Universitat Oberta de Catalunya

uoc.edu

Diseño de la migración de una base de datos *legacy* IDMS a una base de datos relacional PostgresSQL

Cristian Daza Povedano
Grado de Ingeniería Informática

Jordi Ferrer Duran
Josep Corbasí Morales

23/06/2023

Licencias alternativas (elegir alguna de las siguientes y sustituir la de la página anterior)

A) Creative Commons:

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-SinObraDerivada 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento-CompartirIgual 3.0 España de Creative Commons

Esta obra está sujeta a una licencia de Reconocimiento 3.0 España de Creative Commons

B) GNU Free Documentation License (GNU FDL)

Copyright © AÑO TU-NOMBRE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free

C) Copyright

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Diseño de la migración de una base de datos legacy IDMS a una base de datos relacional Postgres*
Nombre del autor:	*Cristian Daza Povedano*
Nombre del consultor/a:	*Jordi Ferrer Duran*
Nombre del PRA:	*Josep Cobarsí Morales*
Fecha de entrega (mm/aaaa):	06/2023
Titulación::	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Bases de datos*
Idioma del trabajo:	*Español*
Palabras clave	*IDMS PostgreSQL migración*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El presente trabajo tiene como finalidad diseñar y aplicar una serie de procesos que permitan migrar la estructura de una base de datos jerárquica IDMS a una base de datos relacional más actual como puede ser PostgreSQL. El objetivo principal no es hacer el ejercicio de la migración de datos en sí, sino definir el proceso que pueda aplicarse a cualquier otra base de datos IDMS.

Esta migración permitiría a una empresa mejorar sus sistemas, así como facilitar la integración con otras aplicaciones y ERP actuales, a la vez que reduce los costes cada vez mayores impuestos por CA IDMS así como la escasez de profesionales.

Para poder llevarlo a cabo, en un primer paso se establecerán unas equivalencias entre estructuras y se definirán una serie de reglas iniciales para la transformación de los componentes. Posteriormente, de forma iterativa se procederá al a normalización e identificación de mejoras.

Como conclusión final se ha conseguido el objetivo, cubriendo tantos aspectos como ha sido posible (índices, funciones, *triggers*, u optimización de consultas entre otros) y obteniendo una base de datos PostgreSQL funcional.

Abstract (in English, 250 words or less):

The objective of this work is to design and apply some series of processes that allow migrating the structure of an IDMS hierarchical database to a more current relational database such as PostgreSQL. The main objective is not the data migration itself, but to define a process that can be applied to any other IDMS database.

This migration would allow a company to improve its systems, as well as ease integration with existing ERP and applications, while reducing the ever-increasing costs imposed by CA IDMS as well as the shortage of professionals.

In order acomplish it, in a first step some equivalences between structures will be established and a series of initial rules will be defined for the transformation of the components. Later, as an iterative process, the standardization and identification of improvements will be carried out.

As a final conclusion, the objective has been achieved, covering as many aspects as possible (indexes, functions, triggers, or query optimization, among others) and obtaining a functional PostgreSQL database.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

El sistema gestor de bases de datos IDMS, acrónimo de *Integrated Database Management System*, tiene su origen a inicio de los años 70 en el centro de datos de B.F. Goodrich Chemical Corporation (BFGCC) en Cleveland. El *B.F Institute* tomó la decisión de que todos sus centros de datos utilizasen equipamiento IBM, y siendo que hasta la fecha IBM únicamente disponía de IDS (*Integrated Data Store*) diseñado por Charlie Bachman para *General Electric*, se hizo patente la necesidad de desarrollar un nuevo sistema gestor de bases de datos que compartía elementos comunes con IDS, como la navegación entre ficheros o las relaciones de membresía entre entidades, que fue ampliado con conceptos CODASYL[1].

Posteriormente, en los años 80, tras ser incorporado por la *Cullinane Corporation* y las mejoras añadidas por ésta, como la integración de componentes en línea - programas *online* que se ejecutan bajo demanda, a diferencia de las tareas programadas conocidas como *batch*- , permitió que IDMS se convirtiera en uno de los sistemas líderes. Muchas empresas del sector industrial en expansión adoptaron IDMS como sistema gestor de bases de datos: un sistema más eficiente en el manejo de grandes volúmenes de información que sus competidores relacionales en el momento, y que fue usado por importantes bancos y entidades como ACME Cleveland Co., Mercedes Benz, Boeing Computer Services, o la ya mencionada General Electrics.

Es de acuerdo a Peter Karasz[2], que tras ser adquirido por Computer Associates en 1989 y la decisión de no potenciar sus posibilidades para consultas SQL como otros sistemas emergentes (a pesar de incorporarlas desde 1992), IDMS entró en un proceso en que dejaría poco a poco de ser competitivo, resultando en la falta de profesionales informáticos cualificados y un incremento constante en el intrincado sistema de licencias de los diferentes componentes[3] necesarios para su funcionamiento.

Este es el marco de nuestro caso de estudio: una mediana empresa de almacenes de piezas del sector automoción creada en los años 80 que actualmente ya no puede hacer frente a los costes de su sistema Mainframe sobre IDMS y decide iniciar el proceso de migración a PostgresSQL, con el objetivo de:

- Facilitar también una modernización de los sistemas con la inclusión de tecnologías orientadas al web sobre dicha base de datos (Java Spring y Angular son un ejemplo).

1. Conference on Data Systems Languages. Consorcio de industrias informáticas que resultó en la creación del lenguje COBOL.
2. Karasz, Peter (1998). "The Origins of IDMS" (editado por Chris Hoelscher). Online postings.
3 https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/release-notes/portfolio-simplification-for-ca-idms.html

- Minimizar no solo costes de mantenimiento, sino también de nuevos desarrollos.
- Facilitar la portabilidad, añadiendo posibilidades futuras de incorporación de otras marcas de piezas.
- Impulsar de nuevo sus almacenes como centros logísticos en un mundo globalizado cada vez más competitivo para recuperar mercado.
- Permitir una mejor integración con otros ERP (*Enterprise Resource Planning*) del mercado que usen tecnologías sobre PostgresSQL.

1.2 Objetivos del Trabajo

El objetivo del presente trabajo de fin de grado es realizar un estudio, análisis, y presentar una posible solución a una necesidad de una empresa de almacenamiento de componentes del sector automóvil que desea modernizar sus sistemas heredados, pero cuyo proceso sería extrapolable a otras bases de datos legacy en IDMS.

Para ello, se presentará la situación de partida y se realizará un estudio de conversión y reestructuración de componentes hacia una base de datos normalizada. Como producto final obtendremos un script SQL que puede ejecutarse para crear la nueva base de datos.

1.3 Enfoque y método seguido

Ante la posibilidad de modernizar su sistema gestor de base de datos, una empresa podría preguntarse si realizar una migración del sistema actual es una inversión realmente alineada con sus objetivos o si por el contario sería más rentable adoptar una de las soluciones ERP conocidas del mercado actual como ORACLE NETSUITE o SAP S/4 HANA y externalizar los costes mediante un contrato SaaS (modalidad en que la empresa contrata el uso y el soporte de una aplicación pero no compra la propiedad).

Es una decisión que se debe tomar a nivel de estrategia empresarial. El cambio conceptual de la base de datos a una nueva proporcionada por el ERP podría tener un elevado impacto transversal a gran parte de la organización e inesperados costes indirectos, haría falta formar de nuevo al personal y menguaría el conocimiento de los procesos adquirido por los miembros más veteranos de la compañía y que es clave para su operatoria. Es por eso que se decide para el presente trabajo de fin de grado que la empresa desea migrar su sistema, manteniendo la estructura básica, pero no siendo una copia uno a uno, pues arrastraría los problemas del pasado y los *hotfixes* (soluciones a errores y necesidades críticas que hubieron de implementarse a lo largo

de más de 40 años), por lo que se deberá realizar las mejoras necesarias para obtener un producto eficiente y flexible.

1.4 Planificación del Trabajo

1.4.1 Recursos

Los recursos necesarios para llevar a término el presente TFG son:

- Una herramienta digital de modelado UML. La principal opción que se tiene en cuenta es Microsoft VISIO por la existencia de plantillas para Diagramas de Bachmann en el internet, pero como alternativa la aplicación gratuita draw.io también será tenida en cuenta.
- Editor de texto de Microsoft Office, y el conjunto de aplicaciones disponibles en caso de ser necesarias, si bien Open Office puede considerarse como alternativa.
- Instalación localhost de PostgresSQL versión 13.4
- Como UI con la base de datos se usará pgAdmin versión 5.2

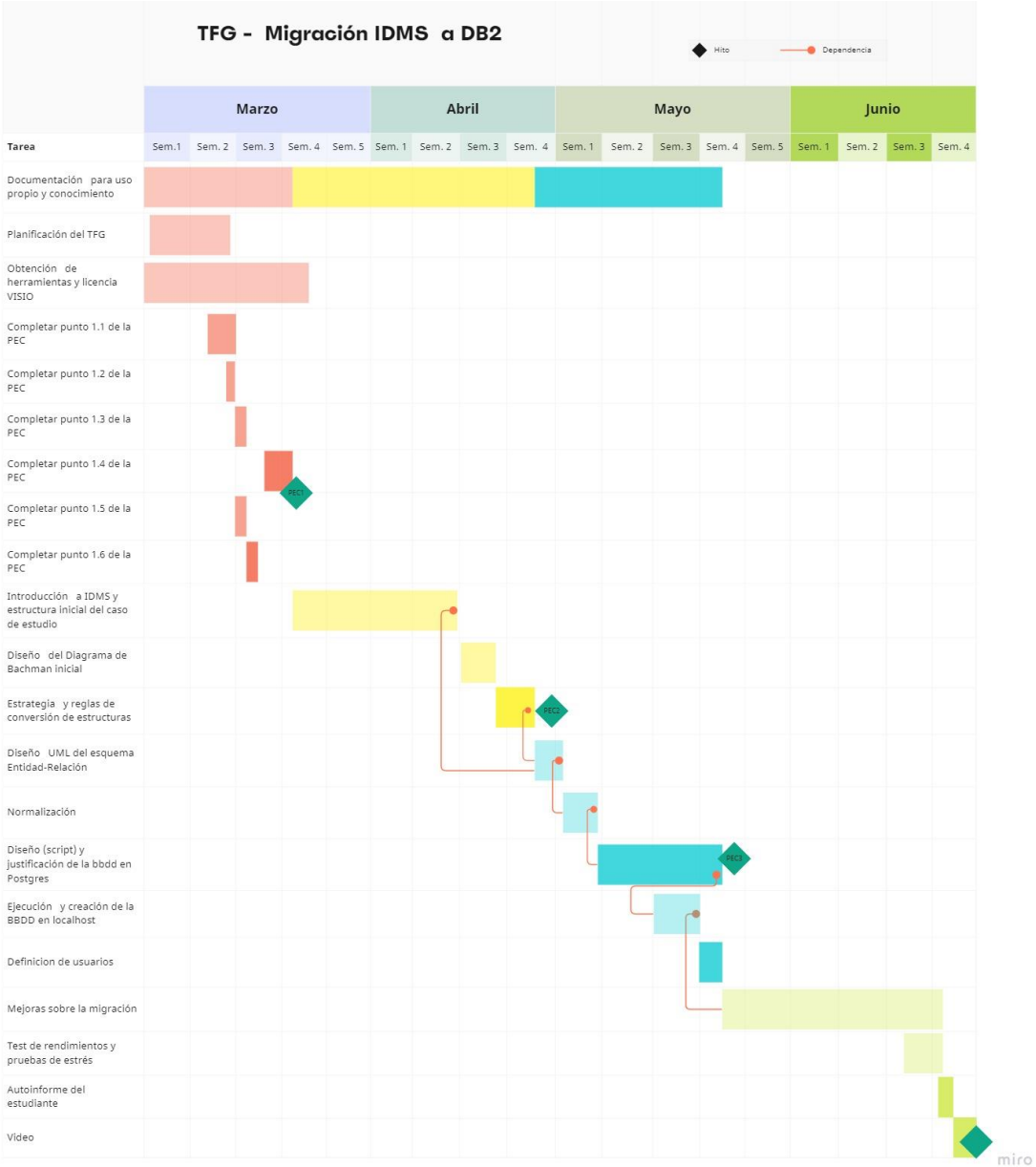
1.4.2 Planificación temporal e hitos

Los hitos principales del proyecto junto con el alcance esperado para cada uno en el momento de realizar la primera etapa (PEC1) y una estimación en horas a alto nivel correspondería con:

Hito	Tarea	Fecha inicio	Fecha fin	Horas estimadas
PEC1		02/03/2023	20/03/2023	15,20
	Planificación del TFG	02/03/2023	10/03/2023	3,00
	Obtención de herramientas y licencia VISIO	02/03/2023	20/03/2023	4,00
	Completar punto 1.1 de la PEC	10/03/2023	12/03/2023	2,00
	Completar punto 1.2 de la PEC	12/03/2023	12/03/2023	0,50
	Completar punto 1.3 de la PEC	13/03/2023	14/03/2023	0,50
	Completar punto 1.4 de la PEC	16/03/2023	20/03/2023	4,00
	Completar punto 1.5 de la PEC	13/03/2023	14/03/2023	0,20
	Completar punto 1.6 de la PEC	14/03/2023	15/03/2023	1,00
	Documentación para uso propio y conocimiento	02/023/2023	20/03/2023	8,00

PEC2		21/03/2023	24/04/2023	52,00
	Introducción a IDMS y estructura inicial del caso de estudio	21/03/2023	15/04/2023	20,00
	Diseño del Diagrama de Bachman inicial	15/04/2023	21/04/2023	20,00
	Estrategia y reglas de conversión de estructuras	21/04/2023	24/04/2023	4,00
	Documentación para uso propio y conocimiento	21/03/2023	24/04/2023	8,00
PEC 3		25/04/2023	25/05/2023	54,00
	Diseño (script) y justificación de la bbdd en Postgres	05/05/2023	25/05/2023	30,00
	Diseño UML del esquema Entidad-Relación	25/04/2023	01/05/2023	6,00
	Normalización	01/05/2023	05/05/2023	6,00
	Ejecución y creación de la BBDD en localhost	17/05/2023	20/05/2023	2,00
	Definición de usuarios	20/05/2023	25/05/2023	2,00
	Documentación para uso propio y conocimiento	25/04/2023	25/05/2023	8,00
Entrega Final		26/05/2023	23/06/2023	34,00
	Mejoras sobre la migración	26/05/2023	21/06/2023	20,00
	Test de rendimientos y pruebas de estrés	15/06/2023	21/06/2023	12,00
	Autoinforme del estudiante	21/06/2023	22/06/2026	2,00
	Video	22/06/2023	23/06/2023	2,00
Tribunal evaluación		26/06/2023	30/06/2023	4,00

Siendo su representación en forma de diagrama de Gantt la siguiente:



1. Diagrama de Gantt

1.4.2 Análisis de riesgos y contingencias

El objetivo principal de análisis de riesgos es identificar las causas de las amenazas potenciales que pueden surgir durante la realización del proyecto y poder realizar un plan de contingencia acorde que permita minimizar dichos riesgos.

Fase	Riesgo	Contingencia	
		Medidas preventivas	Acciones correctivas
PEC 2	Falta de tiempo general debido a elementos externos: aumento de la carga en el trabajo, larga enfermedad, etc…	Ceñirse a la planificación y adelantar tareas en la medida de lo posible.	1. Menor extensión en el apartado de introducción a IDMS (cambio de texto por fuentes de referencia) 2. Reducir el número de tablas en esta primera fase. 3. Reducir el número de atributos de las tablas a los mínimos indispensables.
PEC 2	Problemas en obtener la licencia de VISIO	Durante PEC1 en contacto con soporte a usuarios.	Uso de otras herramientas como draw.io
PEC 2	Curva de aprendizaje de uso detallado de VISIO excesiva	--	Uso de otras herramientas como draw.io y disminuir el alcance de las tablas durante la presente fase
PEC 3	Falta de tiempo general debido a elementos externos: aumento de la carga en el trabajo, larga enfermedad, etc…	Ceñirse a la planificación y adelantar tareas en la medida de lo posible.	1. Simplificar estructura de tablas (puede afectar a los resultados de la fase anterior) 2. Simplificar diseño UML
PEC 3	Problemas con la BBDD en localhost o PgAdmin	Se ha comprobado el actual funcionamiento de pgAdmin v5.2 con PostgreSQL v13.4	Actualizar a una versión más reciente de pgAdmin o PostgreSQL
Entrega Final	Falta de tiempo general debido a elementos externos: aumento de la carga en el trabajo, larga enfermedad, etc…	Ceñirse a la planificación y adelantar tareas en la medida de lo posible.	Disminuir el alcance de las mejoras.
Transversal	Problemas con la máquina de trabajo	Guardar copias en la nube con cierta periodicidad	Solventar el problema o instalar en una nueva máquina
Transversal	Perdida de conectividad	--	Utilizar alguno de los puntos de acceso a la red alternativos (datos móviles, familiares, bibliotecas, …)
Transversal	Falta de tiempo debido a otras asignaturas	Correcta planificación y coordinación con el TFG	En última instancia, priorizar el TFG
Transversal	Incapacidad para trabajar por enfermedad (esclerosis múltiple)	Adelantar el máximo trabajo que sea posible	Hablar con el profesorado y replanificar si es posible.

1.5 Breve sumario de productos obtenidos

Fichero texto con las sentencias SQL para la creación de la base de datos.
Imágenes de los esquema UML a lo largo de las distintas fases de la creación de la nueva base de datos Postgres.

1.6 Breve descripción de los otros capítulos de la memoria

En los siguientes capítulos se abordará la implementación de la migración así como las decisiones tomadas:

- Introducción a IDMS y estructura inicial. Para aquellos que no conozcan las bases de datos de ficheros IDMS este capítulo proveerá una visión inicial sobre cómo se definen sus estructuras, como se relacionan las entidades, cómo se navega entre entidades y cómo se representan mediante un diagrama de Bachman, y algunos de los comandos más comunes,.

- Estrategia y reglas de conversión de estructuras. En esta sección revisaremos las distintas alternativas que se pueden adoptar en cuanto al diseño de la nueva base de datos, y comentaremos qué equivalencias pueden observarse entre ambos tipos de bases de datos y qué reglas generales seguiremos. Será en el siguiente capítulo donde se mostrará la aplicación de las mismas y las decisiones tomadas durante el proceso de diseño de la implementación en PostgreSQL.

- Estructura y justificación de la bbdd en Postgres. Incluye el detalle de por qué se decide la creación de cada componente en base a las reglas definidas en el capítulo anterior y cómo se relacionan los mismos con respecto a la base de datos original, así como la correcta normalización al menos hasta la Forma de Boyce-Codd, estudiando si se aplicará 4º y/o 5º Forma Normal. El resultado final de este capítulo será el documento SQL que permita la creación de la base de datos.

- Mejoras sobre la migración. Una vez la estructura básica ha sido creada en Postgres, añadiremos estructuras propias que no vienen de la base de datos original en IDMS como es la definición de vistas, de *triggers*, de funciones, y de índices que puedan resultar en un mejor rendimiento del sistema.

2. Introducción a IDMS

Para entender de forma correcta el objetivo y utilidad del presente trabajo de fin de grado, así como las decisiones tomadas y su razonamiento, es necesario primero realizar una pequeña introducción los lectores a los conceptos IDMS y sus estructuras, ya que es un tipo de base de datos que actualmente no se enseña en los diferentes medios educativos y sus profesionales expertos son cada vez más escasos.

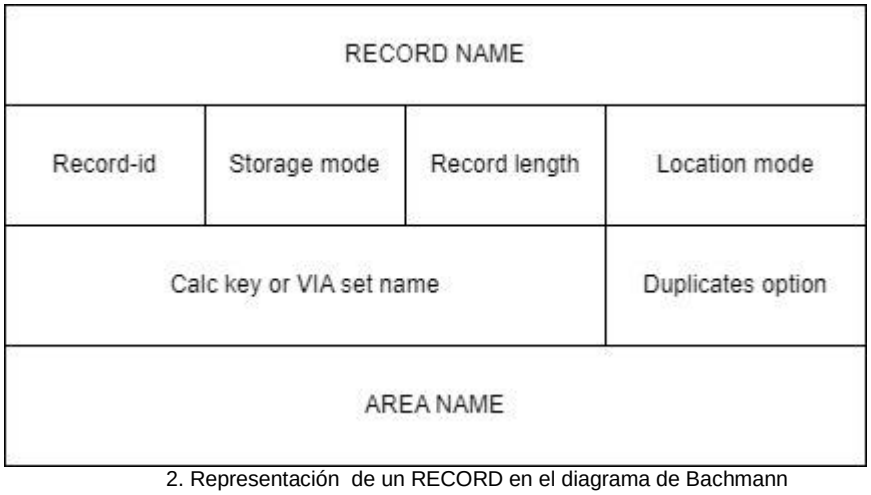
Una base de datos IDMS se divide físicamente en áreas, y cada área a su vez se divide en páginas, siendo una página la unidad más pequeña posible de transferencia entre la memoria principal y el disco duro. Es posible que diferentes áreas dentro de una misma base de datos tengan tamaños de página diferente. Cada página puede almacenar un máximo de 255 líneas (*rows*), por lo que el tamaño de página para un área concreta viene determinado por el tipo de los registros que se alocarán, ya que todas las ocurrencias de un mismo tipo de registro (tabla en Postgres) estarán siempre dentro de la misma área.

