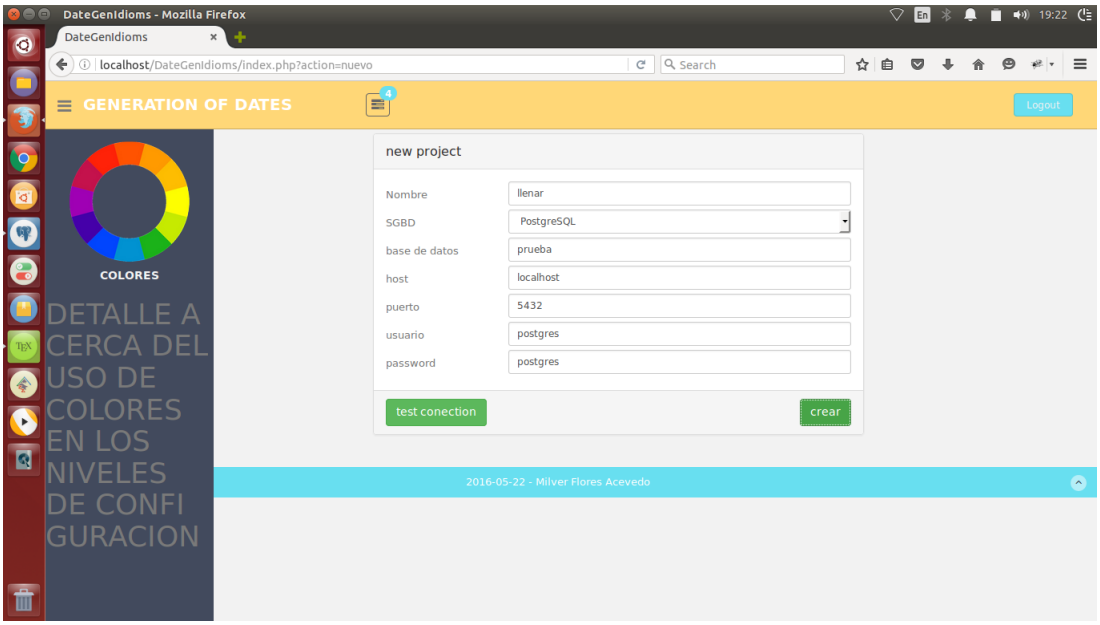
password la contraseña del usuario

Figura 7.5: conexion exitosa



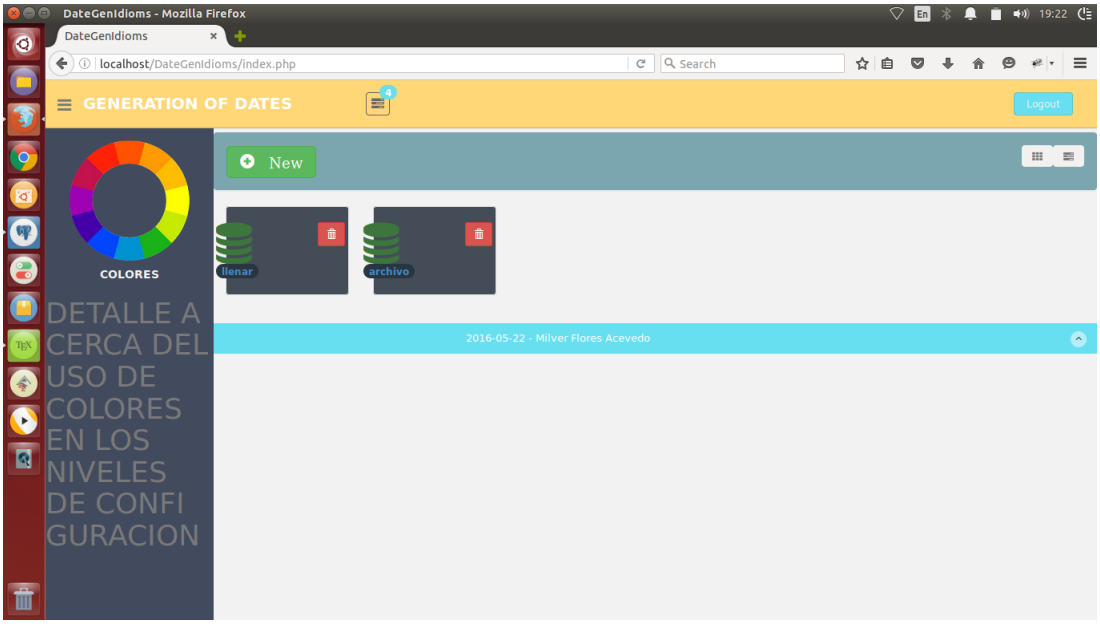
Una vez llenada el formulario lo siguente es probar la conexion dando clic en el boton de test connection, si es exitosa muestra el mensaje de una conexion exitosa.