En IDMS, el equivalente sintáctico del concepto de tabla se indica como tipo de registro. Para simplificar a la vez que diferenciar de la nomenclatura Postgres, de ahora en adelante lo identificaremos mediante RECORD, el concepto de línea o registro, como RECORD *OCCURENCE*, y la relación entre dos entidades se denomina SET.

En estas bases de datos sólo puede haber un esquema, que incluye una descripción completa de todos los RECORD, RECORD OCCURENCE, SET, archivos y áreas. Estos son los conocidos como *Network-defined*[1]. Esto, como vemos, es un concepto de esquema diferente a PostgreSQL, que permite diferentes esquemas en cada base de datos y que además puede permitirse el acceso entre ellos o a elementos comunes propios de la base de datos. En un esquema puede haber definidos diferentes subesquemas, con la particularidad que un programa cobol sobre IDMS no puede acceder a más de un subesquema simultáneamente. También es interesante poner de relevancia que en cobol sobre IDMS se puede limitar el acceso programáticamente a los subesquemas y áreas (por ejemplo, que unas sean de solo lectura y otras permitan escritura) al ser estos en realidad ficheros.
A título informativo, dado que IDMS permite instrucciones SQL básicas desde inicios de los 90, también existen los llamados esquemas *SQL-defined* y *SQL schema for network database*, que están fuera del alcance de este proyecto.
Una forma común de representar los *records* y su algunas de sus características es mediante los *Data Structure Diagram* (DSD) y los diagramas de Bachmann.

1. https://knowledge.broadcom.com/external/article/210835/idms-the-three-types-of-schema-explained.html

Así, siguiendo la nomenclatura definida por Charlie Bachmann, la estructura de un RECORD la representaremos en nuestro diagrama inicial como como:



2. Representación de un RECORD en el diagrama de Bachmann

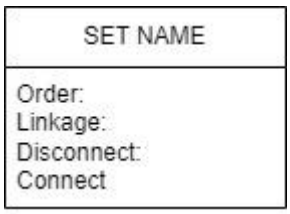
- Record name: debe ser único por cada esquema y puede tener entre 1 y 16 caracteres. La primera letra debe ser de la A-Z, #, $ o @, el resto puede ser alfanumérico o guion.
- Record-id: es un campo numérico sin signo que se utiliza como identificador interno y cuyo valor está comprendido entre 10 y 9999, y que debe ser único por cada área.
- Storage mode: los posible valores son F si las RECORD *OCCURENCE* son de longitud fija, V si son de longitud variable, o C si son comprimidas. Hay que tener en cuenta que si define el tipo como V, la longitud máxima no podrá sobrepasar la definida en el atributo *Record length*.
- Record length: tamaño máximo en bytes de un RECORD *OCCURENCE*.
- Location mode: identifica la forma en que se almacenarán físicamente las ocurrencias del RECORD. Los posibles son:
    - CALC: un conjunto de atributos se declararán como calc-key y actuarán como clave lógica, permitiendo ser accedidos directamente.
    - VIA: los RECORD OCCURENCE se guardarán físicamente cercanos a los RECORD de tipo CALC con los que están relacionados. Sólo pueden ser accedidos directamente mediante la DB-key física, no por clave lógica.
    - DIRECT: para estas RECORD OCCURENCE el programa que guarda el dato debe indicar explícitamente en qué página debe guardarse el dato.
- CALC key / VIA SET name: en los RECORD de tipo CALC se indicará el atributo que hace de clave (puede ser compuesto), mientras que en las VIA se indica el SET por el que son accesibles.

- Duplicates option: Solo se especifica para los registros de tipo CALC, e indica (a diferencia de las claves primarias que veremos en Postgres) si es posible guardar más de un RECORD OCCURENCE con la misma CALC-key. Los posibles valores que son:
    - DN: no se permiten duplicados.
    - DF: duplicados primeros. El último elemento guardado será el primero recuperado al acceder a los datos mediante la CALC-key (posteriormente comentaremos algunas de las instrucciones comunes en IDMS para la navegación entre datos).
    - DL: duplicado al final. El último elemento guardado será el último retornado al acceder a los datos mediante la CALC-key.
- Area name: nombre del área donde se guardarán todas las ocurrencias del RECORD.

En una base de datos IDMS, cada RECORD OCCURENCE guardada en la base de datos tiene asociada una clave física (*db-key*) de 32 bits que la identifica de forma única, y que es independiente de la clave lógica, o CALC-key, de existir. A modo de ejemplo, en lenguajes como Cobol sobre IDMS es frecuente guardar los *db-key* en una variable de proceso para más tarde durante la ejecución volver a reposicionarnos en el registro concreto mediante una instrucción 'FIND record-name DB-KEY IS saved-key'. Esto establece lo que se denomina como *currency*, es decir, a qué registro está marcando actualmente el puntero del proceso: un concepto importante a la hora de entender cómo se navega a través de las RECORD OCCURENCE y de un RECORD a otro según sean CALC o VIA.

Para una definición completa de la sentencia y atributos de creación de un RECORD, referirse a la documentación de Broadcom[1].

Otra de las estructuras características son los SET, que definen la relación entre 2 RECORD. Aunque la relación habitual es entre un *record* de tipo CALC (llamado *owner* en adelante) y uno de tipo VIA (llamado *member* en adelante), también es posible la relación entre 2 CALC y 2 VIA en los que uno adquirirá siempre el rol de *owner*.
La representaremos en nuestro diagrama inicial con la siguiente información en las líneas de unión:



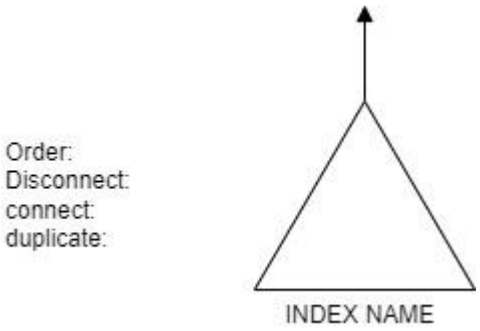
3. Representación de un SET en el diagrama de Bachmman

1. https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/idms-database/schema-statements/record-statement.html (consultado online 07.04.2023)

- Set name: debe tener entre 1 y 16 caracteres y ser único entre todos los componentes del esquema.
- Order: especifica el orden lógico en que se guardarán las RECORD OCCURENCE accedidas a través del el SET. Los posibles valores son:
    - FIRST: se posiciona el registro nuevo inmediatamente después del registro propietario, implementando por tanto un acceso LIFO.
    - LAST: se posiciona el registro nuevo inmediatamente antes del registro propietario, implementando por tanto un acceso FIFO.
    - NEXT: se posiciona el registro nuevo inmediatamente después del actual del *SRRT*.
    - PRIOR: se posiciona el registro nuevo inmediatamente antes del actual del SET.
    - SORTED: se posiciona el registro nuevo dependiendo del valor de uno más de sus atributos.
- Linkage: indica los tipos de puntero usados para el acceso. Posibles valores son:
    - N: cada elemento en el SET contiene un puntero al siguiente elemento del SET.
    - P: cada elemento en el SET contiene un puntero al anterior elemento del SET.
    - NP: combinación de los dos anteriores.
    - NPO: extensión de NP en que también es posible acceder al registro *owner* que referencia al puntero actual.
- Disconnect: especifica en qué forma una RECORD OCCURENCE puede desconectarse de un SET una vez que ya ha sido establecida su membresía.
    - M (*mandatory*): un registro solo puede ser desconectado de un SET si es borrado.
    - O (*optional*): una RECORD OCCURENCE puede desconectarse del SET y conectarse a otro, o a ninguno.
- Connect: especifica en qué forma una RECORD OCCURENCE se conecta al SET cuando se crea. Posibles valores son:
    - A (*automatic*): cuando se inserta un registro, IDMS lo conectará automáticamente a todos los *owner* para los que la *currency* haya sido establecida.
    - M (manual): un nuevo registro debe ser conectado programáticamente

Los SET son una diferencia importante con Postgres, que establece la relación entre diferentes tablas mediante la definición de una clave foránea directa (FOREIGN KEY en lenguaje SQL), no permitiendo un INSERT si no existe el dato referenciado.

Adicionalmente, IDMS permite definir un tipo especial de *SET* indexado que se define con la opción SORTED. Este concepto de índice es diferente del concepto de índice en Postgres, ya que en realidad guarda las db-key de los RECORD miembro en el orden particular en que serán recuperados según los atributos que formen parte del SET (que no tienen por qué ser miembros de la calc-key).
Lo representaremos en nuestro caso de estudio de la siguiente forma:



4 Representación de un ORDERED SET en el diagrama de Bachmann

- Index name: como en los SET, debe tener entre 1 y 16 caracteres y ser único entre todos los componentes del esquema.
- Order: indica qué atributos del RECORD al que pertenece el índice forman parte de la clave de ordenación y si es ascendente o descendente.
- Disconnect: como en los SET, indica en que forma una RECORD OCCURENCE se comportará para dejar de ser parte del índice.
- Connect: como en los sets, indica en que forma una RECORD OCCURENCE se comportará con respecto al índice durante la creación.
- Duplicate: indica como debe comportarse el acceso a datos en caso de haber duplicados referenciados. Posibles valores son:
    - DN (*duplicates not allowed*): No permite conectar valores duplicados al SET.
    - DF (*duplicates first*): En caso de que la clave de ordenación tenga duplicados, estos se guardarán en el SET antes que los existentes, reproduciendo así un acceso LIFO
    - DL (*duplicates last*): En caso de que la clave de ordenación tenga duplicados, estos se guardarán en el SET posteriormnente que los existentes, reproduciendo así un acceso FIFO

Finalmente, aunque no son un componente propiamente dicho de la base de datos, debemos mencionar los COPY ELEMENT[1]. Una *copy* en IDMS (o en Cobol) define la descripción de los datos de un *record*. Al ser

1. https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/idms-database/schema-statements/copy-elements-substatement.html

IDMS una base de datos de ficheros de longitud máxima definida, los atributos también son de longitud fija definida, en contraposición por ejemplo con el tipo de datos VARCHAR de PostgreSQL. Esta estructura se almacena en el diccionario del *schema*. Es una estructura jerarquizada, donde el nivel con menor número engloba a todo lo que haya con mayor número justo a continuación, hasta encontrar otro elemento del mismo nivel, siendo posible definir niveles dentro de niveles. Los tipos de dato más comunes son alfanuméricos (representado por 'X'), numérico (representado por '9'), o numérico comprimido (representado por COMP), no existiendo el concepto de NULL ya que todo campo ocupa el espacio designado (aunque puede estar no inicializado).

Esto también nos impactará en la migración ya que en postgres los campos compuestos son contrarios a la normalización.
Para facilitar el entendimiento, un ejemplo para guardar un usuario y su edad podría ser:

```
01 nivel-principal.
       02 subnivel-1-usuario-id.
              05 subnivel-2-nombre          PIC X(10).
              05 subnivel-2-apellido        PIC X(10).
       02 subnivel-1-usuario-edad         PIC 9(3) COMP.
```

A título ilustrativo, un ejemplo de uso en IDMS sería :

1. Obtener datos de un pedido xxxxxxx y posicionar la *currency* en él.
   ```
   OBTAIN  PEDIDO WHERE CALC = 'xxxxxxx'
   ```

2. Obtener la primera la líneas de ese pedido según el SET.
   ```
   OBTAIN FIRST LINEA WITHIN PEDIDO-LINEA
   ```

3. Bucle para recorrer todas las líneas obteniéndolas una a una para su uso hasta encontrar la deseada.
   ```
   OBTAIN NEXT LINEA WITHIN PEDIDO-LINEA
   ```

3. Estructura inicial del caso de estudio.

La estructura inicial representa la base de datos actual del almacén de componentes, que ha ido cambiando a través de los años según las necesidades y como es habitual en el mundo empresarial, ajustado a presupuesto. Esto se traduce a menudo en que la estructura no tiene por qué seguir los estándares académicos, estar normalizada, evitar repetición de atributos, etc … Forma parte del presente trabajo de fin de grado también el que, aprovechando que la empresa cambia de sistema de bases de datos, se debe revisar el estado actual y depurarlo.

A continuación describiremos las entidades principales del sistema original, sus relaciones, y sus principales atributos, de acuerdo a la información que se nos ha proporcionaría como parte de un proyecto para planificar la migración.

Algunas palabras clave que deberemos conocer son:

- FILLER: se trata de un nivel o elemento que no tiene un nombre definido porque no es necesario o relevante, pero cuyo espacio está reservado en el fichero.
- OCCURS: define que una estructura está repetida N veces, y cada una puede ser referenciada mediante un indicador (por ejemplo, puede accederse a la ocurrencia número 5).

CLIENTE (RECORD)
Son las tiendas, talleres, mayoristas o fabricantes que solicitan piezas al almacén. Es de tipo CALC.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		cliente-clave	x				9	CIF.
02		nombre					30	
02		direccion-envio						
	03	calle	x				30	
	03	numero		x			3	
	03	complemento	x				50	escalera, piso, etc
	03	cpostal		x			5	código postal
02		direccion-factura						
	03	calle	x				30	
	03	numero		x			3	
	03	complemento	x				50	escalera, piso, etc
	03	cpostal		x			5	código postal
02		fecha-alta		x			8	formato yyyymmdd
02		fecha-baja		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo 2= bloqueado ordenes 3= bloqueado factura 4= bloqueo total

CLI_ESTA (SET)
Relaciona CLIENTE como *owner* con CLI_ESTADISTICAS como *member*. Conecta ambos RECORD de forma automática durante la creación de CLI_ESTADISTICAS y una vez establecida la relación es permanente hasta el borrado de las estadísticas.

CLI_ESTADISTICAS (RECORD)
Información adicional de un cliente, cuantos pedidos hace al mes de cada tipo, fecha del último pedido, cuantía total pedidos. Estos datos son utilizados para estadísticas y elaboración de informes. Es un RECORD de tipo VIA.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		estadistica-periodo						
	03	año		x			2	formato yy
	03	mes		x			2	formato mm
02		pedidos-realizado		x			3	
02		pedidos-acumulado		x			5	
02		cuantia		x	2	x	8	6 entero + 2 decimales
02		cuantia-acumulada		x	2	x	12	10 entero + 2 decimales
02		fecha-ultimo-pedido		x			8	formato yyyymmdd

CLI_PEDIDO (SET)
Este *set* relaciona CLIENTE como *owner* con CLIENTE_PEDIDO como *member*, ya que un cliente puede tener muchos pedidos pero un pedido pertenece solo a un cliente. Al ser ambas de tipo CALC, la desconexión entre ambos RECORD se ha definido como opcional. Así mismo, la conexión se hará de forma programática.

CLIENTE_PEDIDO (RECORD)
Representa la cabecera de un pedido de cliente, por lo que tiene campos como total de coste de pedido, fecha de pedido, estado, o fecha de estado. Es de tipo CALC.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		pedido-clave						
	03	id-pedido		x			9	único entre todos los clientes
02		fecha		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo 2= completado 3= completado con backorder
02		fecha-estado						
	03	fecha-estado1		x			8	formato yyyymmdd
	03	fecha-estado2		x			6	formato hhmmss
02		prioridad		x			1	1= prioritario
02		Total-neto		x	2	x	8	6 entero + 2 decimales
02		Total-bruto		x	2	x	8	6 entero + 2 decimales

CLI_PED_LINEA (SET)
Relaciona la cabecera de un pedido representada por CLIENTE_PEDIDO como *owner* con las distintas líneas (componentes) de CLI_PED_LINEA que lo componen. Es por eso que se define la navegación posible como NPO para que los procesos puedan recorrer las líneas libremente. Así mismo la conexión es automática pues habremos establecido la *currency* de CLIENTE_PEDIDO al guardar las líneas, y la desconexión obligatoria ya pata borrar la cabecera de un pedido primero habría que borrar todas las líneas.

CLIENTE_PED_LI (RECORD)
Representa el conjunto de piezas/componentes solicitados para un pedido o backorder. Los campos que tiene son número de línea, identificador de pieza, cantidad, prioridad, total neto y total bruto. Es de tipo VIA.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		num-linea		x			3	no se admiten más de 999 líneas por pedido
02		id-pieza	x				30	
02		cantidad		x			3	de 1 a 999
02		coste-neto-unidad		x	2	x	7	5 entero + 2 decimales
02		coste-bruto-unidad			2	x	7	5 entero + 2 decimales
02		total-neto		x	2	x	8	6 entero + 2 decimales
02		total-bruto		x	2	x	8	6 entero + 2 decimales

CLI_LIN_PIEZA (SET)
Relaciona cada parte de PIEZA_LOCAL (*owner*) con las línea de los pedidos de cliente donde aparece (*member*). La desconexión es opcional para evitar que una pieza que ha quedado obsoleta y fuera de producción no bloquee los pedidos que están en estado completado.

CLI_BACKORDER (RECORD)
Cuando un cliente realiza un pedido, es posible que a la hora de servirlo no haya stock para alguna de sus líneas. Cuando esto sucede se crea un pedido de pendientes, o *backorder*. Estos pedidos contienen las líneas que no han podido ser servidas hasta que algún proceso de la aplicación los libera, creando un pedido agrupado de vendedor, es decir, un pedido al proveedor de dichas piezas. Cuando dicho pedido de vendedor (entidad VENDEDOR_PEDIDO) pasa a estado cerrado, otro proceso de la aplicación realiza el proceso en sentido contrario y recorre todas las backorder relacionadas y crea automáticamente nuevos pedidos para los clientes con los elementos que faltaban respecto al pedido original. Por lo tanto, una backorder se relaciona como máximo con 2 pedidos de cliente: el original y el final tras recibir el stock. Es de tipo CALC

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		backorder-clave		x			9	
02		id-pedido-orig		x			9	
02		fecha		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo 2= completado
02		fecha-estado		x			8	formato yyyymmdd

CLI_PED_BACK (SET)
Relaciona un pedido de backorder (*member*) con uno único pedido de cliente. Dado que los backorder se crean programáticamente durante la vida de un pedido, la conexión es manual, y no se podrá eliminar el pedido mientras exista una backorders.

CLI_BACK_LINEA (SET)
Relaciona un pedio de backorder (*owner*) con las líneas (*member*) que no se han podido servir en el pedido regular de cliente. La conexión, es decir, la adición de líneas, se realiza de forma manual durante el procesado del pedido.

CLIENTE_FACTURA (RECORD)
Cuando un pedido está listo, y todas sus líneas están en estado correcto, se añade a una factura. Si el pedido es prioritario, generará una factura automáticamente que solo lo contendrá a él. Si los pedidos no son prioritarios se acumulan en una factura hasta que un proceso semanal inicia el cobro. En caso de que haya que generar cargos especiales (líneas) o haya un problema, la factura queda en estado bloqueado. Es de tipo CALC.
Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		factura-clave						
	03	id-factura		x			9	
02		id-cliente	x				9	
02		fecha-factura		x			8	yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo 2= completado 3= bloqueada
02		fecha-estado		x			9	yyyymmdd
02		total-neto		x	2	x	8	6 entero + 2 decimales
02		total-bruto		x	2	x	8	6 entero + 2 decimales

CLI_FAC_PED (SET)
Relaciona una factura (*owner*) con un número de pedidos de un cliente (*member*). La conexión es automática durante el proceso de generación de la factura y se debe eliminar todas las líneas de la factura antes de desconectar la factura.

C_FACTURA_LINEA (RECORD)

Representa todas las líneas y cargos de una factura, que incluye los pedidos y sus costes extras asociados (descuentos, cargos, transporte, etc…) , así como cualquier otro concepto que no dependa de un pedido (gastos de gestión si los hubiere, sobrecargos, descuentos a nivel factura, …). Es de tipo VIA.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		num-linea		x			3	número de línea, máx. 999
02		id-pedido-orig					9	existe en CLIENTE_PEDIDO
02		id-concepto	x				30	excluyente con id-pedido-orig, es texto libre actualmente
02		fecha		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelada 1= activa 2= completada
02		fecha-estado		x			8	formato yyyymmdd
02		total neto		x	2	x	8	6 entero + 2 decimales
02		total bruto		x	2	x	8	6 entero + 2 decimales

VENDEDOR_PEDIDO (RECORD)

Representa la cabecera de un pedido a un proveedor, por lo que tiene campos como total de coste de pedido, fecha de pedido, estado, o fecha de estado. Estos pedidos tienen 2 tipos de origen:

- Introducidos desde el almacén para tener existencias
- Como producto de una backorder para completar un pedido de cliente

Es de tipo CALC.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		pedidoclave						
	03	id-pedido		x			9	único entre todos los clientes
02		fecha		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo 2= completado
02		fecha-estado						
	03	fecha-estado1		x			8	formato yyyymmdd
	03	fecha-estado2		x			6	formato hhmmss
02		total neto		x	2	x	8	6 entero + 2 decimales
02		total bruto		x	2	x	8	6 entero + 2 decimales

VEND_PED_LINEA (SET)
Relaciona un pedido a vendedor (*owner*) con las distintas líneas (*member*). La conexión es automática pues se crea durante la creación de VENDEDOR_PEDIDO y la currency está establecida, y la desconexión es manual pues para borrar el pedido primero hay que eliminar las líneas.

VENDEDOR_PED_LI (RECORD)
Representa el conjunto de piezas/componentes solicitados para un pedido a proveedor. Los campos que tiene son número de línea, identificador de pieza, cantidad, total neto y total bruto. Es de tipo VIA.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		num-linea		x			3	no se admiten más de 999 líneas por pedido
02		id-pieza	x				30	referenciado al maestro
02		cantidad		x			3	de 1 a 999
02		total-neto		x	2	x	8	6 entero + 2 decimales
02		total-bruto		x	2	x	8	6 entero + 2 decimales

VEND_LIN_PIEZA (SET)
Relaciona cada parte de PIEZA_MAESTRO (owner) con las línea de los pedidos de proveedor donde aparece (member). La desconexión es opcional para evitar que una pieza que ha quedado obsoleta y fuera de producción no bloquee los pedidos que están en estado completado.

VENDEDOR_FACTURA (RECORD)
Cuando un pedido está listo, y todas sus líneas están en estado correcto, se añade a una pre-factura o proforma. Los pedidos se acumulan en dicha factura hasta que un proceso semanal inicia la petición a cada proveedor mediante una interfaz de una aplicación, pudiendo este confirmarla (completándola) o corrigiendo cualquier error detectado por lo que queda en estado pendiente de aceptación por parte del almacén. Es de tipo CALC.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		factura-clave						
	03	id-factura		x			9	
02		Id-vendedor	x				9	
02		fecha-factura		x			8	yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo 2= completado 3= enviado proveedor 4=pendiente aceptar recalculo
02		fecha-estado		x			9	yyyymmdd
02		total-neto		x	2	x	8	6 entero + 2 decimales
02		total-bruto		x	2	x	8	6 entero + 2 decimales

02		total-neto-conf				x	8	valor devuelto por el proveedor
02		total-bruto-conf				x	8	valor devuelto por el proveedor

VEND_FAC_PED(SET)
Relaciona vendedor (owner) con sus facturas (member).

VEND_FAC_LINEA (SET)
Relaciona una factura (owner) con las líneas que la componen (member).

V_FACTURA_LINEA (RECORD)
Representa todas las líneas de una factura para los pedidos a proveedor. Los totales pueden ser de tipo proforma (propuesto), confirmado, o final (proporcionado por el vendedor). Es de tipo VIA

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		VEND_FAC_LINEA		x			3	número de línea, máx. 999
02		id-pedido-orig					9	existe en CLIENTE_PEDIDO
02		id-concepto	x				30	excluyente con id-pedido-orig, es texto libre actualmente
02		fecha		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelada 1= activa 2= completada
02		fecha-estado		x			8	formato yyyymmdd
02		Total-neto		x	2	x	8	6 entero + 2 decimales
02		Total-bruto		x	2	x	8	6 entero + 2 decimales
02		Total-neto-conf		x	2	x	8	6 entero + 2 decimales
02		Total-bruto-conf		x	2	x	8	6 entero + 2 decimales

VEND_PEDIDO (SET)
Relaciona un vendedor (owner) con todos sus pedidos (member). La conexión es mandatory pues se crea el pedido cuando la currency del vendedor se ha establecido.

VENDEDOR (RECORD)
Son los distintos proveedores a los que se pueden solicitar piezas, o bien las fábricas directamente. Es un *RECORD* de tipo CALC.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		VENDEDOR-CLAVE	x				9	CIF.
02		nombre					30	
02		direccion-factura						
	03	calle	x				30	
	03	numero		x			3	
	03	complemento	x				50	escalera, piso, etc.
	03	cpostal		x			5	código postal

02		fecha-alta		x			8	formato yyyymmdd
02		fecha-baja		x			8	formato yyyymmdd
02		estado	x				1	valores: 0= cancelado 1= activo

VEND_PIEZA (SET)
Relaciona un vendedor (owner) con todas las piezas que puede proveer (member).

PIEZA MAESTRO (RECORD)
El registro del maestro de piezas se carga a partir de la información recibida de los diferentes proveedores de forma diaria. Contiene por tanto información general de la parte, e información específica para el almacén (como el precio), pero no información útil para los clientes. Es de tipo CALC

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		PIEZA-CLAVE						
	03	id-pieza	x				20	cada vendedor usa su nomenclatura por lo que es posible que se repita
	03	id-vendedor		x			6	para evitar duplicados por id-pieza
02		descripcion	x				50	
02		precio-neto-vend		x	2	x	8	6 entero + 2 decimales
02		precio-bruto-vend		x	2	x	8	6 entero + 2 decimales
02		estado						valores: 0= bloqueado 1= activo

PIEZA_MAEST_LOC (SET)
Relaciona una pieza del fichero maestro con datos de proveedor (owner) con las diferentes personalizaciones locales (member). La desconexión es opcional ya que es posible que una pieza deje de ser válida en el maestro para pedirlo a importador, pero nuestro almacén puede tener existencias en stock.

PIEZA LOCAL (RECORD)
El registro de piezas locales representa las personalizaciones y precios de venta por parte del almacén a los clientes. Tiene la particularidad de que la clave puede estar repetida ya que puede haber diferentes valores en diferentes fechas. Es de tipo CALC.
Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		PIEZA-CLAVE						
	03	id-pieza	x				20	cada vendedor usa su nomenclatura por lo que es posible que se repita
	03	id-vendedor		x			6	para evitar duplicados por id-pieza

02		fecha-vigencia-ini		x			8	periodo de validez de esta personalización, formato yyyymmdd
02		fecha-vigencia-fin		x			8	periodo de validez de esta personalización, formato yyyymmdd
02		descripcion						
02		precio-neto-cliente		x	2	x	8	6 entero + 2 decimales
02		precio-bruto-cliente		x	2	x	8	6 entero + 2 decimales
02		estado						valores: 0= bloqueado 1= activo
02		stock		x			6	

PIEZA_LOCALIZA (SET)
Relaciona una pieza local (owner) con las diferentes localizaciones donde está almacenada (member).

LOCALIZACION (RECORD)
Representa las diferentes áreas del almacén donde se guardan las piezas. Una localización solo puede tener una pieza, pero una pieza puede estar en múltiples localizaciones.

Su estructura es:

nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		LOCALIZACION-CLAVE						
	03	id-area		x			4	
	03	id-seccion		x			6	
	03	id-estanteria		x			6	
02		estado		x				Alguna localización puede estar bloqueada por obras, incidencias, etc 0= cancelada 1= activa

Diccionario (RECORD):
La tabla de diccionario es una tabla física que contiene tablas lógicas, las cuales se utilizan a nivel de aplicación para guardar variables globales, o bien para guardar configuraciones de algunos procesos, o bien para rangos de valores posibles, entre otros. Al ser una estructura fija, la clave se compone de 1 columna que es el identificador de tabla lógica. De 1 a 5 campos que son la clave lógica (y que con el anterior forman la clave real), y hasta 10 columnas genéricas.
Algunas de las tablas lógicas guardadas son:

- estado_linea: identificas los posibles estados de las líneas en los diferentes documentos, pues son comunes.
- estado_basico: utilizado en las entidades que solo pueden estar activas o bloqueadas.
- estado_pedido_vendedor
- estado_pedido_cliente

- estado_backorder
- rango_pedido_vendedor: almacena el último número de pedido hecho a un proveedor, y es utilizado como generador. Además es posible que los usuarios definan el rango manual.
- rango_backorder: igual que el caso anterior, para backorders.
- Rango_pedido_cliente: igual que los anteriores.
- bloqueo_factura: permite bloquear el sistema de facturación entre 2 fechas si es necesario
- Bloqueo_orden: permite bloquear las ordenes de cliente entre 2 fechas.
- codigos_postales: actualmente, solo para España, los códigos válidos.

Su estructura

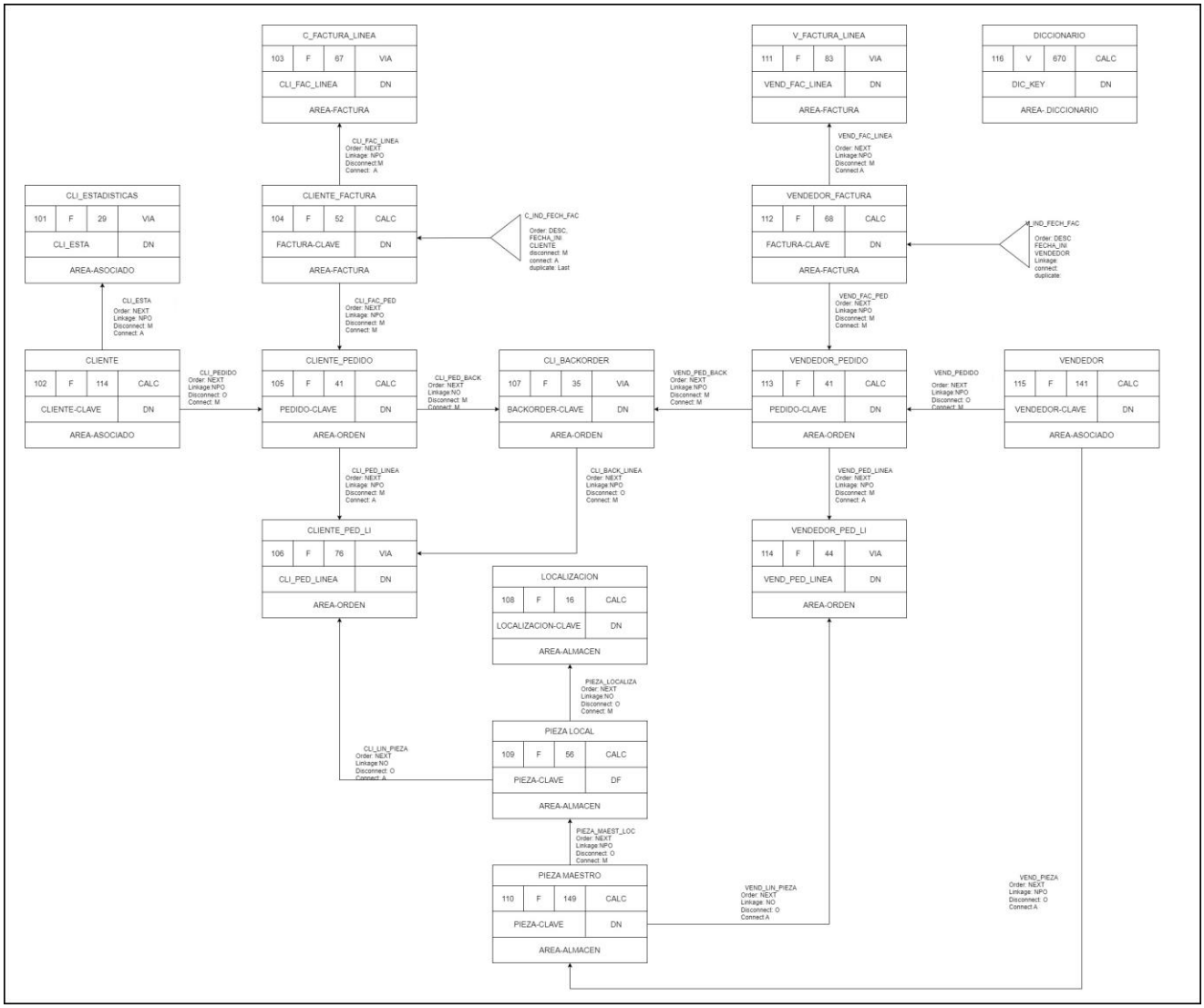
nivel		atributo	Alfanumérico	Numérico	n decimales	Compactado	longitud	notas
02		DICCIONARIO-LCLAVE	x					
	03	tabla	x				20	
	03	clave1	x				20	
	03	clave2	x				20	
	03	clave3	x				20	
	03	clave4	x				20	
	03	clave5	x				20	
02		descripcion	x				50	
02		atributo1	x				50	
02		atributo2	x				50	
02		atributo3	x				50	
02		atributo4	x				50	
02		atributo5	x				50	
02		atributo6	x				50	
02		atributo7	x				50	
02		atributo8	x				50	
02		atributo9	x				50	
02		atributo10	x				50	

Por otro lado, también tenemos los SET que actúan de índices, permitiendo acceder a los datos de forma ordenada.

C_IND_FECH_FAC (SET)
Accede a las facturas de cliente ordenado de forma descendente por los campos fecha_ini (creación factura) e id_cliente, y es utilizado para extraer todas las facturas a un cliente a partir de una fecha dada.

V_IND_FECH_FAC (SET)
Accede a las facturas de proveedor ordenado de forma descendente por los campos fecha_ini e id_vendedor, y es utilizado para extraer todas las facturas de un proveedor a partir de una fecha dada.

Finalmente, la representación mediante un diagrama de Bachmann sería:



5 Diagrama de Bachmann de origen

4. Estrategia y reglas de conversión de estructuras.

Ahora que ya tenemos definida la estructura del caso de estudio a migrar a partir de la información recibida en el punto anterior, los siguientes pasos serán:

- Definir e implementar equivalencias entre las estructuras IDMS y las estructuras relacionales Postgres.

- Revisar el resultado anterior e implementar las modificaciones que fueran necesarias para garantizar que se conserva toda la funcionalidad que anteriormente era posible.

- Revisar el resultado anterior y aplicar las reglas de normalización necesarias para garantizar, como mínimo, la forma normal de Boyce-Codd.

- Identificar elementos adicionales de Postgres que permitan implementar la misma funcionalidad de la base de datos origen, como pueden ser *constraints*, funciones, *triggers*, o vistas.

- Y finalmente, optimizaciones generales.

En este capítulo abordaremos el primer punto y mostraremos una posible solución a la hora de preparar la migración.

4.1 Esquema y subesquema.

Un esquema en IDMS no puede ser mapeado directamente a un esquema PostgreSQL, pues conceptualmente representan objetos diferentes.

Referenciado la documentación oficial, PostgreSQL define un esquema como:

"*A PostgreSQL database cluster contains one or more named databases. Roles and a few other object types are shared across the entire cluster…. A database contains one or more named* schemas*,...* "[1]

Mientras que para IDMS:

"*The schema is a complete description of a database.*" [2]

1. https://www.postgresql.org/docs/current/ddl-schemas.html
2. https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/idms-database/defining-a-database-using-non-sql/schemas-and-subschemas.html

Entendemos por tanto que el esquema IDMS está más cerca del concepto de base de datos (CRETAE DATABASE) que del de esquema.

Por otro lado, IDMS define el subesquema como "*A subschema provides a view of the database as seen by an application program. … The subschema can restrict access to the database by identifying: Areas, records, elements, and sets that are accesible*". [1]

Como conclusión, se propone que para definir la base de datos en Postgres, mapearemos los elementos del esquema IDMS con elementos globales de la base de datos Postgres, y los del subesquema IDMS con los esquemas PostgreSQL (excluyendo intencionadamente el esquema *public* de Postgres).

4.2 Áreas.

Como se ya vimos en la sección 3, los *RECORD* están asignados a un área, de manera que estén físicamente próximos pues su acceso común es frecuente al estar lógicamente relacionados o ser dependientes. Como ejemplo, tendríamos una cabecera de factura y sus líneas.

Otros elementos de la definición de áreas en IDMS como son el rango de páginas o el tamaño de las mismas, no tienen un equivalente directo en PostgreSQL.

Como conclusión, se propone que las áreas participarán conjuntamente con el subesquema IDMS en una equivalencia a los esquemas Postgres.

4.3 RECORD tipo CALC.

Estas entidades son aquellas que pueden existir por sí mismas. Su mapeado respecto a una entidad fuerte o débil en Postgres dependerá de cómo esté definido el SET que las relaciona.

Se propone que:

- Cada registro de tipo CALC se traduzca en una tabla relacional.
- En las relaciones con otras tablas en que actuase actualmente como *owner*, se tratará como entidad fuerte, mientras que si actuase como *member*, se tratará como una entidad débil con dependencia de identidad.

1. https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/idms-database/defining-a-database-using-non-sql/schemas-and-subschemas.html

- Los atributos de la tabla coincidirán en número y tipo equivalente con los del RECORD original, añadiendo los nuevos atributos necesarios para mantener las FOREIGN KEY*s* que sean necesarias.
- La calc-key del *RECORD* se traducirá como la PRIMARY KEY de la tabla
- Los atributos que engloben a otros, es decir, las definiciones de un nivel que tenga subniveles, se traducirán como campos individuales correspondientes a los subniveles.
- A diferencia de IDMS, PostgreSQL no permite tener una PRIMARY KEY duplicada. Por lo tanto, los RECORD que sí permitan en el modelo original tener la calc-key duplicada, en la nueva base de datos usarán un nuevo atributo de tipo SERIAL como PRIMARY KEY, y los campos que participaban en la calc-key original se definirán como un índice.

4.4 RECORD de tipo VIA.

Son aquellos que no pueden existir por sí mismas. Su mapeado correspondería a una entidad débil.

Se propone por tanto:

- Cada registro de tipo VIA se traduzca en una tabla relacional.
- Se añadirá los atributos necesarios y una FOREIGN KEY para poder referenciar a la tabla de que dependan. Se utilizará el nombre del SET o similar con terminación '_fk" como nombre de la *constraint*.
- Se añadirá una PRIMARY KEY cuya definición dependerá de que los campos actuales pueden proveer de una combinación que garantice la unicidad de la clave. De no ser posible, se utilizará un atributo de tipo SERIAL
- Los atributos que engloben a otros, es decir, las definiciones de un nivel que tenga subniveles, se traducirán como campos individuales correspondientes a los subniveles.
- Los atributos de la tabla coincidirán en número y tipo equivalente con los del *RECORD* original, añadiendo los nuevos atributos necesarios para mantener las FOREIGN KEY*s* que sean necesarias como se indica en el punto anterior.

4.5 SETS.

En IDMS los sets pueden representar la relación entre 2 o más entidades, representando una relación 1 a N, de manera que, aunque en cuanto a lógica una relación pueda ser 1 a 1 o incluso fija 1 a 5 (como ejemplo), el DBMS siempre admitirá N debiendo controlar el

límite de forma programática. Los SET implementan por tanto una relación 1 (owner) a muchos (member).
Además los SET especifican en su declaración si se puede desplazar el puntero hacia adelante o atrás, o si se puede obtener el registro owner estando situados en el member. Y también el orden.

Como conclusión:

- No se permite la navegación (NEXT/FIRST/etc..) entre tuplas pero PostgreSQL sí permite recuperar todas las que dependan por su clave de una entidad padre, cubriendo así esta funcionalidad.
- No se permite una ordenación en el acceso por definición de la FOREIGN KEY. Si fuera necesaria, esta puede conseguirse mediante funciones, vistas, o de forma programática.
- Identificaremos los SETS como las claves foráneas propias de una base de datos relacional, siendo la tabla equivalente VIA la que tendrá clave foránea hacia la CALC.
- En los SETS entre 2 RECORD de tipo CALC, como norma general la entidad que actúa como MEMBER tendrá clave foránea a la OWNER.
- Al introducir claves foránea, puede ser necesario añadir los campos referenciados a la tabla, ya que en IDMS no es requisito para establecer la relación del SET, pero en PostgreSQL sí para definir la clave foránea.