Figura 7.6: boton crear



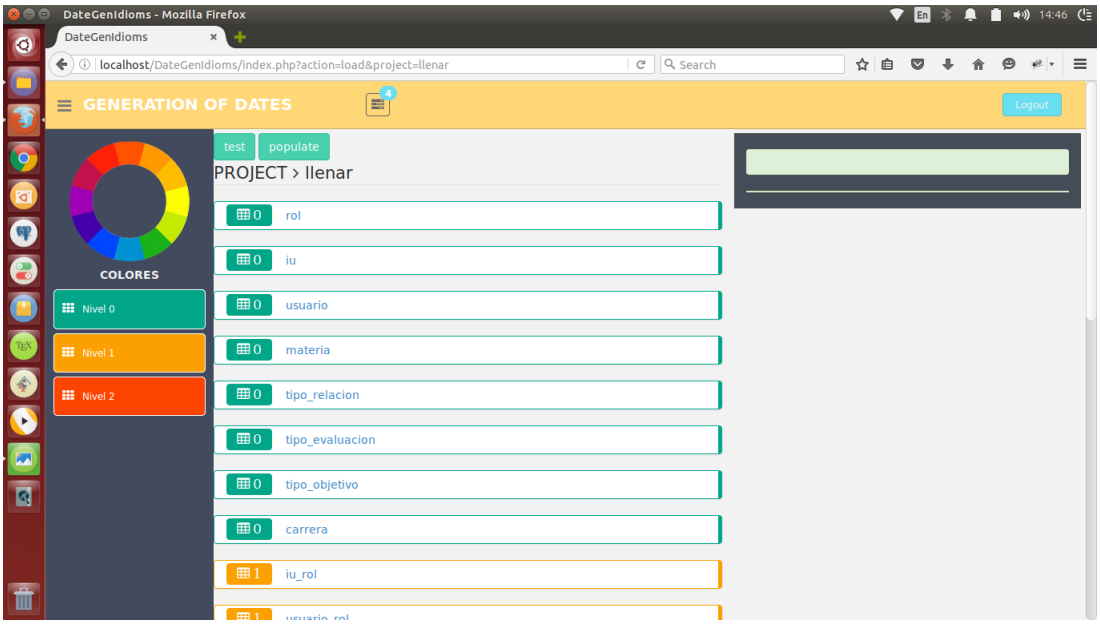
Si los datos del formulario son correctas damos clic en el boton crear ver Figura 7.6, a continuacióin nos redirigirá a la lista de proyectos en la cual aparecera el proyeto que creamos.

Figura 7.7: proyecto creado



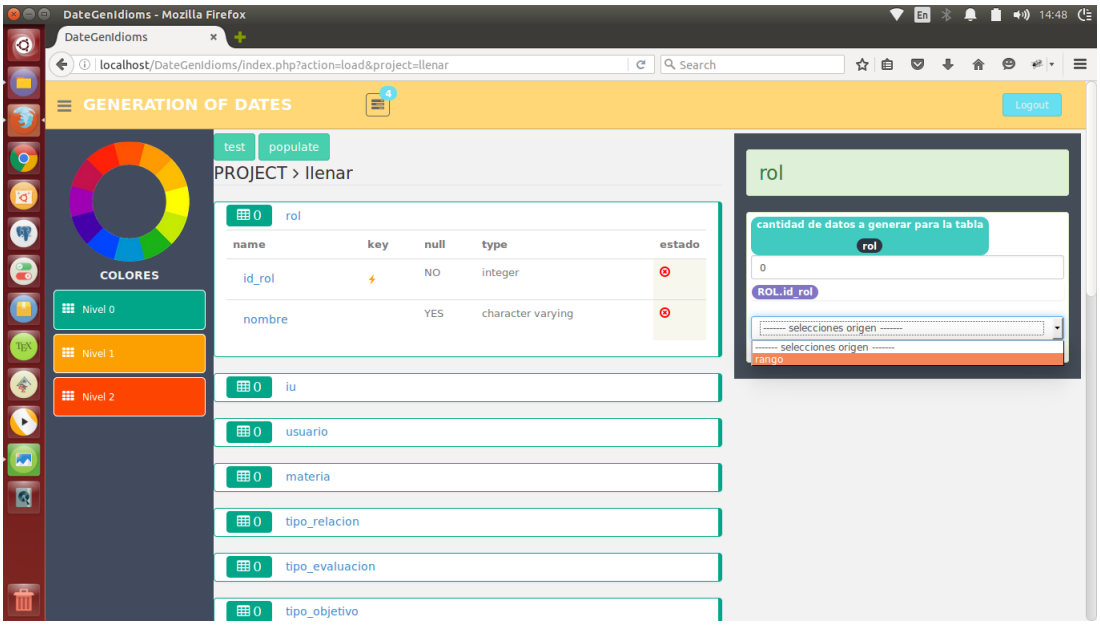
A continuacion damos clic en el nombre del proyeto lo cual no muestra toda la estructura del proyecto creado y ahi tenemos listo con el orden en que se debe llenar la base de datos iniciando primero todos los que son de color verde terminando con los rojos.

Figura 7.8: 7



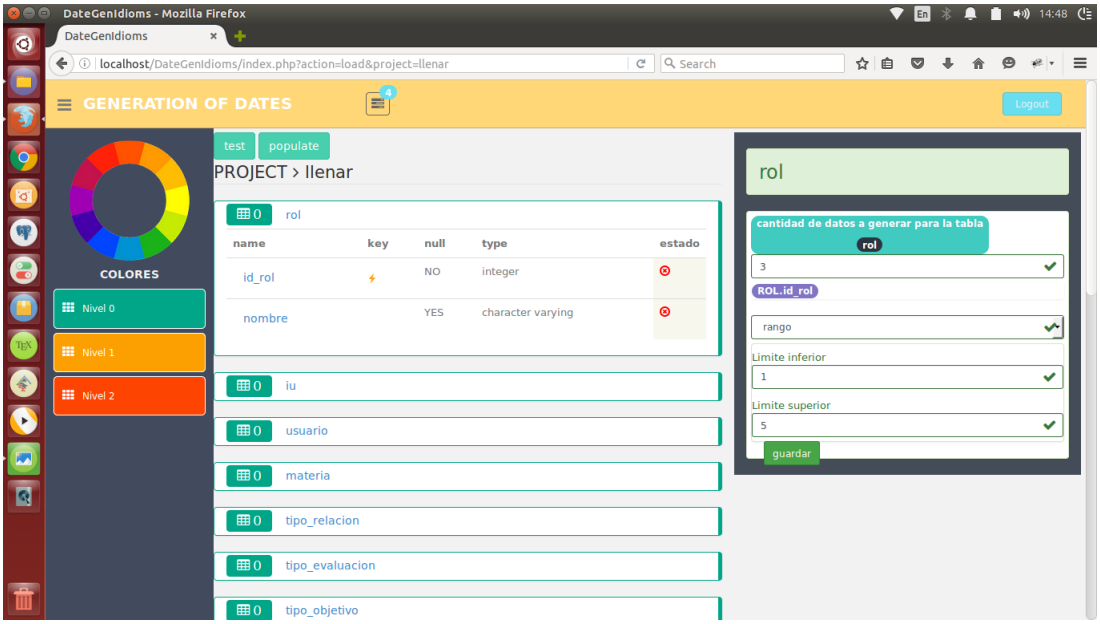
Si observamos en la Figura ?? se tiene `id_rol, nombre`, donde `id_rol` es la llave primaria por lo cual este tipo de datos se genera en un cierto rango de número.