4.6 INDEXED SETS.

Un caso particular de SET es aquél que no referencia ningún RECORD owner. En su lugar, permite acceder de forma ordenada a los datos de un registro.

El concepto de índice en PostgreSQL no corresponde con el de IDMS.

Como conclusión:

- Definiremos en Postgres los índices que sean necesarios para un correcto rendimiento. El uso de un índice u otro, será normalmente decidido por el mismo SGBD. Éstos, por tanto, no están relacionados con los INDEXED SET de IDMS.
- En caso de necesitar acceder a datos de forma ordenada, definiremos vistas.

4.7 Atributos compuestos.

Hemos visto que en la jerarquía de datos de IDMS un atributo se puede descomponer en otros según el nivel asignado 01, 02, … N (de hecho, toda tabla no es más que la descomposición del nivel 01

correspondiente al RECORD). Pero si trasladamos el nivel superior, nuestra base de datos relacional no cumpliría la primera forma normal. Podríamos argumentar la posibilidad de definir estos niveles como TYPE de PostgreSQL, pero para la realización de este trabajo se ha optado por generar una columna por cada atributo del nivel más bajo. Así, por ejemplo la dirección del RECORD VENDEDOR generará como mínimo las columnas calle, numero, complemento y cpostal en la futura tabla.

4.8 Limitación de valores.

Hemos visto en la descripción de los diferentes RECORD que algunos tienen actualmente limitado sus valores (ya sea por aplicación o por sus correspondencias con el RECORD diccionario).

Recordemos que a nivel de base de datos IDMS no está realizando estas validaciones (no hay SET), sino que son a nivel programático mediante la tabla diccionario. En caso de desear implementarlo a nivel de bases de datos, las estrategia que adoptaremos dependerá en cada caso, pero a grandes rasgos:

- Si un atributo cumple un rango de valores estable en el tiempo, este puede definirse como un TYPE y una ENUM en Postgres, o usar una tabla diccionario.
- Si los valores pueden variar con frecuencia, pero no tienen peso en su uso directo, los mantendremos como diccionario
- Si los valores pueden variar, tienen peso/signifcado, y debe mantenerse su relación, utilizaremos tablas dedicadas.

4.9 Casos particulares: diccionario.

Es común en muchas aplicaciones actuales el uso del patrón diccionario que a cada clave le asigna un valor. En algunas aplicaciones legacy, como es en nuestro caso de estudio, no se estimó necesario definir tablas completas solo por mantener su validación referencial (es decir, solo para limitar los valores disponibles), con el sobrecoste que esto podía acarrear tanto en mantenimiento como en rendimiento.

En su lugar, existe una estructura parecida pero en realidad más compleja que permite definir una serie de tablas lógicas y sus valores. Solo se usa cuando añadir una tabla no aportaría mejora a la base de datos IDMS ni a su funcionalidad. Un ejemplo puede ser nombres de procesos usados por la aplicación que corre sobre nuestra base de datos y horarios de su planificación, descripciones en múltiples lenguaje para alguno de los campos usados en los informes extraídos, etc.

Esta estructura, este símil de diccionario, en su forma más básica consta de un campo que identifica la tabla lógica. Un máximo de X

campos que hacen de clave, y un máximo de Y campos que se retornan cuando el proceso encuentra coincidencia para la combinación de tabla y X claves.

La estrategia que definiremos para mapear estos diccionarios será:

- Se estudiará de forma individual cada tabla lógica del RECORD Diccionario, su significado, y sus posibles dependencias
- En general, aplicaremos que todo par de tipo “tabla lógica + 1 clave-> 1 valor” se mantendrá en una nueva tabla de diccionario,
- y aquellos que requieran claves más complejas, o conjunto de atributos más complejo, se generarán como tabla independiente.

4.10 Tipos de datos.

Si bien se tendrá en cuenta cada tabla y sus características, aplicaremos las siguientes reglas generales para la conversión de tipos:

Dato en IDMS	Dato en Postgres	Justificación
PIC X(n)	VARCHAR(n)	Donde n es la longitud.
PIC 9(n)	SMALLINT	cuando N sea menor o igual a 4, ya que el máximo valor de SMALLINT es 32767 y no 99999 si usáramos 5 dígitos completos Es necesario añadir una restricción de que no acepte números negativos ya que el formato origen no es "signed".
PIC 9(n)	INTEGER	cuando N sea menor o igual a 9, ya que el máximo valor de INTEGER ES 2147483647 y no 9999999999 si usáramos 10 dígitos completos Es necesario añadir una restricción de que no acepte números negativos ya que el formato origen no es "signed".
PIC 9(n)	BIGINT	En cualquier caso cuando N sea mayor o igual a 10.
PIC 9(8)	DATE/TIMESTAMP	Cuando un campo numérico represente una fecha, lo guardaremos como DATE o TIMESTAMP, dependiendo del uso y significado del campo.
PIC 9(1) o PIC X(1)	BOOLEAN	Cuando los atributos valgan un único byte para indicar 2 estados, se traducirá como un BOOLEAN
PIC S9(n)	SMALLINT/ INTEGER/ BIGINT	Igual que los casos anteriores, pero al ser un valor con signo no es necesario añadir ninguna restricción.
PIC 9(n)V9(m)	NUMERIC(n+m, m)	Permite cubrir diferente casuística de los formatos de número de nuestra base de datos. Es necesario añadir una restricción de que no acepte números negativos ya que el formato origen no es "signed".
PIC 9(n)V9(m) COMP	NUMERIC(n+m, m)	Los datos compactados en IDMS se tratarán como numéricos normales en Postgres.

4.11 Estructuras adicionales.

Los puntos anteriores cubren ya nuestra definición inicial de la base de datos origen. Pero existen otros elementos[1] que hay que tener en cuenta a la hora de migrar desde IDMS:

- *LOGICAL RECORD*: el cual permite agrupar varios record del subesquema. Otros conceptos asociados son los PATH y PATH GROUP. Al definir un LOGICAL RECORD se indican que elementos forman parte y que PATH (es decir, que serie de acciones concatenadas) se realizará. Esta funcionalidad podría simularse con una función de Postgres, si bien no se encuentra en nuestra base de datos original.
- *Database procedure*: los cuales permiten definir procesos que transformen los datos antes o despues de guardarlos. Una estructura similar, pero mucho más amplia en Postgre, sería el uso de *triggers*.
- *Record synonims*: IDMS permite dar más de un nombre a un record. Podríamos simular esta funcionalidad mediante una vista Postgres.

4.12 Consideraciones adicionales.

Un elemento propio a tener en cuenta de las bases de datos Postgres sin equivalencia son los SERIAL.

Respecto a su uso, podemos identificar 2 estrategias:

- Que toda clave de una tabla sea un SERIAL (o un IDENTITY), y definir índices para las claves lógicas que correspondan a los atributos de las calc-key de origen
- Alternativamente, mantener los atributos originales como PRIMARY KEYS y usar solo SERIAL cuando este uso resulte en incompatibilidad con Postgres (por ejemplo el Record PIEZA_LOCAL).

Se podría hacer un estudio respecto si el uso de la clave compuesta como PRIMARY KEY implica alguna mejora de rendimiento en algunas consultas. Para nuestro caso, hemos decidido centrarnos en la segunda estrategia pues tenemos en cuenta que el volumen de operaciones y registros en las tablas puede ser de varios cientos de miles o muy pocos millones, lo cual se puede considerar un volumen bajo, y que facilitará el uso de las aplicaciones actuales ejecutándose sobre la base de datos minimizando cualquier impacto por los cambios.

Verificaremos esta decisión durante las pruebas de rendimiento que se hagan tras finalizar la implementación inicial.

1. https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/idms-database/subschema-statements.html

5. Migración.

Abordaremos la migración a PostgreSQL mediante un proceso iterativo en 3 etapas:

1º. Iteración inicial o migración en “crudo”: en este paso inicial aplicaremos a la estructura inicial las reglas definidas en el capítulo 4.
2º. Iteración media o verificación de la corrección de la solución: se realizarán los cambios necesarias para asegurar que la base de datos sigue manteniendo toda funcionalidad original y se verificará y corregirá que se cumpla como mínimo la forma normal de Boyce-Codd.
3º. Iteración final o adaptaciones a PostgreSQL: este es el paso final en el que de ser necesario se justificará y definirá estructuras que no existían “as is” en IDMS, como pueden ser vistas, tipos o funciones entre otras.

5.1 Iteración inicial.

Éste es el proceso de aplicar directamente las reglas definidas.

5.1.1 Schema y subschema IDMS.

Crearemos una base de datos inicial con las opciones básicas, que corresponden con la información que disponemos. Es posible que, tras la migración, deba analizarse las diferentes opciones y parámetros como puede ser especificar un número máximo de conexiones. Esto estaría fuera del alcance actual del presente proyecto.

```
CREATE DATABASE warehouse;
```

5.1.2 Areas IDMS.

Las áreas las migraremos como esquemas. Viendo el diagrama de Bachmann de la base de datos inicial identificamos 5 áreas: AREA-ASOCIADO, AREA-FACTURA, AREA-ORDEN, AREA-DICCIONARIO y AREA-ALMACEN. Obtendremos los siguientes esquemas.

```
CREATE SCHEMA IF NOT EXISTS asociado;
CREATE SCHEMA IF NOT EXISTS factura;
CREATE SCHEMA IF NOT EXISTS orden;
CREATE SCHEMA IF NOT EXISTS almacen;
CREATE SCHEMA IF NOT EXISTS diccionario;
```

5.1.3 RECORDs y SETs.

Nótese que en esta sección denominaremos a los RECORD por sus nombres en mayúsculas, mientras que las tablas estarán en minúscula.

Esto obedece además a que internamente PostgreSQL guarda los nombres en minúscula si no se ha definido entre comillas.

Tendremos en cuenta de forma general:

- Ninguna de las tablas se definirá como TEMP.
- Para ninguna de las tablas se definirá particionado ya que su tamaño no es crítico.[1]
- Dado el tamaño de la base de datos, y el uso actual de diferentes esquemas, no se definirá ningún *tablespace* al crear la tabla.

CLIENTE es de tipo CALC, por lo tanto corresponderá a una tabla. Por cada campo de nivel 02 en IDMS (es decir, aquellos que engloban otros atributos de nivel mayor que 02) se crean tantas columnas como subniveles (es el caso del atributo dirección). Se aplica la transformación de tipos definida en el apartado 4.8. Obtenemos:

```
CREATE TABLE IF NOT EXISTS asociado.cliente (
      cif                                VARCHAR(9),
      nombre                             VARCHAR(30),
      direccion_factura_calle            VARCHAR(30),
      direccion_factura_numero           SMALLINT,
      direccion_factura_complemento      VARCHAR(50),
      direccion_factura_cpostal          SMALLINT,
      direccion_envio_calle              VARCHAR(30),
      direccion_envio_numero             SMALLINT,
      direccion_envio_complemento        VARCHAR(50),
      direccion_envio_cpostal            SMALLINT,
      fecha_alta                         DATE,
      fecha_baja                         DATE,
      estado                             VARCHAR(1),
      CONSTRAINT cliente_pk PRIMARY KEY(cif)
);
```

CLI_ESTADISTICAS es de tipo VIA y se relaciona mediante el SET CLI_ESTA con CLIENTE. Por lo tanto, la nueva tabla tendrá una clave foránea hacia asociado.cliente. El nivel 02 (estadistica-periodo) se descompondrá en tantos atributos como subcampos en IDMS. El resultado de esta descomposición, junto con el atributo de clave foránea, formarán la nueva clave primaria. Los valores numéricos con decimales se transforman a NUMERIC tal como se indicó en las reglas de migración, que además especificaremos no pueden ser NULL. Formará también parte del esquema asociado.

```
CREATE TABLE IF NOT EXISTS asociado.cliente_estadistica(
      cif                          VARCHAR(9),
      anyo                         SMALLINT,
      mes                          SMALLINT,
      pedidos_realizado            SMALLINT,
      pedidos_acumulado            INTEGER,
      cuantia                      NUMERIC(8,2)   NOT NULL,
      cuantia_acumulada            NUMERIC(12,2)  NOT NULL,
      fecha_ultimo_pedido          DATE,
      CONSTRAINT cliente_estadistica_pk PRIMARY KEY(cif,anyo,mes),
      CONSTRAINT cli_esta_fk FOREIGN KEY (cif)
                  REFERENCES asociado.cliente (cif)
);
```

1. https://hevodata.com/learn/postgresql-partitions/#t10

VENDEDOR es de tipo CALC y MEMBER en ningún SET por lo que no tiene claves foráneas. Pertenece al esquema asociado. Si aplicamos las reglas de migración obtenemos:

```
CREATE TABLE IF NOT EXISTS asociado.vendedor(
      cif                                VARCHAR(9),
      nombre                             VARCHAR(30),
      direccion_factura_calle            VARCHAR(30),
      direccion_factura_numero           SMALLINT,
      direccion_factura_complemento      VARCHAR(50),
      direccion_factura_cpostal          SMALLINT,
      fecha_alta                         DATE    NOT NULL,
      fecha_baja                         DATE,
      estado                             BOOLEAN NOT NULL,
      CONSTRAINT vendedor_pk PRIMARY KEY(cif)
);
```

PIEZA_MAESTRO es de tipo CALC por lo que originará una tabla, siendo su clave un nivel 02 que se descompone en 2 niveles 03 por lo que originará una clave primaria compuesta en Postgres. Además, forma parte del SET VEND_PIEZA como member por lo que aladirá una clave foránea a vendedor. Tras aplicar las reglas de migración definidas:

```
CREATE TABLE IF NOT EXISTS almacen.pieza_maestro(
      id_pieza              VARCHAR(20),
      id_vendedor           VARCHAR(9),
      descripcion           VARCHAR(50),
      precio_neto_vend      NUMERIC(8,2)   NOT NULL,
      precio_bruto_vend     NUMERIC(8,2)   NOT NULL
      estado                BOOLEAN        NOT NULL,
      CONSTRAINT pieza_maestro_pk   PRIMARY KEY (id_pieza, id_vendedo),
      CONSTRAINT vend_pieza_fk      FOREIGN KEY (id_vendedor)
                 REFERENCES asociado.vendedor (cif),
      );
```

PIEZA_LOCAL se relaciona con PIEZA_MAESTRO mediante el SET PIEZA_MAEST_LOC, pero en este caso se permiten duplicados (según las fechas de validez) por lo que no puede usarse la clave compuesta como primaria, por lo que se introducirá un campo de tipo ‘serial’ tal como se indicaba en las reglas del apartado 4.3, y los que formaban la clave anterior como un índice:

```
CREATE TABLE IF NOT EXISTS almacen.pieza_local(
      id_pieza_gen          SERIAL,
      id_pieza              VARCHAR(20)    NOT NULL,
      id_vendedor           VARCHAR(9)     NOT NULL,
      fecha_vigencia_ini    DATE           NOT NULL,
      fecha_vigencia_fin    DATE           NOT NULL,
      descripcion           VARCHAR(100),
      precio_neto_cliente   NUMERIC(8,2)   NOT NULL,
      precio_bruto_cliente  NUMERIC(8,2)   NOT NULL,
      estado                BOOLEAN        NOT NULL,
      stock                 INTEGER        NOT NULL,
      CONSTRAINT pieza_local_pk     PRIMARY KEY (id_pieza_gen),
      CONSTRAINT pieza_maest_loc_fk        FOREIGN KEY (id_pieza, id_vendedor)
                 REFERENCES  almacen.pieza_maestro (id_pieza, id_vendedor),
      CONSTRAINT precio_neto_cliente_chk  CHECK (precio_neto_cliente >= 0),
      CONSTRAINT precio_bruto_cliente_chk CHECK (precio_neto_cliente >= 0),
      CONSTRAINT stock_chk CHECK (stock >= 0)
);

CREATE INDEX pieza_local_ind ON almacen.pieza_local(id_pieza, id_vendedor );
```

De forma similar al anterior caso, LOCALIZACION es una tabla CALC que se relaciona con PIEZA_LOCAL. Tiene un nivel 02 que es la clave, por lo que aplicando las reglas se descompone en los valores inferiores que darán lugar a una clave compuesta. Además, el set PIEZA_LOCALIZA en el que participa como MEMBER se traducirá en una clave foránea, que apuntará al SERIAL que hemos creado ya que la pieza local admitía duplicados en IDMS. Destacar que en este caso la parte al no formar parte de la clave puede ser null (es decir, una zona de almacenamiento puede estar vacía).

```
CREATE TABLE IF NOT EXISTS almacen.localizacion(
      id_area        VARCHAR(4),
      id_seccion     VARCHAR(6),
      id_estanteria  VARCHAR(6),
      id_parte       INTEGER,
      estado         BOOLEAN,
      CONSTRAINT localizacion_pk
                  PRIMARY KEY (id_area, id_seccion, id_estanteria),
      CONSTRAINT pieza_localiza_fk  FOREIGN KEY(id_parte)
                 REFERENCES almacen.pieza_local(id_pieza_gen)
);
```

CLIENTE_FACTURA es de tipo CALC. Nótese que tiene el atributo cliente, pero no tiene un SET directo a cliente (en IDMS la relación se mantiene a navegando desde CLIENTE_FACTURA a CLIENTE_PEDIDO y luego obteniendo el OWNER en CLIENTE), por lo que aplicando las normas en esta primera iteración no definiremos esta *foreign key*. Su sentencia SQL sería:

```
CREATE TABLE IF NOT EXISTS factura.cliente_factura(
      id_factura             INTEGER,
      id_cliente             VARCHAR(9)     NOT NULL,
      fecha_factura          DATE           NOT NULL,
      estado                 VARCHAR(1)     NOT NULL,
      fecha_estado           DATE           NOT NULL,
      total_neto             NUMERIC(8,2),
      total_bruto            NUMERIC(8,2),
      CONSTRAINT cliente_factura_pk        PRIMARY KEY (id_factura),
      CONSTRAINT id_factura_chk
                 CHECK (id_factura > 0 AND id_factura < 999999999)
);
```

En IDMS disponemos de un OREDERD SET C_IND_FECH_FAC que permite acceder a los datos de forma ordenada. Como se indica en las reglas, definiremos una VISTA equivalente. En fases posteriores revisaremos la idoneidad de los atributos de la vista. Adicionalmente, como ninguno de los campos de ordenamiento forma parte de la clave, definiremos un índice.

```
CREATE INDEX IF NOT EXISTS  c_ind_fech_fac_idx ON factura.cliente_factura
(fecha_factura, id_cliente);

CREATE VIEW c_ind_fech_fac_view AS
SELECT * FROM factura.cliente_factura
ORDER BY fecha_factura desc, id_cliente DESC;
```

C_FACTURA_LINEA es una VIA que se relaciona con CLIENTE_FACTURA mediante el SET CLI_FAC_PED, pero que no tiene un atributo que refleje el id de factura. Por ello siguiente las reglas definidas, añadiremos dicho campo. Añadiremos también la

restricciones para cumplir que el máximo número de línea sea 999 y la exclusividad entre los atributos id-pedido-orig e id-concepto.

```
CREATE TABLE IF NOT EXISTS factura.c_factura_linea (
      id_factura            INTEGER,
      num_linea             SMALLINT,
      id_pedido_orig        INTEGER,
      id_concepto           VARCHAR(30),
      fecha                 DATE          NOT NULL,
      estado                VARCHAR(1)    NOT NULL,
      fecha_estado          DATE          NOT NULL,
      total_neto            NUMERIC(8,2)  NOT NULL,
      total_bruto           NUMERIC(8,2)  NOT NULL,
      CONSTRAINT c_factura_linea_pk PRIMARY KEY (id_factura, num_linea),
      CONSTRAINT cli_fac_linea_fk   FOREIGN KEY (id_factura)
                REFERENCES factura.cliente_factura(id_factura),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT pedido_concepto_chk  CHECK (
             (id_pedido_orig IS NULL and id_concepto IS NOT NULL)       OR
             (id_pedido_orig IS NOT NULL and id_concepto IS NULL))
);
```

De forma parecida a la factura de cliente, VENDEDOR_FACTURA es una CALC que actúa siempre como owner, por lo que no tendrá clave externa. Si aplicamos las transformaciones acordadas y la limitación de longitud de la clave para ser equivalente al origen:

```
CREATE TABLE IF NOT EXISTS factura.vendedor_factura(
      id_factura            INTEGER,
      id_vendedor           VARCHAR(9)    NOT NULL,
      fecha_factura         DATE          NOT NULL,
      estado                VARCHAR(1)    NOT NULL,
      fecha_estado          DATE          NOT NULL,
      total_neto            NUMERIC (8,2),
      total_bruto           NUMERIC (8,2),
      total_neto_conf       NUMERIC (8,2),
      total_bruto_conf      NUMERIC (8,2),
      CONSTRAINT vendedor_factura_pk       PRIMARY KEY (id_factura),
      CONSTRAINT id_factura_chk
                CHECK (id_factura > 0 AND id_factura < 999999999)
);
```

También, de forma parecida al cliente, esta tabla dispone de un INDEXED SET en IDMS, por lo que transformaremos como una vista y un índice.

```
CREATE INDEX IF NOT EXISTS  v_ind_fech_fac_idx ON factura.vendedor_factura
(fecha_factura, id_vendedor);

CREATE VIEW v_ind_fech_fac_view AS
SELECT * FROM factura.vendedor_factura
ORDER BY fecha_factura desc, id_vendedor desc;
```

V_FACTURA_LINEA es de tipo VIA, y se relaciona mediante el SET VEND_FAC_LINEA con la CALC VENDEDOR_FACTURA, lo cual se traducirá en una foreign key y será necesario para ello añadir un atributo que forme la clave compuesta. Aplicando las regles de transformación:

```
CREATE TABLE IF NOT EXISTS factura.v_factura_linea(
      id_factura            INTEGER,
      num_linea             SMALLINT,
      id_pedido_orig        INTEGER,
      id_concepto           VARCHAR(30),
      fecha                 DATE          NOT NULL,
      estado                VARCHAR(1)    NOT NULL,
      fecha_estado          DATE          NOT NULL,
```

```
      total_neto            NUMERIC(8,2)   NOT NULL,
      total_bruto           NUMERIC(8,2)   NOT NULL,
      CONSTRAINT v_factura_linea_pk PRIMARY KEY (id_factura, num_linea),
      CONSTRAINT vend_fac_linea_fk FOREIGN KEY(id_factura)
                REFERENCES factura.vendedor_factura (id_factura),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT pedido_concepto_chk  CHECK (
             (id_pedido_orig IS NULL     AND id_concepto IS NOT NULL)   OR
             (id_pedido_orig IS NOT NULL AND id_concepto IS NULL))
);
```

CLIENTE_PEDIDO es de tipo CALC y se relaciona con CLIENTE mediante el SET CLI_PEDIDO y con FACTURA mediante el SET CLI_FAC_PED. Dado que todas son CALC en IDMS, y se especifica que pedido-clave es único entre todos los pedidos, la clave foránea hacia la tabla cliente no formará parte de la clave en PostgreSQL, e igual sucede con factura. El campo prioridad solo tiene significado cuando su valor es 1, por lo tanto actúa como un booleano. Se creará dentro del esquema correspondiente al mapeado del AREA IDMS.
Tras aplicar las reglas definidas en el punto 4 obtendríamos una estructura equivalente inicial:

```
CREATE TABLE IF NOT EXISTS orden.cliente_pedido(
      id_pedido             INTEGER,
      cif                   VARCHAR(9),
      fecha                 DATE,
      estado                VARCHAR(1),
      fecha_estado1         DATE           NOT NULL,
      fecha_estado2         TIME           NOT NULL,
      prioridad             BOOLEAN        NOT NULL,
      total_neto            NUMERIC(8,2)   NOT NULL,
      total_bruto           NUMERIC(8,2)   NOT NULL,
      id_factura            INTEGER,
      CONSTRAINT cliente_pedido_pk  PRIMARY KEY (id_pedido),
      CONSTRAINT cli_pedido_fk            FOREIGN KEY (cif)
                REFERENCES asociado.cliente (cif),
      CONSTRAINT cli_fac_ped_fk           FOREIGN KEY (id_factura)
                REFERENCES factura.cliente_factura (id_factura),
      CONSTRAINT id_pedido_chk CHECK (id_pedido > 0 AND id_pedido < 999999999),
      CONSTRAINT total_neto_chk  CHECK (total_neto  >= 0),
      CONSTRAINT total_bruto_chk CHECK (total_bruto >= 0)
);
```

CLI_BACKORDER es una VIA compleja, como la funcionalidad a la que corresponde. Su estructura básica es parecida a un pedio de cliente, pues se origina a partir de estos. Mantiene por tanto la relación con el pedido, como con sus líneas (no todas las líneas del pedido original se añaden a la backorder, solo aquella para las que no ha habido stock). Además, la relación del SET VEND_PED_BACK es debido a que un pedido de vendedor puede estar compuesto por varias backorders (es un proceso programado que las agrupa al final del día para hacer la solicitud de piezas a un proveedor).

En esta primera iteración no debatiremos si el diseño es correcto para Postgres, o si la referencia a pedido de cliente es innecesaria al tener la referencia a las líneas, o si sería más conveniente tener una tabla de líneas de backorder independiente, realizaremos la conversión solo aplicando las reglas. Obtendríamos:

```
CREATE TABLE IF NOT EXISTS orden.cli_backorder(
      backorder_clave       INTEGER,
      id_pedido_orig        INTEGER        NOT NULL,
      id_pedido_vendedor    INTEGER,
      fecha                 DATE           NOT NULL,
      estado                VARCHAR(1)     NOT NULL,
      fecha_estado          DATE           NOT NULL,
      CONSTRAINT cli_backorder_pk   PRIMARY KEY (backorder_clave),
      CONSTRAINT cli_back_ped_fk    FOREIGN KEY (id_pedido_orig)
                REFERENCES orden.cliente_pedido(id_pedido),
      CONSTRAINT vend_ped_back_fk   FOREIGN KEY (id_pedido_vendedor)
                REFERENCES orden.vendedor_pedido(id_pedido)
);
```

CLIENTE_PED_LI es de tipo VIA y se relaciona con CLIENTE_PEDIDO mediante el SET CLI_PED_LINEA. Por lo tanto en la nueva tabla tendrá una clave foránea a cliente_pedido, que formará junto con el atributo num_linea la clave primaria. Se añadirán también las restricciones especificas a los campos respecto al número de líneas o la cantidad de piezas.
Además CLIENTE_PED_LI también se relaciona con PIEZA_LOCAL mediante el SET CLI_LIN_PIEZA, que resultará en otra foreign key a la tabla de piezas local la cual no puede ser NULL. Y con una backorder por el set CLI_PED_BACK, que es opcional.

```
CREATE TABLE IF NOT EXISTS orden.cliente_ped_li(
      id_pedido                     INTEGER,
      num_linea                     SMALLINT,
      id_pieza                      INTEGER        NOT NULL,
      cantidad                      SMALLINT       NOT NULL,
      coste_neto_unidad             NUMERIC(7,2)   NOT NULL,
      coste_bruto_unidad            NUMERIC(7,2)   NOT NULL,
      total_neto                    NUMERIC(8,2)   NOT NULL,
      total_bruto                   NUMERIC(8,2)   NOT NULL,
      backorder                     INTEGER,
      CONSTRAINT cliente_ped_li_pk  PRIMARY KEY (id_pedido,num_linea),
      CONSTRAINT cli_ped_linea_fk           FOREIGN KEY (id_pedido)
                REFERENCES orden.cliente_pedido(id_pedido),
      CONSTRAINT cli_lin_pieza_fk           FOREIGN KEY (id_pieza)
                REFERENCES almacen.pieza_local(id_pieza_gen),
      CONSTRAINT cli_back_linea_fk          FOREIGN KEY (backorder)
                REFERENCES orden.cli_backorder(backorder_clave),
      CONSTRAINT num_linea_chk      CHECK (num_linea > 0 and num_linea < 1000),
      CONSTRAINT cantidad_chk       CHECK (cantidad  > 0 and cantidad  < 1000),
      CONSTRAINT coste_neto_unidad_chk CHECK (coste_neto_unidad>=0),
      CONSTRAINT coste_bruto_unidad_chk CHECK (coste_bruto_unidad>=0),
      CONSTRAINT total_neto_chk CHECK (total_neto>=0),
      CONSTRAINT total_bruto_chk CHECK (total_bruto>=0)
);
```

VENDEDOR_PEDIDO es de tipo CALC y se relaciona como member mediante el SET VEND_PEDIDO con VENDEDOR y con VEND_FAC_PED con VENDEDOR_FACTURA, lo que dará lugar a claves foráneas.

```
CREATE TABLE IF NOT EXISTS orden.vendedor_pedido(
      id_pedido             INTEGER,
      cif                   VARCHAR(9),
      fecha                 DATE,
      estado                VARCHAR(1),
      fecha_estado1         DATE           NOT NULL,
      fecha_estado2         TIME                   NOT NULL,
      total_neto            NUMERIC(8,2)   NOT NULL,
      total_bruto           NUMERIC(8,2)   NOT NULL,
      id_factura            INTEGER,
      CONSTRAINT cliente_pedido_pk  PRIMARY KEY (id_pedido),
      CONSTRAINT cli_pedido_fk              FOREIGN KEY (cif)
```

```
                REFERENCES asociado.vendedor (cif),
      CONSTRAINT vend_fac_ped_fk          FOREIGN KEY (id_factura)
                REFERENCES factura.vendedor_factura(id_factura),
      CONSTRAINT id_pedido_chk CHECK (id_pedido > 0 AND id_pedido < 999999999),
      CONSTRAINT total_neto_chk  CHECK (total_neto  >= 0),
      CONSTRAINT total_bruto_chk CHECK (total_bruto >= 0)
);
```

VENDEDOR_PED_LI es una VIA que se relaciona con las CLIENTE_PEDIDO y PIEZA MAESTRO, por lo que habrá que añadir los atributos necesarios.

```
CREATE TABLE IF NOT EXISTS orden.vendedor_ped_li(
      id_pedido             INTEGER,
      num_linea             SMALLINT,
      id_pieza              VARCHAR(20)    NOT NULL,
      id_vendedor           VARCHAR(9)     NOT NULL,
      cantidad              SMALLINT,
      total_neto            NUMERIC(8,2)   NOT NULL,
      total_bruto           NUMERIC(8,2)   NOT NULL,
      CONSTRAINT vendedor_ped_li_pk        PRIMARY KEY (id_pedido, num_linea),
      CONSTRAINT cli_ped_linea_fk   FOREIGN KEY (id_pedido)
                REFERENCES  orden.vendedor_pedido(id_pedido),
      CONSTRAINT cli_lin_pieza_fk   FOREIGN KEY (id_pieza, id_vendedor)
                REFERENCES  almacen.pieza_maestro(id_pieza, id_vendedor),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT cantidad_chk  CHECK (cantidad  > 0 AND cantidad  < 999)
);
```

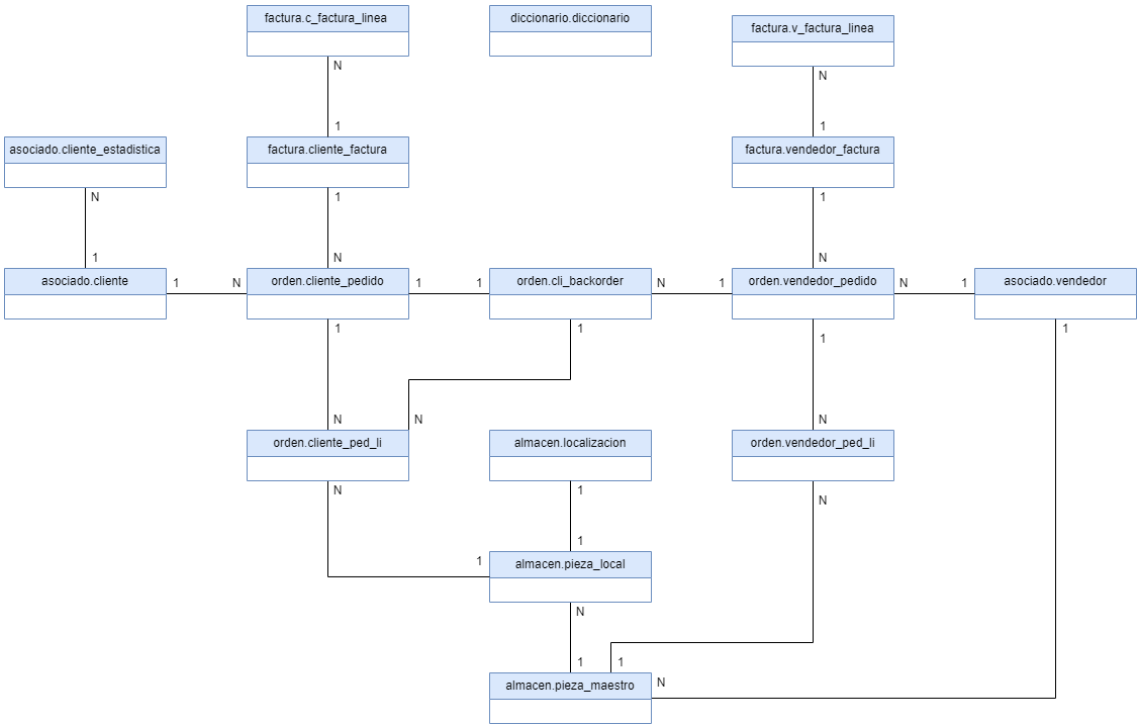
Finalmente tendríamos el RECORD Diccionario. Este es un caso que requerirá especial atención. Un patrón de tipo ordinario relacionaría una clave con un valor, pero vemos que el uso de este RECORD es más complejo, admitiendo claves de variable en número (entre 1 y 5 atributos) que retornan también valores relacionados variables en número (hasta 10 atributos). Además, para nuestro caso de estudio faltaría decidir qué valores se quieren implementar para que mantengan una validación débil a través del diccionario, y cuales se desea que se implementen mediante relaciones entre tablas. En esta iteración. Realizaremos la conversión directa. Como en Postgresql los elementos de la clave primaria no pueden ser nulos, utilizaremos un serial. Para mantener la misma integridad, será necesario también definir un índice único sobre los antiguos campos clave. Obtendríamos:

```
CREATE TABLE diccionario.diccionario(
      id              SERIAL,
      tabla           VARCHAR(20)    NOT NULL,
      clave1          VARCHAR(20),
      clave2          VARCHAR(20),
      clave3          VARCHAR(20),
      clave4          VARCHAR(20),
      clave5          VARCHAR(20),
      descripcion     VARCHAR(50),
      atributo1       VARCHAR(50),
      atributo2       VARCHAR(50),
      atributo3       VARCHAR(50),
      atributo4       VARCHAR(50),
      atributo5       VARCHAR(50),
      atributo6       VARCHAR(50),
      atributo7       VARCHAR(50),
      atributo8       VARCHAR(50),
      atributo9       VARCHAR(50),
      atributo10      VARCHAR(50),
      CONSTRAINT id_pk PRIMARY KEY (id)
);
```

```
CREATE UNIQUE INDEX diccionario_idx ON diccionario.diccionario (tabla, clave1,
clave2, clave3, clave4, clave5);
```

5.1.4 Esquema ER inicial.

Tras aplicar las reglas del punto 5.1, obtendríamos un esquema ER en bruto tal que:



6. Esquema ER 1º iteración

5.2 Iteración media.

Durante la planificación inicial se estimó que tras aplicar las reglas de conversión se procedería a la normalización de la base de datos postgres resultante, pero durante dicho ejercicio pronto fue evidente que las entidades obtenidas mantenían una estructura E-R anómala. Esto es así porque en el paso anterior hemos definido unas estructuras SQL equivalentes a nuestras entidades en la base de datos original, pero IDMS (dejando de lado IDMS/R o IDMS/SQL) es una base de datos jerárquica, no relacional. Por lo que como primer paso de esta iteración realizaremos una revisión de las entidades y propondremos una primera aproximación al esquema entidad-relación.

A continuación, procederemos a la normalización para evitar anomalías como la redundancia de datos, mejorar la flexibilidad, mejorar el rendimiento, proteger la integridad de los datos, y facilitar el mantenimiento. Para nuestro caso de estudio de la migración de la base de datos se nos ha solicitado que ésta debe estar, al menos, en la forma normal de Boyce-Codd.

Se considera que una base de datos está en una formal normal (FN en adelante) cuando cumple los siguientes criterios:

- Primera forma normal (1FN):
  - Todos los atributos son atómicos.
  - Los campos no clave son dependientes funcionalmente de la clave.
  - Existe independencia en el orden de filas y columnas.
  - No hay filas duplicadas
- Segunda formal normal (2FN):
  - Cumple 1FN
  - Todo atributo que no forma parte de una clave candidata depende completamente de toda la clave candidata.
- Tercera forma normal (3FN):
  - Cumple 2FN
  - Ningún atributo que no forma parte de una clave candidata depende funcionalmente de una parte de una clave candidata.
- Forma normal de Boyce-Codd (FNBC):
  - Cumple 3FN
  - Toda dependencia no trivial tiene una clave candidata como determinante.
- Cuarta forma normal (4FN):
  - Cumple FNBC o 3FN
  - No presenta dependencias multivaluadas independientes.
- Quinta forma normal (5FN):
  - Cumple 4FN
  - Toda dependencia de unión en la tabla es implicada por las claves candidatas.

Como consideración final, tendremos en cuenta que en ocasiones es necesario realizar una desnormalización para mejorar el rendimiento o cubrir funcionalidades muy específicas. De considerarse en nuestro caso de estudio esta característica, será debidamente justificada.

5.2.1 Adaptaciones adicionales al modelo relacional.

En este esquema pueden identificarse las siguientes dependencias anómalas:

- Los clientes_pedido están ligados a la factura a la que pertenece para permitir la navegación en IDMS, y adicionalmente se les menciona en las tuplas de c_factura_linea. Para mantener correctamente la integridad, trasladaremos la dependencia de cliente_factura a c_factura_linea. (para Postgres, esto sigue manteniendo correctamente la relación ya que todo cliente_pedido es fácilmente accesible mediante JOINS desde su c_factura_linea, y viceversa, toda factura que contenga el pedido es fácilmente accesible a través de JOINS).

- Aplicaremos el mismo procedimiento para la terna vendedor_pedido, vendedor_factura, y v_factura_linea.
- Sabemos que varias backorder pueden conformar un vendedor_pedido, y que estas se conforman solo con las líneas de cliente_pedido que no se pudieron servir (es decir, no hay conceptos adicionales como en las facturas), por lo que para evitar duplicar información en la base de datos relacional la referencia debe mantener a nivel de vendedor_ped_li y la cliente_ped_li que pertenece a dicha backorder.
- Las facturas, tanto de vendedor como de cliente, deben mantener la referencia a éstos, ya que las líneas de tipo concepto no dependen de un pedido (que si está relacionado con cliente y vendedor) y podría darse la anomalía de una factura solo con conceptos que no perteneciera a ningún cliente, por lo que se conserva el atributo que ya existía pero se añade la relación en formato clave foránea.