Figura 7.9: 9



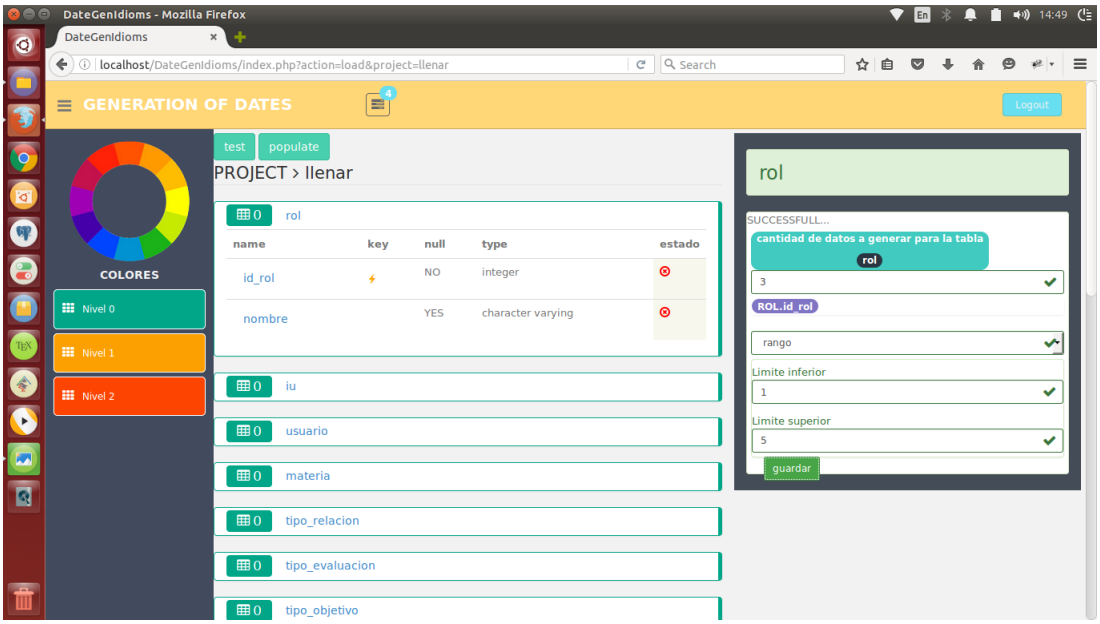
Según el tipo de dato el formulario tiene una variación y las opciones que ofrece como queremos generarlo, para el caso de `id_rol` al ser una llave primaria tenemos la opcion de generar en un cierto rango de numeros.