Las entidades actualizadas en este punto quedarían :

```
CREATE TABLE IF NOT EXISTS orden.cli_backorder(
      backorder_clave       INTEGER,
      fecha                 DATE          NOT NULL,
      estado                VARCHAR(1)    NOT NULL,
      fecha_estado          DATE          NOT NULL,
      CONSTRAINT cli_backorder_pk   PRIMARY KEY (backorder_clave)
);

CREATE TABLE IF NOT EXISTS orden.cliente_pedido(
      id_pedido             INTEGER,
      cif                   VARCHAR(9),
      fecha                 DATE,
      estado                VARCHAR(1),
      fecha_estado1         DATE          NOT NULL,
      fecha_estado2         TIME          NOT NULL,
      prioridad             BOOLEAN       NOT NULL,
      total_neto            NUMERIC(8,2)  NOT NULL,
      total_bruto           NUMERIC(8,2)  NOT NULL,
      CONSTRAINT cliente_pedido_pk  PRIMARY KEY (id_pedido),
      CONSTRAINT cli_pedido_fk      FOREIGN KEY (cif) REFERENCES
                 asociado.cliente (cif),
      CONSTRAINT id_pedido_chk CHECK (id_pedido > 0 AND id_pedido < 999999999),
      CONSTRAINT total_neto_chk     CHECK (total_neto  >= 0),
      CONSTRAINT total_bruto_chk    CHECK (total_bruto >= 0)
);

CREATE TABLE IF NOT EXISTS orden.cliente_ped_li(
      id_pedido             INTEGER,
      num_linea             SMALLINT,
      id_pieza              INTEGER       NOT NULL,
      cantidad              SMALLINT,
      coste_neto_unidad     NUMERIC(8,2)  NOT NULL,
      coste_bruto_unidad    NUMERIC(8,2)  NOT NULL,
      total_neto            NUMERIC(8,2)  NOT NULL,
      total_bruto           NUMERIC(8,2)  NOT NULL,
      backorder             INTEGER,
      CONSTRAINT cliente_ped_li_pk PRIMARY KEY (id_pedido, num_linea),
      CONSTRAINT cli_ped_linea_fk FOREIGN KEY (id_pedido) REFERENCES
                orden.cliente_pedido(id_pedido),
      CONSTRAINT cli_back_linea_fk FOREIGN KEY (backorder) REFERENCES
                orden.cli_backorder(backorder_clave),
      CONSTRAINT cli_lin_pieza_fk FOREIGN KEY (id_pieza) REFERENCES
                almacen.pieza_local(id_pieza_gen),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT cantidad CHECK (cantidad > 0 AND cantidad < 999)
);
```

```
CREATE TABLE IF NOT EXISTS factura.cliente_factura(
      id_factura            INTEGER,
      id_cliente            VARCHAR(9)     NOT NULL,
      fecha_factura         DATE           NOT NULL,
      estado                VARCHAR(1)     NOT NULL,
      fecha_estado          DATE           NOT NULL,
      total_neto            NUMERIC(8,2),
      total_bruto           NUMERIC(8,2),
      CONSTRAINT cliente_factura_pk PRIMARY KEY (id_factura),
      CONSTRAINT id_cliente_fk      FOREIGN KEY (id_cliente) REFERENCES
                asociado.cliente(cif),
      CONSTRAINT id_factura_chk CHECK
                (id_factura > 0 AND id_factura < 999999999)
);

CREATE TABLE IF NOT EXISTS factura.c_factura_linea (
      id_factura            INTEGER,
      num_linea             SMALLINT,
      id_pedido_orig        INTEGER,
      id_concepto           VARCHAR(30),
      fecha                 DATE           NOT NULL,
      estado                VARCHAR(1)     NOT NULL,
      fecha_estado          DATE           NOT NULL,
      total_neto            NUMERIC(8,2)   NOT NULL,
      total_bruto           NUMERIC(8,2)   NOT NULL,
      CONSTRAINT c_factura_linea_pk PRIMARY KEY (id_factura, num_linea),
      CONSTRAINT cli_fac_linea_fk   FOREIGN KEY (id_factura) REFERENCES
                factura.cliente_factura(id_factura),
      CONSTRAINT cli_fac_ped_fk FOREIGN KEY(id_pedido_orig) REFERENCES
                orden.cliente_pedido (id_pedido),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT pedido_concepto_chk  CHECK (
             (id_pedido_orig IS NULL and id_concepto IS NOT NULL)       OR
             (id_pedido_orig IS NOT NULL and id_concepto IS NULL))
);

CREATE TABLE IF NOT EXISTS orden.vendedor_pedido(
      id_pedido             INTEGER,
      cif                   VARCHAR(9),
      fecha                 DATE,
      estado                VARCHAR(1),
      fecha_estado1         DATE           NOT NULL,
      fecha_estado2         TIME           NOT NULL,
      total_neto            NUMERIC(8,2)   NOT NULL,
      total_bruto           NUMERIC(8,2)   NOT NULL,
      CONSTRAINT vendedor_pedido_pk         PRIMARY KEY (id_pedido),
      CONSTRAINT vend_pedido_fk             FOREIGN KEY (cif) REFERENCES
                asociado.vendedor (cif),
      CONSTRAINT id_pedido_chk CHECK (id_pedido > 0 AND id_pedido < 999999999),
      CONSTRAINT total_neto_chk     CHECK (total_neto  >= 0),
      CONSTRAINT total_bruto_chk    CHECK (total_bruto >= 0)
);

CREATE TABLE IF NOT EXISTS orden.vendedor_ped_li(
      id_pedido             INTEGER,
      num_linea             SMALLINT,
      id_pieza              VARCHAR(20) NOT NULL,
      id_vendedor           VARCHAR(9) NOT NULL,
      cantidad              SMALLINT,
      total_neto            NUMERIC(8,2) NOT NULL,
      total_bruto           NUMERIC(8,2) NOT NULL,
      backorder      INTEGER,
      CONSTRAINT vendedor_ped_li_pk         PRIMARY KEY (id_pedido, num_linea),
      CONSTRAINT vend_ped_linea_fk  FOREIGN KEY (id_pedido) REFERENCES
                orden.vendedor_pedido(id_pedido),
      CONSTRAINT vend_lin_pieza_fk  FOREIGN KEY (id_pieza, id_vendedor)
                REFERENCES  almacen.pieza_maestro(id_pieza, id_vendedor),
      CONSTRAINT vend_back_linea_fk FOREIGN KEY (backorder) REFERENCES
                orden.cli_backorder(backorder_clave),
      CONSTRAINT num_linea_chk      CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT cantidad_chk       CHECK (cantidad  > 0 AND cantidad  < 999)
);

CREATE TABLE IF NOT EXISTS factura.vendedor_factura(
      id_factura            INTEGER,
      id_vendedor           VARCHAR(9)     NOT NULL,
```

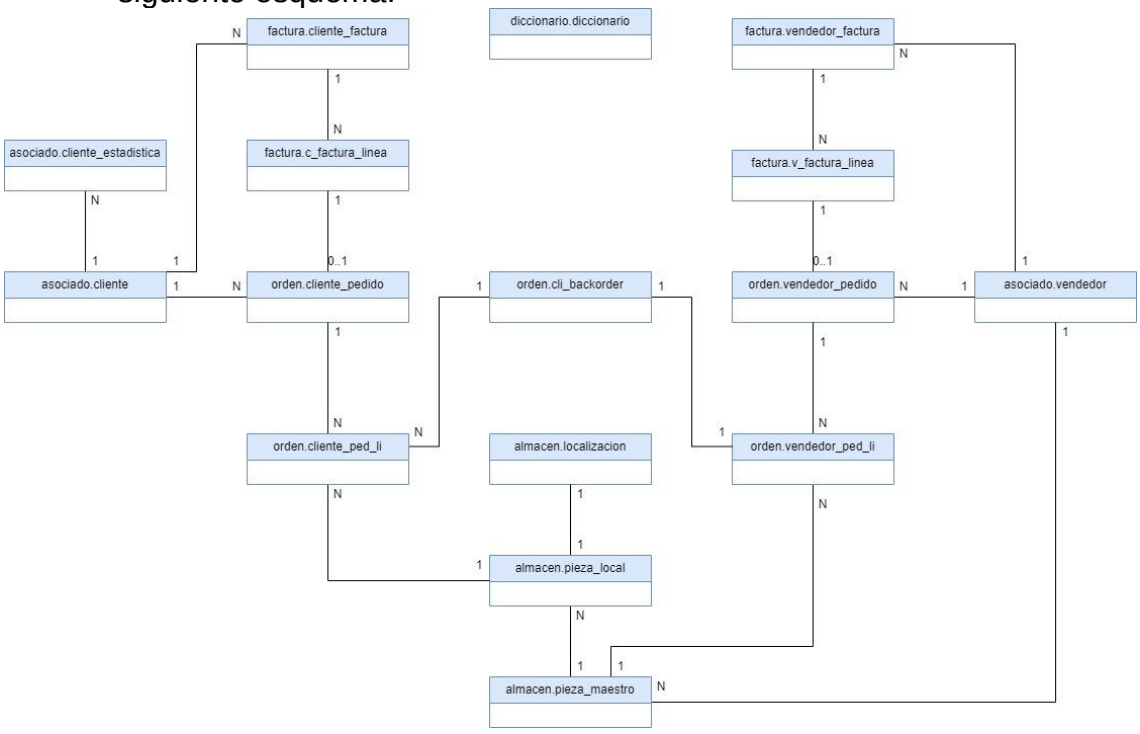
```
        fecha_factura          DATE           NOT NULL,
        estado                 VARCHAR(1)     NOT NULL,
        fecha_estado           DATE           NOT NULL,
        total_neto             NUMERIC (8,2)  NOT NULL,
        total_bruto            NUMERIC (8,2)  NOT NULL,
        total_neto_conf        NUMERIC (8,2)  NOT NULL,
        total_bruto_conf       NUMERIC (8,2)  NOT NULL,
        CONSTRAINT vendedor_factura_pk       PRIMARY KEY (id_factura),
        CONSTRAINT id_vendedor_fk     FOREIGN KEY (id_vendedor) REFERENCES
                   asociado.vendedor(cif),
        CONSTRAINT id_factura_chk CHECK
                   (id_factura > 0 AND id_factura < 999999999)
);

CREATE TABLE IF NOT EXISTS factura.v_factura_linea(
        id_factura             INTEGER,
        num_linea              SMALLINT,
        id_pedido_orig         INTEGER,
        id_concepto            VARCHAR(30),
        fecha                  DATE           NOT NULL,
        estado                 VARCHAR(1)     NOT NULL,
        fecha_estado           DATE           NOT NULL,
        total_neto             NUMERIC(8,2)   NOT NULL,
        total_bruto            NUMERIC(8,2)   NOT NULL,
        CONSTRAINT v_factura_linea_pk PRIMARY KEY (id_factura, num_linea),
        CONSTRAINT vend_fac_linea_fk  FOREIGN KEY (id_factura) REFERENCES
                   factura.vendedor_factura(id_factura),
        CONSTRAINT vend_fac_ped_fk FOREIGN KEY(id_pedido_orig) REFERENCES
                   orden.vendedor_pedido (id_pedido),
        CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999),
        CONSTRAINT pedido_concepto_chk  CHECK (
               (id_pedido_orig IS NULL AND id_concepto IS NOT NULL)       OR
               (id_pedido_orig IS NOT NULL AND id_concepto IS NULL))

);
```

5.2.2 Revisión del esquema ER.

Aplicando las correcciones del apartado anterior obtendríamos el siguiente esquema:



7. Esquema ER 2º iteración

5.2.3 Normalización y redundancia de datos.

Revisaremos a continuación todas las nuevas tablas y aplicaremos los cambios necesarios.

asociado.cliente: al haber descompuesto las direcciones (nivel 02 IDMS) en sus atributos componentes (nivel 03 IDMS) cumplimos con los requisitos de atomicidad, pero tenemos dos grupos de atributos que puede considerarse que pertenecen al mismo dominio: dirección, por lo que lo sacaremos como una entidad propia de dirección cliente, que además sólo puede ser de tipo 'envío' o 'factura' (al menos para cumplir con la funcionalidad mínima, en un futuro esto podría cambiar). Respecto al resto de los elementos de la tabla, dado que la clave no es compuesta, que todos los campos son determinados por ésta, y que no hay dependencias entre campos que no sean parte de la clave, esta tabla está en FNBC. Además, al no haber dependencia multivalor también estará en 4FN Y 5FN.

```
CREATE TABLE IF NOT EXISTS asociado.cliente (
      cif            VARCHAR(9),
      nombre         VARCHAR(30),
      fecha_alta     DATE,
      fecha_baja     DATE,
      estado         VARCHAR(1),
      CONSTRAINT cliente_pk PRIMARY KEY(cif)
);

CREATE TABLE IF NOT EXISTS asociado.direccion_cliente (
      cif                   VARCHAR(9),
      tipo_direccion        VARCHAR(10),
      calle                 VARCHAR(30),
      numero                SMALLINT,
      complemento           VARCHAR(50),
      direccion_cpostal     SMALLINT,
      CONSTRAINT direccion_cliente_pk      PRIMARY KEY(cif, tipo_direccion),
      CONSTRAINT cif_fk      FOREIGN KEY (cif) REFERENCES
                  asociado.cliente(cif),
      CONSTRAINT tipo_direccion_chk CHECK (tipo_direccion = 'envio' or
                tipo_direccion = 'factura')
);
```

asociado.cliente_estadistica: todos los atributos son atómicos y contiene una clave primaria única cif-anyo-mes que los determina, pero puede observarse que pedidos_acumulados (durante el año) y cuantía acumulada dependen solo de una parte de la clave que es cif+año, por lo que para cumplir con 2FN deberemos generar una nueva tabla. No se observan dependencias transitivas, ni multivaluadas, por lo que el resultado sería:

```
CREATE TABLE IF NOT EXISTS asociado.cliente_estadistica_anual(
      cif                   VARCHAR(9),
      anyo                  SMALLINT,
      pedidos_acumulado     INTEGER,
      cuantia_acumulada     NUMERIC(12,2) NOT NULL,
      CONSTRAINT cliente_estadistica_anual_pk PRIMARY KEY(cif,anyo),
      CONSTRAINT cli_esta_anual_fk FOREIGN KEY (cif)
                REFERENCES asociado.cliente (cif),
      CONSTRAINT cuantia_acumulada_anual_chk CHECK (cuantia_acumulada>=0)
 );
```

Nótese que en las reglas para cumplir la FN no se hace mención a las referencias en las tablas resultantes de forma específica, pero

observamos que los datos mensuales tienen sentido si están ligados a los anuales, por lo que se comportaría como una entidad débil y obtendríamos:

```
CREATE TABLE IF NOT EXISTS asociado.cliente_estadistica_mensual(
      cif                   VARCHAR(9),
      anyo                  SMALLINT,
      mes                   SMALLINT,
      pedidos_realizado     SMALLINT,
      cuantia               NUMERIC(8,2) NOT NULL,
      fecha_ultimo_pedido   DATE,
      CONSTRAINT cliente_estadistica_mensual_pk PRIMARY KEY(cif,anyo,mes),
      CONSTRAINT cli_esta_mensual_fk FOREIGN KEY (cif, anyo) REFERENCES
             asociado.cliente_estadistica_anual (cif, anyo),
      CONSTRAINT cuantia_chk CHECK (cuantia>=0)
);
```

Para asociado.vendedor todos los atributos son atómicos y dependen solo de la clave. Observamos que, como en el caso de los clientes, varios atributos conforman en realidad el dominio dirección, por lo que aplicaremos la misma lógica obteniendo una nueva tabla de dirección_vendedor, si bien solo puede ser de tipo factura.

```
CREATE TABLE IF NOT EXISTS asociado.vendedor(
      cif           VARCHAR(9),
      nombre        VARCHAR(30),
      fecha_alta    DATE          NOT NULL,
      fecha_baja    DATE,
      estado        BOOLEAN       NOT NULL,
      CONSTRAINT vendedor_pk PRIMARY KEY(cif)
);

CREATE TABLE IF NOT EXISTS asociado.direccion_vendedor (
      cif                   VARCHAR(9),
      tipo_direccion        VARCHAR(10),
      calle                 VARCHAR(30),
      numero                SMALLINT,
      complemento           VARCHAR(50),
      direccion_cpostal     SMALLINT,
      CONSTRAINT direccion_vendedor_pk PRIMARY KEY(cif, tipo_direccion),
      CONSTRAINT cif_fk     FOREIGN KEY (cif) REFERENCES
                  asociado.vendedor(cif),
      CONSTRAINT tipo_direccion_chk CHECK (tipo_direccion = 'factura')
);
```

Se nos plantea tras esta transformación una duda muy interesante: sería posible unificar las tablas de cliente y vendedor añadiendo un atributo “tipo_de_asociado” para diferenciar unos de otros, y entonces agrupar las direcciones. Realmente esta podría ser una solución, pero se ha descartado en base a:

- Esta transformación obligaría a cambiar toda la base de datos, pudiendo tener efectos inesperados en las tablas de pedidos de vendedor y de cliente, o sus facturas.
- Se han definido unas reglas iniciales, y queremos seguirlas en la medida de lo posible.
- Aunque parecido en este caso de estudio, un cliente y un proveedor pueden evolucionar y tener muchos datos diferentes, por lo que unificarlos sería contraproducente para la evolución funcional de la base de datos en un entorno más próximo a la realidad que el presente trabajo de fin de grado.

En almacen.pieza_maestro vemos que todos los atributos ya son atómicos, todos dependen de la combinación de id_pieza y id_vendedor (ya que podría darse el caso que diferentes proveedores coincidieran de forma casual al numerar sus piezas) y no hay dependencias transitivas, por lo que ya estaría normalizada.

La tabla almacen.pieza_local es un caso especial. Como hemos visto en la iteración inicial, la clave podía estar repetida y fue necesario introducir una clave primaria artificial que fuerza toda dependencia a ella, no por significado, sino por decisión de implementación. Podríamos contemplar usar fecha de vigencia inicial como parte de la clave (con id de pieza y de vendedor), pero con los datos de que disponemos y sabiendo que pueden solaparse, es posible que en algún caso 2 versiones tengan la misma fecha, si bien improbable. También se podría haber usado un nuevo atributo versión, pero se ha decidido mantener la propuesta inicial pues son las reglas que hemos establecido (de forma parecida a como comentábamos al inicio, en ocasiones es necesaria una desnormalización). Sin tener esto en cuenta, el resto de los atributos harían que la tabla sí estuviera normalizada.

Almacen.localizacion cumple que todos los atributos son atómicos, su único atributo depende de toda la clave que es la que identifica el lugar físico en el almacén por lo que cumple 2FN y por extensión al tener solo un atributo, también 3FN y Boyce-Codd.

En orden.cli_backorder, tras los cambios del apartado 5.2.1, nos han quedado todos los campos determinados solo por la clave backorder_clave y sin claves externas, pero el atributo fecha_estado que es no clave depende funcionalmente de estado que también es no clave (y también de backorder, ya que “estado” conceptualmente es el “estado de esta backorder”), por lo que para cumplir la tercera forma normal y siguientes extraeremos el atributo a una nueva entidad orden.cli_backorder_estado, lo cual será la forma de proceder recurrente en las tablas con fecha_estado. En el caso presente obtendríamos:

```
CREATE TABLE IF NOT EXISTS orden.cli_backorder(
      backorder_clave       INTEGER,
      estado                VARCHAR(1)     NOT NULL,
      fecha                 DATE           NOT NULL,
      CONSTRAINT cli_backorder_pk   PRIMARY KEY (backorder_clave)
);

CREATE TABLE IF NOT EXISTS orden.cli_backorder_estado     (
      backorder_clave              INTEGER,
      estado                VARCHAR(1)     NOT NULL,
      fecha_estado          DATE           NOT NULL,
      CONSTRAINT cli_backorder_estado_pk PRIMARY KEY (backorder_clave,estado),
      CONSTRAINT backorder_clave_fk FOREIGN KEY (backorder_clave) REFERENCES
                orden.cli_backorder(backorder_clave)
);
```

En orden.cliente_pedido vemos que cumple que todos los atributos son atómicos y determinados por la clave, pero existe la dependencia entre

estado y fecha_estado, y además los campos total_neto y total_bruto son calculados en función de la tabla de líneas y no necesarios para evitar duplicidad de datos. Por lo que para poder confirmar hasta FNBC, aplicamos:

```
CREATE TABLE IF NOT EXISTS orden.cliente_pedido(
      id_pedido      INTEGER,
      cif            VARCHAR(9),
      fecha          DATE,
      estado         VARCHAR(1),
      prioridad      BOOLEAN        NOT NULL,
      CONSTRAINT cliente_pedido_pk  PRIMARY KEY (id_pedido),
      CONSTRAINT cli_pedido_fk             FOREIGN KEY (cif) REFERENCES
asociado.cliente (cif),
      CONSTRAINT id_pedido_chk CHECK (id_pedido > 0 AND id_pedido < 999999999)
);

CREATE TABLE IF NOT EXISTS orden.cliente_pedido_estado(
      id_pedido      INTEGER,
      estado         VARCHAR(1),
      fecha_estado1  DATE           NOT NULL,
      fecha_estado2  TIME           NOT NULL,
      CONSTRAINT cliente_pedido_estado_pk  PRIMARY KEY (id_pedido, estado),
      CONSTRAINT id_pedido_fk              FOREIGN KEY (id_pedido) REFERENCES
                orden.cliente_pedido (id_pedido)
);
```