Figura 7.10: 10



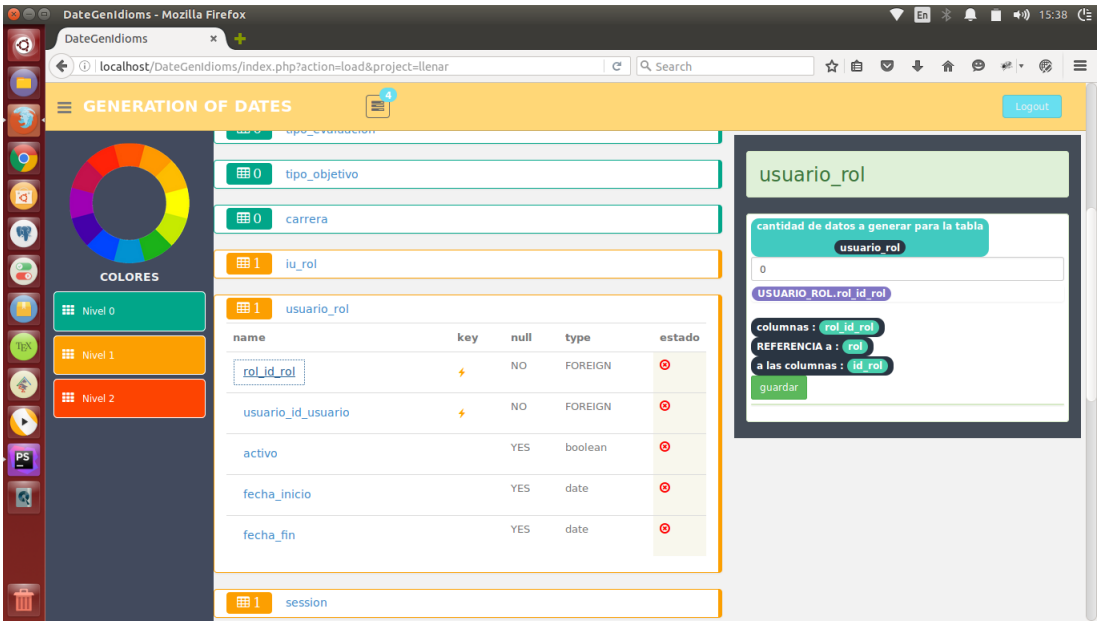
Una vez llenada el formulario damos clic en guradar.

Figura 7.11: 11



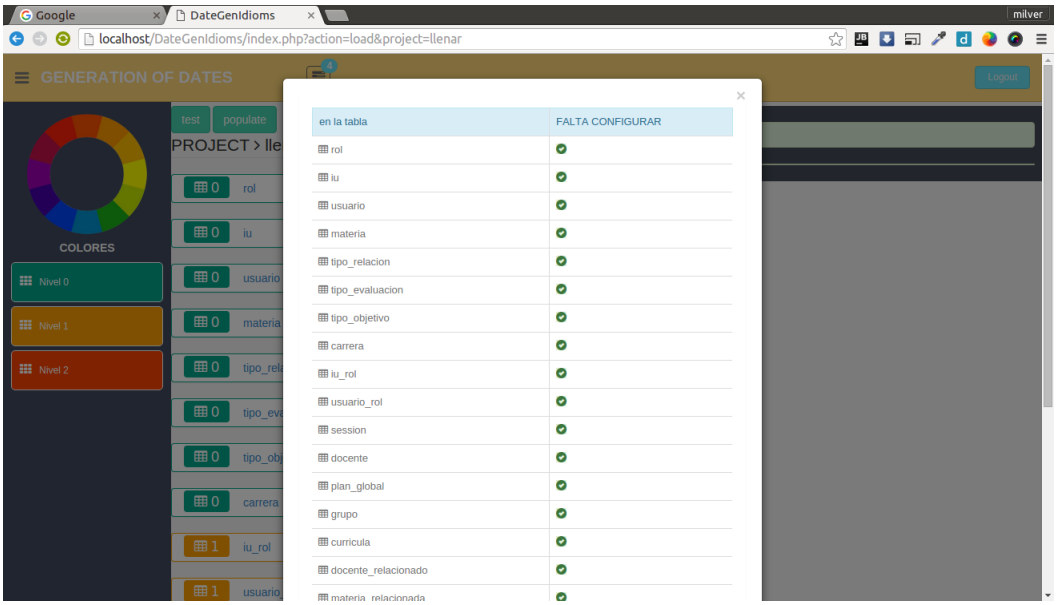
Ahora veamos a una tabla que haga referencia `usuario_rol` donde no es necesario llenar formularios.