Para sus líneas en orden.cliente_ped_li, todos los atributos son atómicos y dependen de la clave primaria que es la línea a la que pertenecen, pero vemos el caso de un unos atributos que son calculados y se trata total_neto que se obtiene a partir de la cantidad y coste_neto_unidad, e igualmente total_bruto que se obtiene de coste_bruto_unidad y unidad. Como forma de eliminar esta dependencia de campos no clave, eliminaremos de la base de datos ambos valores (no son referenciados, y la query que recupere los datos o mediante una vista los puede retornar a partir de su atributos componentes).

```
CREATE TABLE IF NOT EXISTS orden.cliente_ped_li(
      id_pedido             INTEGER,
      num_linea             SMALLINT,
      id_pieza              INTEGER        NOT NULL,
      cantidad              SMALLINT,
      coste_neto_unidad     NUMERIC(8,2)   NOT NULL,
      coste_bruto_unidad    NUMERIC(8,2)   NOT NULL,
      backorder             INTEGER,
      CONSTRAINT cliente_ped_li_pk PRIMARY KEY (id_pedido, num_linea),
      CONSTRAINT cli_ped_linea_fk  FOREIGN KEY (id_pedido) REFERENCES
                orden.cliente_pedido(id_pedido),
      CONSTRAINT cli_back_linea_fk FOREIGN KEY (backorder) REFERENCES
                orden.cli_backorder(backorder_clave),
      CONSTRAINT cli_lin_pieza_fk FOREIGN KEY (id_pieza) REFERENCES
                almacen.pieza_local(id_pieza_gen),
      CONSTRAINT num_linea_chk      CHECK (num_linea > 0 AND num_linea < 999),
      CONSTRAINT cantidad           CHECK (cantidad  > 0 AND cantidad  < 999)
);
```

Factura.cliente_factura vemos que todos sus atributos son atómicos y dependen de toda la clave que es el identificador de factura, pero los campos tota_neto y total_bruto son calculables a partir de las líneas (información duplicada) y el atributo fecha_estado depende funcionalmente de estado, por lo que para poder cumplir hasta FNBC

sacaremos el atributo a una nueva entidad factura.cliente_factura_estado, obteniendo así:

```
CREATE TABLE IF NOT EXISTS factura.cliente_factura(
      id_factura            INTEGER,
      id_cliente            VARCHAR(9)     NOT NULL,
      fecha_factura         DATE           NOT NULL,
      estado                VARCHAR(1)     NOT NULL,
      CONSTRAINT cliente_factura_pk        PRIMARY KEY (id_factura),
      CONSTRAINT id_factura_chk
                 CHECK (id_factura > 0 AND id_factura < 999999999)
);

CREATE TABLE IF NOT EXISTS factura.cliente_factura_estado(
      id_factura            INTEGER,
      estado                VARCHAR(1),
      fecha_estado          DATE           NOT NULL,
      CONSTRAINT cliente_factura_estado_pk PRIMARY KEY (id_factura, estado),
      CONSTRAINT id_factura_fk FOREIGN KEY (id_factura)
               REFERENCES factura.cliente_factura(id_factura)
);
```

Teníamos en nuestra primera iteración una vista c_ind_fech_fac_view definida sobre factura.cliente_factura. Para adaptarla a los cambios anteriores, deberemos modificar su definición:

```
CREATE VIEW factura.c_ind_fech_fac_view AS
 SELECT fact.id_factura, fact.id_cliente, fact.fecha_factura, estado.estado,
         estado.fecha_estado, SUM(linea.total_neto), SUM(linea.total_bruto)
 FROM factura.cliente_factura fact,
      factura.cliente_factura_estado estado,
      factura.c_factura_linea linea
 WHERE  fact.id_factura = estado.id_factura
   AND  fact.estado = estado.estado
   AND  fact.id_factura = linea.id_factura
 GROUP BY  1,2,3,4,5,6,7
 ORDER BY fecha_factura desc, id_cliente desc
```

factura.c_factura_linea es una entidad compleja. Por una parte, como hemos visto anteriormente, contiene un atributo no clave fecha_estado que depende del atributo no clave estado, por lo que como en el caso anterior para cumplir la 3FN y FNBC crearemos la tabla factura.c_factura_linea_estado con una clave foránea a la tabla origen. Y por otro lado vemos que 2 de sus atributos, id_pedido_orig y concepto, son excluyentes por lo que siempre habrá una columna con valor nulo, que si bien no incumple directamente con los puntos de control de las formas normales, si va claramente contra su espíritu. Esto es así porque la tabla de líneas de factura es una generalización de las líneas que tienen su origen en un pedido y los cargos extra que se introducen manualmente, por lo que para cumplir esto y evitar los nulos generaremos 2 tablas c_factura_linea_pedido y c_factura_linea_concepto. Pero haciendo esto, necesitamos mantener aún la exclusividad entre ambos: para ello usaremos un trigger BEFORE INSERT que compruebe que ese número de línea no existe previamente en ninguna tabla.

```
CREATE TABLE IF NOT EXISTS factura.c_factura_linea (
      id_factura     INTEGER,
      num_linea      SMALLINT,
      fecha          DATE           NOT NULL,
      estado         VARCHAR(1),
      total_neto     NUMERIC(8,2)   NOT NULL,
```

```
      total_bruto    NUMERIC(8,2)   NOT NULL,
      CONSTRAINT c_factura_linea_pk PRIMARY KEY (id_factura, num_linea),
      CONSTRAINT cli_fac_linea_fk   FOREIGN KEY (id_factura) REFERENCES
                factura.cliente_factura(id_factura),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999)
);

CREATE TABLE IF NOT EXISTS factura.c_factura_linea_estado (
      id_factura    INTEGER,
      num_linea      SMALLINT,
      estado         VARCHAR(1),
      fecha_estado   DATE           NOT NULL,
      CONSTRAINT c_factura_linea_estado_pk
                PRIMARY KEY (id_factura, num_linea, estado),
      CONSTRAINT id_factura_num_linea_fk FOREIGN KEY (id_factura,num_linea)
                REFERENCES factura.c_factura_linea (id_factura, num_linea)
);

CREATE TABLE IF NOT EXISTS factura.c_factura_linea_pedido (
      id_factura            INTEGER,
      num_linea             SMALLINT,
      id_pedido_orig        INTEGER,
      CONSTRAINT c_factura_linea_pedido_pk
                PRIMARY KEY (id_factura, num_linea, id_pedido_orig),
      CONSTRAINT cli_fac_lin_ped_fk FOREIGN KEY(id_factura, num_linea)
                REFERENCES factura.c_factura_linea (id_factura, num_linea),
      CONSTRAINT cli_fac_ped_fk FOREIGN KEY(id_pedido_orig)
                REFERENCES orden.cliente_pedido (id_pedido)
);

CREATE TABLE IF NOT EXISTS factura.c_factura_linea_concepto (
      id_factura     INTEGER,
      num_linea      SMALLINT,
      concepto       VARCHAR(30),
      CONSTRAINT c_factura_linea_concepto_pk
                PRIMARY KEY (id_factura, num_linea, concepto),
      CONSTRAINT cli_fac_lin_con_fk FOREIGN KEY(id_factura, num_linea)
                REFERENCES factura.c_factura_linea (id_factura, num_linea)
);

CREATE FUNCTION factura.factura_linea_chk()  RETURNS trigger AS $func$
BEGIN
   PERFORM *
      FROM  factura.c_factura_linea_pedido
      WHERE id_factura = NEW.id_factura
        AND num_linea  = NEW.num_linea;
   IF FOUND THEN
      RAISE EXCEPTION 'YA EXISTE LINEA DE FACTURA DE TIPO PEDIDO';
   END IF;

   PERFORM *
      FROM  factura.c_factura_linea_concepto
      WHERE id_factura = NEW.id_factura
        AND num_linea  = NEW.num_linea;
   IF FOUND THEN
      RAISE EXCEPTION 'YA EXISTE LINEA DE FACTURA DE TIPO CONCEPTO';
   END IF;
   RETURN NEW;
END;
$func$  LANGUAGE plpgsql;

CREATE TRIGGER factura_linea_pedido_chk BEFORE INSERT ON
  factura.c_factura_linea_pedido FOR EACH ROW EXECUTE PROCEDURE
  factura.factura_linea_chk();

CREATE TRIGGER factura_linea_concepto_chk BEFORE INSERT ON
  factura.c_factura_linea_concepto FOR EACH ROW EXECUTE PROCEDURE
  factura.factura_linea_chk();
```

En el caso de orden.vendedor_pedido, vemos que todos sus atributos son atómicos y determinados por la clave, siendo la única dependencia entre atributos no clave la de estado y fecha_estado, y además siendo total_neto y total_bruto campos calculados en función de los valores de

las líneas, por lo que para no tener redundancia de datos y cumplir hasta FNBC extraemos en una nueva tabla el estado y elinamos los totales, obteniendo:

```
CREATE TABLE IF NOT EXISTS orden.vendedor_pedido(
      id_pedido             INTEGER,
      cif                   VARCHAR(9),
      fecha                 DATE,
      estado                VARCHAR(1),
      CONSTRAINT vendedor_pedido_pk        PRIMARY KEY (id_pedido),
      CONSTRAINT vend_pedido_fk            FOREIGN KEY (cif) REFERENCES
                asociado.vendedor (cif),
      CONSTRAINT id_pedido_chk CHECK (id_pedido > 0 AND id_pedido < 999999999)
);

CREATE TABLE IF NOT EXISTS orden.vendedor_pedido_estado(
      id_pedido             INTEGER,
      estado                VARCHAR(1),
      fecha_estado1         DATE           NOT NULL,
      fecha_estado2         TIME           NOT NULL,
      CONSTRAINT vendedor_pedido_estado_pk PRIMARY KEY (id_pedido, estado),
      CONSTRAINT id_pedido_fk              FOREIGN KEY (id_pedido) REFERENCES
                orden.vendedor_pedido (id_pedido));
```

Y por supuesto también las líneas de vendedor_ped_li que observamos que todos sus atributos son atómicos y dependen de la clave, de toda la clave, estando en FNBC (y de hecho, como en la mayoría de casos de nuestro estudio, también validaría hasta 5FN), por lo que no son necesarios cambios.

Para factura.vendedor_factura se puede ver fácilmente que la única dependencia interna necesaria para que esté en FNBC es la ya conocida entre estado y fecha_estado. Adicionalmente total_neto y total_bruto son campos calculados que pueden obtenerse de las líneas y que eliminaremos para evitar duplicidad de datos. Nótese que total_neto_conf y total_neto_bruto no son calculados por el sistema sino que vienen del proveedor de partes, y por lo tanto los conservaremos.

```
CREATE TABLE IF NOT EXISTS factura.vendedor_factura(
      id_factura            INTEGER,
      id_vendedor           VARCHAR(9)     NOT NULL,
      fecha_factura         DATE           NOT NULL,
      estado                VARCHAR(1)     NOT NULL,
      total_neto_conf       NUMERIC (8,2)  NOT NULL,
      total_bruto_conf      NUMERIC (8,2)  NOT NULL,
      CONSTRAINT vendedor_factura_pk       PRIMARY KEY (id_factura),
      CONSTRAINT id_vendedor_fk     FOREIGN KEY (id_vendedor)
                REFERENCES asociado.vendedor(cif),
      CONSTRAINT id_factura_chk CHECK
                (id_factura > 0 AND id_factura < 999999999)
);

CREATE TABLE IF NOT EXISTS factura.vendedor_factura_estado(
      id_factura     INTEGER,
      estado         VARCHAR(1)     NOT NULL,
      fecha_estado   DATE           NOT NULL,
      CONSTRAINT vendedor_factura_estado_pk
                PRIMARY KEY (id_factura, estado)
);
```

Y al hacer este cambio, también impactamos en la vista v_ind_fech_fac_view que teníamos definida sobre factura.vendedor_factura que ahora quedará como:

```
CREATE VIEW factura.v_ind_fech_fac_view AS
SELECT fact.id_factura,  fact.id_vendedor, fact.fecha_factura, fact.estado,
        SUM(linea.total_neto) AS "total_neto",  SUM(linea.total_bruto)  AS
        "total_bruto",  fact.total_neto_conf, fact.total_bruto_conf
FROM  factura.vendedor_factura fact,
      factura.vendedor_factura_estado est,
      factura.v_factura_linea linea
WHERE fact.id_factura = est.id_factura
  AND fact.estado     = est.estado
  AND fact.id_factura = linea.id_factura
GROUP BY  1,2,3,4,8
ORDER BY fecha_factura desc, id_vendedor desc;
```

Finalmente, nos queda las líneas de la factura que adicionalmente de extraer el campo estado y fecha_estado en una tabla externa vemos que como en el caso de las líneas de pedido de cliente, en los pedidos para los proveedores estas líneas de pedido pueden tener origen o bien en la petición de partes que realiza el mismo almacén para su stock o bien porque son necesarias actualmente para poder completar un pedido de cliente (a través de la backorder generada), o bien ser un carga extra, oferta aplicable a la factura, etc. Encararemos esta casuística como una especialización para evitar tener columnas exclusivas id_pedido_orig e id_concepto, obteniendo las tablas en FNBC:

```
CREATE TABLE IF NOT EXISTS factura.v_factura_linea(
      id_factura            INTEGER,
      num_linea             SMALLINT,
      fecha                 DATE          NOT NULL,
      estado                VARCHAR(1)    NOT NULL,
      total_neto            NUMERIC(8,2)  NOT NULL,
      total_bruto           NUMERIC(8,2)  NOT NULL,
      CONSTRAINT v_factura_linea_pk PRIMARY KEY (id_factura, num_linea),
      CONSTRAINT vend_fac_linea_fk  FOREIGN KEY (id_factura)
                REFERENCES factura.vendedor_factura(id_factura),
      CONSTRAINT num_linea_chk CHECK (num_linea > 0 AND num_linea < 999)
);

CREATE TABLE IF NOT EXISTS factura.v_factura_linea_estado(
      id_factura            INTEGER,
      num_linea             SMALLINT,
      estado                VARCHAR(1)    NOT NULL,
      fecha_estado          DATE          NOT NULL,
      CONSTRAINT v_factura_linea_estado_pk
                PRIMARY KEY (id_factura, num_linea, estado),
      CONSTRAINT id_factura_num_linea_fk FOREIGN KEY (id_factura,num_linea)
                REFERENCES factura.V_factura_linea (id_factura, num_linea)

);

CREATE TABLE IF NOT EXISTS factura.v_factura_linea_pedido(
      id_factura            INTEGER,
      num_linea             SMALLINT,
      id_pedido_orig        INTEGER,
      CONSTRAINT v_factura_linea_pedido_pk
                PRIMARY KEY (id_factura, num_linea, id_pedido_orig),
      CONSTRAINT vend_fac_lin_fk    FOREIGN KEY (id_factura, num_linea)
                REFERENCES factura.v_factura_linea(id_factura, num_linea),
      CONSTRAINT vend_fac_lin_ped_fk FOREIGN KEY(id_pedido_orig)
                REFERENCES orden.vendedor_pedido (id_pedido)
);

CREATE TABLE IF NOT EXISTS factura.v_factura_linea_concepto(
      id_factura            INTEGER,
      num_linea             SMALLINT,
      id_concepto           VARCHAR(30),
      CONSTRAINT v_factura_linea_concepto_pk
                PRIMARY KEY (id_factura, num_linea, id_concepto),
      CONSTRAINT vend_fac_lin_fk    FOREIGN KEY (id_factura, num_linea)
```

```
                REFERENCES factura.v_factura_linea(id_factura, num_linea)
);
CREATE FUNCTION factura.v_factura_linea_chk()  RETURNS trigger AS $func$
BEGIN
   PERFORM *
      FROM  factura.v_factura_linea_pedido
      WHERE id_factura = NEW.id_factura
        AND num_linea  = NEW.num_linea;
   IF FOUND THEN
      RAISE EXCEPTION 'YA EXISTE LINEA DE FACTURA DE TIPO PEDIDO ';
   END IF;

   PERFORM *
      FROM factura.v_factura_linea_concepto
      WHERE id_factura = NEW.id_factura
        AND num_linea  = NEW.num_linea;
   IF FOUND THEN
      RAISE EXCEPTION 'YA EXISTE LINEA DE FACTURA DE TIPO CONCEPTO';
   END IF;
   RETURN NEW;
END;
$func$  LANGUAGE plpgsql;

CREATE TRIGGER v_factura_linea_pedido_chk BEFORE INSERT ON
      factura.v_factura_linea_pedido FOR EACH ROW EXECUTE PROCEDURE
      factura.v_factura_linea_chk();

CREATE TRIGGER v_factura_linea_concepto_chk BEFORE INSERT ON
      factura.v_factura_linea_concepto FOR EACH ROW EXECUTE PROCEDURE
      factura.v_factura_linea_chk();
```

La última tabla, la tabla de diccionario, tras aplicar las normas definidas de migración cumple funcionalmente con su cometido tal como lo hacía en IDMS, pero vemos claramente que tenemos varios campos del mismo dominio de clave y varios campos del mismo dominio de atributo para esas claves y que los valores de estos dependen de un número variable de los componentes de la clave lógica (recordamos, definida a través de un índice para poder aceptar nulos). En este punto, no podemos aplicar ninguna transformación estructural adicional que nos permita normalizarla sin bajar hasta el nivel de datos, por lo que la veremos en detalle en la siguiente sección.

5.3 Iteración final.

En el apartado 5.1 hemos aplicado las reglas que habíamos definido para obtener una equivalencia sintáctica de IDMS en PostgreSQL, y en el apartado 5.2 hemos implementado las adaptaciones necesarias al modelo relacional normalizado. En esta última revisión razonaremos y analizaremos a partir de los datos y no la estructura la conveniencia de la tabla diccionario, propondremos los cambios que creamos necesarios respecto a esta y cualquier otra tabla, crearemos componentes nuevos que nos permitan aprovechar el potencial de PostgresSQL como son las vistas y funciones, y añadiremos roles para restringir la visibilidad y usabilidad de la base de datos.

5.3.1 Tabla de diccionario.

Inicialmente, habíamos definido la tabla de diccionario como una tabla de tablas (lógicas), que contenía los posibles valores de estado_linea,

estado_basico, estado_pedido_vendedor, estado_pedido_cliente o estado_backorder, los rangos usados para generar identificadores en rango_pedido_vendedor, rango_backorder, o el rango_pedido_cliente, también permite definir si se bloquea el sistema de facturación en bloqueo_factura o de forma parecida para las ordenes en bloqueo_orden, o contener los valores válidos en codigos_postales. Estos forman nuestro caso de estudio, pero en un entorno productivo podría haber un número indeterminado.

Para poder llevar esta tabla a FNBC se realizan se aplican las siguientes decisiones:

- Toda tabla lógica que consiste en un par clave-valor, puede seguir formando parte de la tabla diccionario, que ahora solo tendrá un campo que actúa de clave y un atributo.
- Cualquier otro caso, debe generar su propia tabla.

Si revisamos el listado del contenido de ejemplo de la sección 3, vemos que en nuestro diccionario tendremos las siguientes tablas lógicas que siguen la estructura de clave-valor: estado_linea, estado_basico, estado_pedido_vendedor, estado_pedido_cliente, estado_backorder, rango_pedido_vendedor, rango_backorder, y rango_pedido_cliente. Para éstas definimos las siguientes tablas:

```
CREATE TABLE diccionario.diccionario_tabla(
      tabla          VARCHAR(20),
      descripcion    VARCHAR(50),
      CONSTRAINT diccionario_pk PRIMARY KEY (tabla)
);

CREATE TABLE diccionario.diccionario_linea(
      tabla          VARCHAR(20),
      clave          VARCHAR(100),
      valor          VARCHAR(100),
      CONSTRAINT diccionario_linea_pk PRIMARY KEY (tabla, clave),
      CONSTRAINT tabla_fk FOREIGN KEY (tabla) REFERENCES
                diccionario.diccionario_tabla(tabla)
);
```

Del resto de tablas originales, la tabla bloqueo_factura al tener 2 atributos deberá tener su propia tabla, y de la misma forma bloqueo_orden que pueden ser agrupadas tal que:

```
CREATE TABLE diccionario.bloqueos_cliente (
      tabla          VARCHAR(20),
      CIF            VARCHAR(9) NOT NULL,
      fecha_inicio   DATE,
      fecha_fin      DATE,
      CONSTRAINT bloqueos_cliente_pk PRIMARY KEY (tabla, CIF),
      CONSTRAINT fecha_inicio_fin_chk CHECK (fecha_inicio > fecha_fin)
);
```

Con la particularidad de que CIF admite el valor * cuando afecte a todos el sistema, o un identificador cuando afecte a un cliente o vendedor (ya que los CIF son únicos).