Figura 7.12: 12



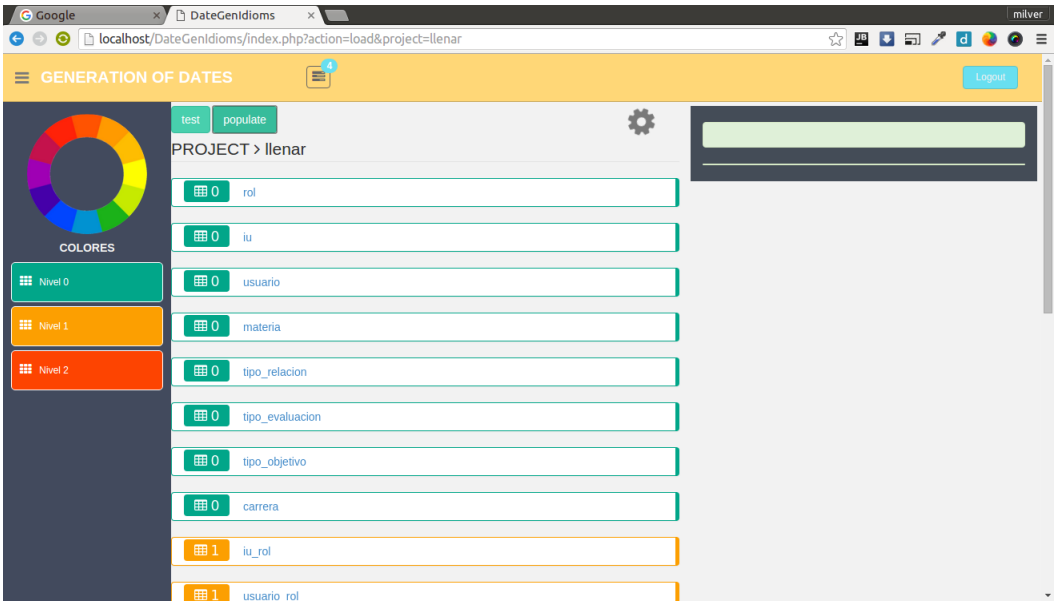
Una vez que se conpleta la configuración de todas las tablas damos clic en el boton test y como podemos observar en la Figura 7.13

Figura 7.13: estado de configuracion



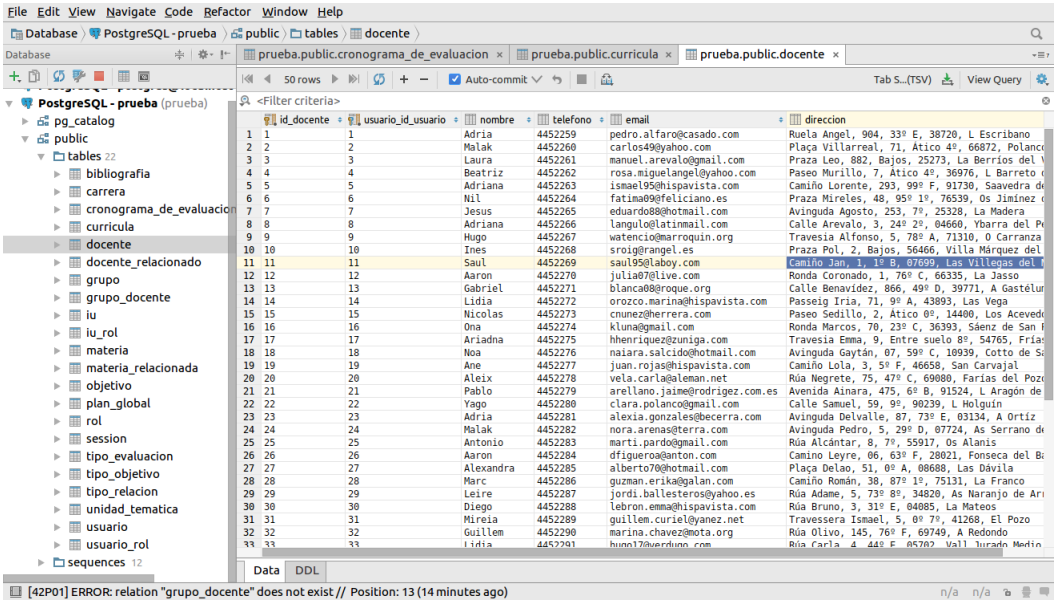
Si se tiene toda la configuración ya se puede llenar la base de datos dando clic en el boton populate, esperamos que haga el llenado para posteriormente ver como se ven reflajado en la base de datos.

Figura 7.14: llenando la base de datos



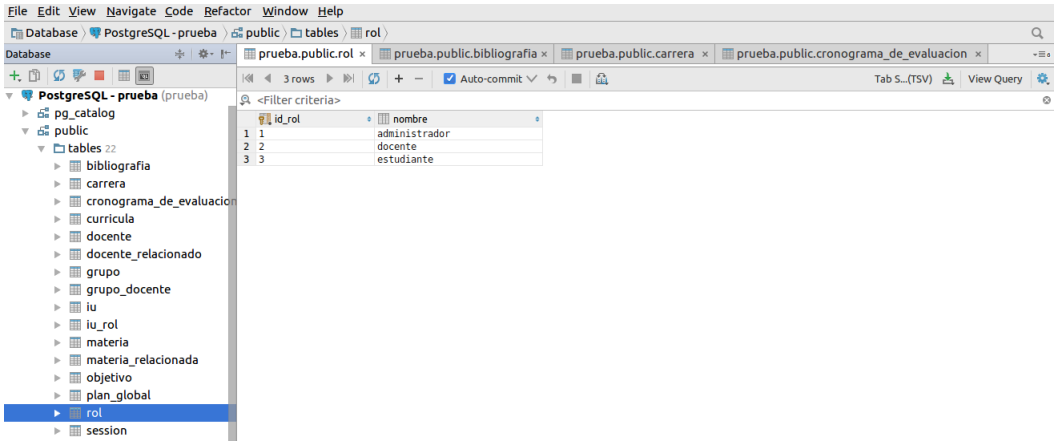
Si vemos en la base de datos la tabla `docente` ya se tiene datos de prueba como vemos en la Figura 7.15

Figura 7.15: tabla docente



pasa lo mismo con la tabla `rol` podemos observar los tres datos que ingresamos.

Figura 7.16: tabla rol



si se tiene la misma base de datos en distintas maquinas podemos usar el sql generado por el generador.

Figura 7.17: sql generado

```
INSERT INTO rol (id_rol, nombre) VALUES (1, 'administrador');
INSERT INTO rol (id_rol, nombre) VALUES (2, 'docente');
INSERT INTO rol (id_rol, nombre) VALUES (3, 'estudiante');

INSERT INTO iu (id_iu, activo) VALUES (1, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (2, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (3, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (4, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (5, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (6, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (7, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (8, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (9, TRUE);
INSERT INTO iu (id_iu, activo) VALUES (10, TRUE);

INSERT INTO usuario (id_usuario, password, login, activo) VALUES (1, 'Raul', 'Marco', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (2, 'Vega', 'Pol', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (3, 'Manuel', 'Ainara', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (4, 'Zoe', 'Dario', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (5, 'Jan', 'Cesar', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (6, 'Nahia', 'Ruben', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (7, 'Izan', 'Manuel', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (8, 'Marina', 'Nahia', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (9, 'Rafael', 'Julia', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (10, 'Ariadna', 'Gerard', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (11, 'Claudia', 'Miguel', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (12, 'Diego', 'Ignacio', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (13, 'Paola', 'Elena', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (14, 'Blanca', 'Bruno', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (15, 'Diana', 'Alexandra', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (16, 'Jimena', 'Jimena', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (17, 'Jose Manuel', 'Nicolas', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (18, 'Diana', 'Sofia', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (19, 'Erik', 'Arnau', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (20, 'Angela', 'Yago', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (21, 'Gabriela', 'Pablo', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (22, 'Ivan', 'Martina', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (23, 'Yaiza', 'Malak', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (24, 'Jimena', 'Jan', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (25, 'Daniel', 'Paula', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (26, 'Manuel', 'Nadia', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (27, 'Victor', 'Ismael', TRUE);
INSERT INTO usuario (id_usuario, password, login, activo) VALUES (28, 'Aaron', 'Biel', TRUE);
```

Capítulo 8

Conclusiones

En este trabajo se diseño e implemento un conjunto de algoritmos para generar datos de prueba para base de datos y algunas técnicas para obtener el orden en que se debe llenarlo, con el objetivo automatizar el proceso del llenado de una base de datos, entre las técnicas y algoritmos mas importantes podemos mencionar.

1. La implementacíon de algorítmos para obtener el orden correcto apartir de una lista de tablas pertenecientes a una base de datos, para lo cual se tomó encuenta diferentes casos que podria darse en el diseño de una base datos. Como resultado se tiene una lista de tablas ordenadas segun el orden correcto en que deben ser llenados, en el resultado se podria tener un lista de conjuntos de tablas donde las tablas que estén en el mismo conjunto tienen el mismo nivel de prioridad.

2. La implemetación de técnicas para el manejo referencial de las llaves primarias y foraneas tomando encuenta que una base de datos esté basada en el concepto E-R Idioms. Obteneniendo como resultado el correcto manejo de las llaves foraneas evitando asi la inconsistencia de datos.

3. La implementacíon de algoritmos para generar palabras, nombres, fechas, correos electronicos.

4. La implementacion de un prototipo

Bibliografía

[1] 1keydata. Sql tutorial - learn sql query language. 2001. URL http://www.1keydata.com/sql/sql.html. Online; accessed 30-Abril-2015.

[2] Henry F. Korth Abraham Silberschatz. *Fundamentos de base de datos*. Mcgraw-hill/interamericana de espa na, S.A.U, 2002.

[3] Lorenzo Alberton. Extracting meta information from postgresql. 2006. URL http://www.alberton.info/postgresql_meta_info.html#.VUKMcif_6ko. Online; accessed 30-Abril-2015.

[4] aulaClic S.L. Ddl, lenguaje de definición de datos. 2010. URL http://www.aulaclic.es/sqlserver/t_8_1.htm. Online; accessed 30-Abril-2015.

[5] Solvusoft Corporation. What is mydatagen. 2011. URL http://www.solvusoft.com/en/files/error-virus-removal/exe/windows/ems-database-management-solutions-inc/sql-manager-net-ems-database-management-solutions/mydatagen-exe/. Online; accessed 30-Abril-2015.

[6] Datanamic. Datanamic. 2014. URL http://www.datanamic.com/. Online; accessed 30-Abril-2015.

[7] Datanamic. Generating test data with default settings. 2014. URL http://www.datanamic.com/support/vd-ddg001.html. Online; accessed 02-Abril-2015.

[8] Generatedata. Generatedata. 2014. URL http://www.generatedata.com/#t3. Online; accessed 30-Abril-2015.

[9] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 1986.

[10] Addison Wesley Longman. *Introducción a los sistemas de bases de datos*. Design and Production Services, 7 edón., 2001.

[11] Marcelo Flores Soliz. Er idioms. 2006. URL https://marcelofloress.wordpress.com/er-idioms/. Online; accessed 30-Abril-2015.

[12] EMS Database Management Solutions. Ems datagenetor for postgresql. 1999. URL http://www.sqlmanager.net/en/products/postgresql/datagenerator. Online; accessed 30-Abril-2015.

[13] thara gireesh. Getting meta information of a postgresql database. 2010. URL http://www.alberton.info/postgresql_meta_info.html#.VUKMcif_6ko. Online; accessed 30-Abril-2015.