Finalmente, hay 2 dominios de datos que claramente también pueden considerarse aptos para tener su propia tabla dedicada ya que son un conjunto de valores constantes contra los que hay que validar. Son los

códigos postales, y los países. Además, en el enunciado habíamos dicho que uno de los propósitos de la migración era favorecer la integración con otros CRM que facilitase la expansión de la empresa a otros mercados o países, por lo que añadiremos a las direcciones el país en formato ISO2, lo cual también nos implica la creación de una tabla de países. Como resultado:

```
CREATE TABLE diccionario.pais(
      iso2           VARCHAR(2),
      iso3           VARCHAR(3),
      descripcion    VARCHAR(100)   NOT NULL,
      CONSTRAINT pais_pk PRIMARY KEY (iso2)
);

CREATE TABLE diccionario.codigo_postal(
      iso2_pais      VARCHAR(2),
      cpostal        VARCHAR(10),
      CONSTRAINT codigo_postal_pk PRIMARY KEY (iso2_pais, cpostal),
      CONSTRAINT iso2_fk FOREIGN KEY (iso2_pais)
                REFERENCES diccionario.pais(iso2)
);
```

En consecuencia, en las tablas de direcciones añadiremos la validación de código postal, que pasa a ser alfanumérico en conformidad con los posibles valores a nivel europeo. La de país no es necesaria ya que nos vendrá también dada por la tabla de código postal al formar parte de su clave, y deber existir:

```
CREATE TABLE IF NOT EXISTS asociado.direccion_cliente (
      cif                  VARCHAR(9),
      tipo_direccion       VARCHAR(10),
      calle                VARCHAR(30),
      numero               SMALLINT,
      complemento          VARCHAR(50),
      direccion_cpostal    VARCHAR(10)    NOT NULL,
      pais                 VARCHAR(2)     NOT NULL,
      CONSTRAINT direccion_cliente_pk PRIMARY KEY(cif, tipo_direccion),
      CONSTRAINT cif_fk     FOREIGN KEY (cif) REFERENCES asociado.cliente(cif),
      CONSTRAINT direccion_cpostal_fk
            FOREIGN KEY (pais, direccion_cpostal) REFERENCES
            diccionario.codigo_postal(iso2_pais, cpostal),
      CONSTRAINT tipo_direccion_chk
           CHECK (tipo_direccion = 'envio' or tipo_direccion = 'factura')
);

CREATE TABLE If NOT EXISTS asociado.direccion_vendedor (
      cif                  VARCHAR(9),
      tipo_direccion       VARCHAR(10),
      calle                VARCHAR(30),
      numero               SMALLINT,
      complemento          VARCHAR(50),
      direccion_cpostal    VARCHAR(10)    NOT NULL,
      pais                 VARCHAR(2)     NOT NULL,
      CONSTRAINT direccion_vendedor_pk PRIMARY KEY(cif, tipo_direccion),
      CONSTRAINT cif_fk     FOREIGN KEY (cif)
                REFERENCES asociado.vendedor(cif),
      CONSTRAINT direccion_cpostal_fk
               FOREIGN KEY (pais, direccion_cpostal) REFERENCES
               diccionario.codigo_postal(iso2_pais, cpostal),
      CONSTRAINT tipo_direccion_chk CHECK (tipo_direccion = 'factura')
);
```

Respecto a los estados, estos tienen casuística muy variada entre las distintas entidades por lo que no hay una forma eficaz de unificarlos en una tabla y que sean referenciados, pero sí es evidente que la base de datos debe ofrecer robustez y no permitir que pueda grabarse un valor

no correcto (quizás por un mal uso del diccionario, inyección de código, etc…). Por lo tanto se ha optado por añadir los diferentes valores posibles de cada estado en forma de una restricción CHECK. Las entidades afectadas por tanto quedarían como:

```
CREATE TABLE IF NOT EXISTS asociado.cliente (
      cif                   VARCHAR(9),
      nombre                VARCHAR(30),
      fecha_alta            DATE,
      fecha_baja            DATE,
      estado                VARCHAR(1),
      CONSTRAINT cliente_pk PRIMARY KEY(cif),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2', '3', '4'))
);

CREATE TABLE IF NOT EXISTS asociado.vendedor(
      cif                   VARCHAR(9),
      nombre                VARCHAR(30),
      fecha_alta            DATE          NOT NULL,
      fecha_baja            DATE,
      estado                BOOLEAN       NOT NULL,
      CONSTRAINT vendedor_pk PRIMARY KEY(cif),
      CONSTRAINT estado_chk CHECK(estado in ('0', '1'))
);

CREATE TABLE IF NOT EXISTS orden.cli_backorder_estado(
      backorder_clave       INTEGER,
      estado                VARCHAR(1)    NOT NULL,
      fecha_estado          DATE          NOT NULL,
      CONSTRAINT cli_backorder_estado_pk PRIMARY KEY (backorder_clave,estado),
      CONSTRAINT backorder_clave_fk FOREIGN KEY (backorder_clave)
                REFERENCES orden.cli_backorder(backorder_clave),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2'))
);

CREATE TABLE IF NOT EXISTS orden.cliente_pedido_estado(
      id_pedido             INTEGER,
      estado                VARCHAR(1),
      fecha_estado1         DATE          NOT NULL,
      fecha_estado2         TIME          NOT NULL,
      CONSTRAINT cliente_pedido_estado_pk  PRIMARY KEY (id_pedido, estado),
      CONSTRAINT id_pedido_fk              FOREIGN KEY (id_pedido) REFERENCES
                orden.cliente_pedido (id_pedido),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2', '3'))
);

CREATE TABLE IF NOT EXISTS factura.cliente_factura_estado(
      id_factura            INTEGER,
      estado                VARCHAR(1),
      fecha_estado          DATE          NOT NULL,
      CONSTRAINT cliente_factura_estado_pk
                PRIMARY KEY (id_factura, estado),
      CONSTRAINT id_factura_fk FOREIGN KEY (id_factura)
                REFERENCES factura.cliente_factura(id_factura),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2', '3'))
);

CREATE TABLE IF NOT EXISTS factura.c_factura_linea_estado (
      id_factura            INTEGER,
      num_linea             SMALLINT,
      estado                VARCHAR(1),
      fecha_estado          DATE          NOT NULL,
      CONSTRAINT c_factura_linea_estado_pk
                PRIMARY KEY (id_factura, num_linea, estado),
      CONSTRAINT id_factura_num_linea_fk FOREIGN KEY (id_factura,num_linea)
                REFERENCES factura.c_factura_linea (id_factura, num_linea),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2'))
);

CREATE TABLE IF NOT EXISTS orden.vendedor_pedido_estado(
      id_pedido             INTEGER,
      estado                VARCHAR(1),
      fecha_estado1         DATE          NOT NULL,
```

```
      fecha_estado2         TIME          NOT NULL,
      CONSTRAINT vendedor_pedido_estado_pk PRIMARY KEY (id_pedido, estado),
      CONSTRAINT id_pedido_fk              FOREIGN KEY (id_pedido)
                REFERENCES orden.vendedor_pedido (id_pedido),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2'))
);

CREATE TABLE IF NOT EXISTS factura.vendedor_factura_estado(
      id_factura     INTEGER,
      estado         VARCHAR(1)     NOT NULL,
      fecha_estado   DATE           NOT NULL,
      CONSTRAINT vendedor_factura_estado_pk PRIMARY KEY (id_factura, estado),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2', '3', '4'))
);

CREATE TABLE IF NOT EXISTS factura.v_factura_linea_estado(
      id_factura            INTEGER,
      num_linea             SMALLINT,
      estado                VARCHAR(1)     NOT NULL,
      fecha_estado          DATE           NOT NULL,
      CONSTRAINT v_factura_linea_estado_pk
                PRIMARY KEY (id_factura, num_linea, estado),
      CONSTRAINT id_factura_num_linea_fk FOREIGN KEY (id_factura,num_linea)
                REFERENCES factura.V_factura_linea (id_factura, num_linea),
      CONSTRAINT estado_chk CHECK (estado in ('0', '1', '2'))
);
```

5.3.2 Vistas.

Las vistas nos permiten ocultar la información que no es necesaria en las consultas, simplifican la administración de permisos, simplifican el acceso a datos, enmascaran el acceso a la información de forma que los cambios en la base de datos no afecten las consultas, y no ocupan espacio, sino que son una forma lógica de presentar los datos.

Adicionalmente a la definición anterior de vista, tenemos en PostgreSQL las MATERIALIZED VIEWS, un tipo específico de vista que sí se almacena físicamente e incluso permite el uso de índices. Generalmente usadas en *data warehouses* y *business intelligence* para el acceso a grandes volúmenes de datos que no cambian frecuentemente. Una vez ejecutada, el contenido de la vista permanece estático hasta que se realiza una acción de refresco:

```
REFRESH MATERIALIZED VIEW view_name;
```

Todas las opciones para la definición de vistas en PostgresSQL 13 pueden consultarse en https://www.postgresql.org/docs/13/sql-createview.html y https://www.postgresql.org/docs/13/sql-creatematerializedview.html respectivamente.

Durante todo el proceso de migración hemos podido observar 2 tablas cuyos valores son totalmente calculados: cliente_estadistica_mensual y cliente_estadistica_anual. En este momento nos podemos plantear de representarlas como una vista materializada. Si asumimos que no contienen datos históricos (por ejemplo, tras migrar información de los pedidos en IDMS a archivos), que su uso es solo para la creación de informes puntuales y no un uso diario intensivo, y que los procesos solo necesitan actualizarlas a final de mes o bajo petición de usuario, cumplen perfectamente con los usos de las vistas materializadas por lo que podríamos obtenerlas de forma equivalente con:

```
CREATE MATERIALIZED VIEW IF NOT EXISTS asociado.v_cliente_estadistica_mensual
AS
SELECT cif, date_part('year', fecha) as "anyo", date_part('month', fecha) as
       "mes", sum (coste_bruto_unidad * cantidad) as "total_bruto" , max(fecha)
FROM  orden.cliente_pedido ped, orden.cliente_ped_li lin
WHERE ped.id_pedido = lin.id_pedido
GROUP BY 1,2,3;

CREATE MATERIALIZED VIEW IF NOT EXISTS asociado.v_cliente_estadistica_anual AS
SELECT cif, date_part('year', fecha) as "anyo", COUNT(*) AS "pedido", SUM
      (total_bruto) AS "total_bruto"
FROM  orden.cliente_pedido PED, orden.cliente_ped_li lin
WHERE ped.id_pedido = lin.id_pedido
GROUP BY 1,2;
```

Por otra parte, de acuerdo con nuestro caso de estudio no hay una serie de vistas predefinidas en la aplicación a migrar: las vistas no existen en IDMS clásico sino que aparecen cuando se añade a este compatibilidad con SQL y requieren previamente la definición de las tablas SQL que representan los RECORD[1].

Aún así, como parte de esta iteración final de mejora vamos a definir unas vista de ejemplo con lo que podrían ser algunas de las consultas más comunes:

1. Detalle de un pedido de cliente que muestre el identificador de pedido, el identificador de cliente, el estado, el número de factura al que pertenece en caso de esta facturado, y todas sus líneas incluyendo información de la parte, descripción de la parte, cantidad solicitada, y coste neto y bruto de la línea. Esta vista podría ser usada en una aplicación para recuperar la información completa de un pedido.

   ```
   CREATE VIEW orden.v_detalle_pedido AS
   SELECT pedido.id_pedido, pedido.cif, pedido.estado, factura.id_factura,
          linea.num_linea, pieza.id_pieza, pieza.descripcion, linea.cantidad,
          linea.coste_neto_unidad * linea.cantidad AS "total neto",
          linea.coste_bruto_unidad * linea.cantidadAS "total bruto"
   FROM orden.cliente_ped_li linea, almacen.pieza_local pieza,
        orden.cliente_pedido pedido LEFT JOIN
            factura.c_factura_linea_pedido factura ON
            pedido.id_pedido = factura.id_pedido_orig
   WHERE linea.id_pedido = pedido.id_pedido
   and   linea.id_pieza  = pieza.id_pieza_gen;
   ```

2. Selecciona aquellas líneas de pedido no canceladas que forman parte de una backorder, que todavía no han generado pedido al proveedor, y para las cuales sigue sin haber stock (por lo que deberán generar pedido a proveedor) mostrando el id del pedido original, el id de la backorder, el id de la pieza y su descripción, cantidad solicitada,y stock actual.

1.https://www.ibm.com/docs/en/icfsfz/11.3.0?topic=federation-creating-ca-idms-tables-views-classic

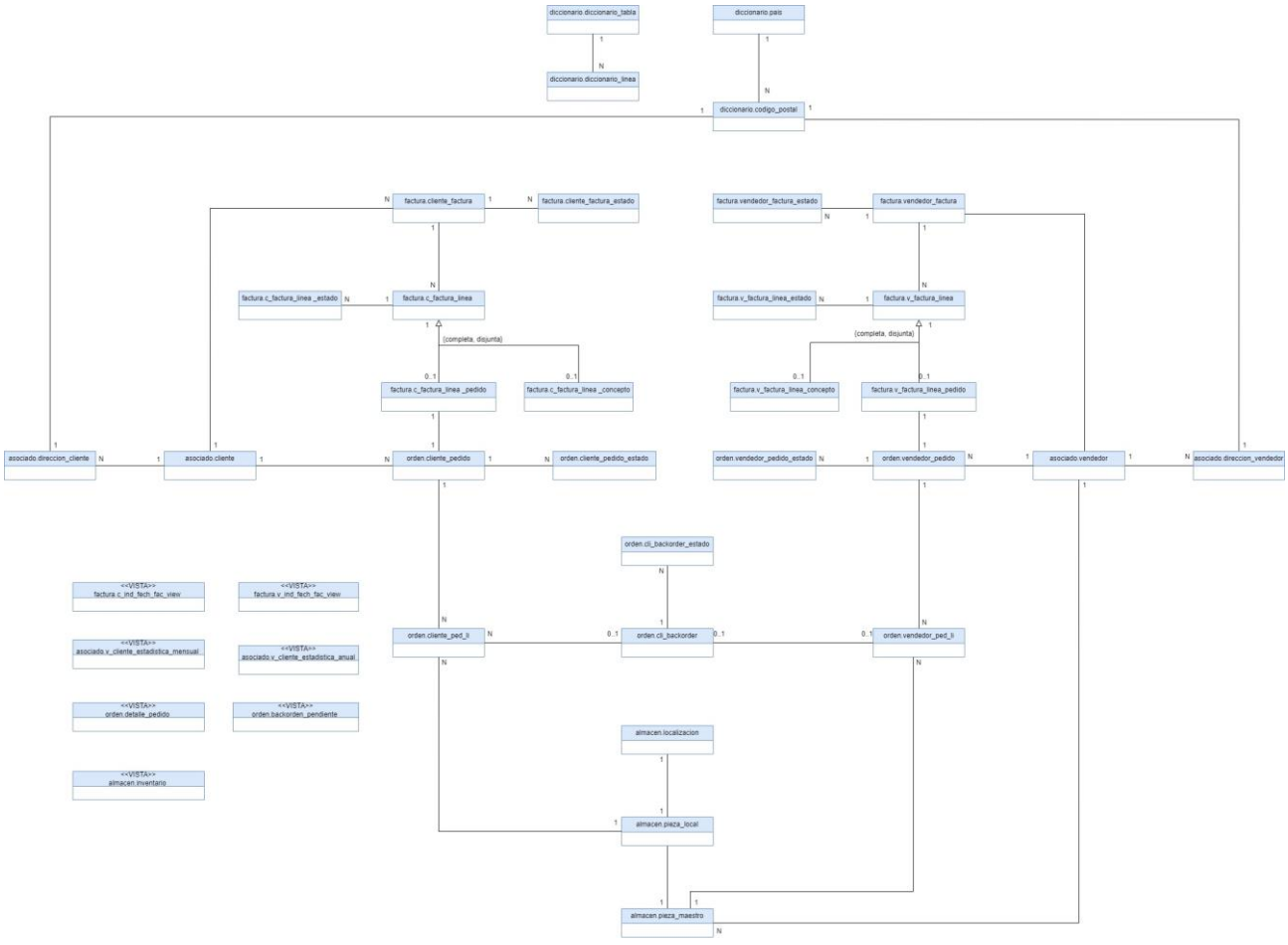
```
CREATE VIEW orden.backorden_pendiente AS
SELECT linea.id_pedido, linea.backorder, pieza.id_pieza, pieza.descripcion,
       linea.cantidad, pieza.stock
FROM orden.cliente_ped_li linea,  almacen.pieza_local pieza
WHERE linea.backorder IS NOT NULL
  AND NOT EXISTS (SELECT *
                  FROM orden.vendedor_ped_li vend_linea
                  WHERE linea.backorder = vend_linea.backorder)
  AND linea.id_pieza = pieza.id_pieza_gen
  AND linea.cantidad > pieza.stock;
```

3. Para poder identificar las localizaciones libres para guardar las piezas nuevas que entran, o para reorganizar el almacén, muestra todas las localizaciones activas disponibles con la parte contenida y su cantidad en caso de no estar vacía

```
CREATE VIEW almacen.inventario AS
SELECT id_area, id_seccion, id_estanteria, id_pieza, descripcion, stock
FROM  almacen.localizacion localizacion
      LEFT JOIN almacen.pieza_local pieza ON
      localizacion.id_parte = pieza.id_pieza_gen
WHERE  localizacion.estado is true;
```

5.3.3 Revisión del esquema ER.

Tras los cambios de las secciones anteriores, nuestro esquema ER quedaría tal que:



8. Esquema ER 3º iteración

5.3.4 Funciones.

Las funciones y los triggers permiten simplificar y automatizar los procesos, haciéndolos trasparentes para cualquier cliente. En apartados anteriores de nuestro caso de estudio habíamos indicado que a nivel de datos en el esquema de diccionario estarían los rangos para poder definir qué numeración se usa para los pedidos de cliente y proveedor, facturas de cliente y proveedor, y backorders.

Un ejemplo de estos datos en el diccionario sería:

```
INSERT INTO diccionario.diccionario_tabla
      VALUES ('rango_backorder', 'siguiente id para las backorder');
INSERT INTO diccionario.diccionario_linea
      VALUES ('rango_backorder', '*', '0');
INSERT INTO diccionario.diccionario_tabla
      VALUES ('rango_factura_cliente', 'siguiente id para factura cliente');
INSERT INTO diccionario.diccionario_linea
      VALUES ('rango_factura_cliente', '*', '0');
INSERT INTO diccionario.diccionario_tabla
      VALUES ('rango_factura_vendedor', 'siguiente id para factura vendedor');
INSERT INTO diccionario.diccionario_linea
      VALUES ('rango_factura_vendedor', '*', '0');
INSERT INTO diccionario.diccionario_tabla
      VALUES ('rango_orden_cliente', 'siguiente id para orden cliente');
INSERT INTO diccionario.diccionario_linea
      VALUES ('rango_orden_cliente', '*', '0');
INSERT INTO diccionario.diccionario_tabla
      VALUES ('rango_orden_vendedor', 'siguiente id para orden vendedor');
INSERT INTO diccionario.diccionario_linea
      VALUES ('rango_orden_vendedor', '*', '0');
```

Por lo tanto necesitamos poder mantener dicho valor, y para ello realizaremos una función que dado un rango, accederá a su línea marcándolo con FOR UPDATE lo cual nos aplicará un bloqueo sobre dicho registro únicamente y no toda la tabla, hasta que acabe la transacción, asegurando así la integridad.

```
CREATE  OR  REPLACE  FUNCTION  diccionario.id_generator  (rango_in  VARCHAR(30))
RETURNS INTEGER AS $$
 DECLARE
    valor_actual INTEGER DEFAULT 0;
    valor_nuevo  INTEGER DEFAULT 0;
 BEGIN
    SELECT valor INTO valor_actual
        FROM diccionario.diccionario_linea
        WHERE tabla = rango_in
        FOR UPDATE;

    IF NOT FOUND THEN
      RAISE EXCEPTION 'ERROR: NO EXISTE LA SECUENCIA % PARA GENERAR ID', rango;
              valor_nuevo := 0;
    ELSE
      valor_nuevo := valor_actual + 1;
              UPDATE  diccionario.diccionario_linea
                SET valor = valor_nuevo
              WHERE tabla = rango_in;
    END IF;
    RETURN valor_nuevo;
 END; $$LANGUAGE plpgsql;
```

5.3.5 Roles y usuarios.

PostgresSQL controla el acceso a los objetos y su información en la base de datos a través del concepto de rol e indicando qué acciones

están permitidas para cada uno sobre dichos objetos de la base de datos. Un rol puede corresponder con un usuario o un grupo de usuarios, por lo que es posible añadir roles a roles ya existentes, como miembros. Por lo tanto, los roles son un elemento clave para restringir el acceso a la información y dar robustez a nuestra base de datos.

Hay que tener en cuenta que desde la versión de Postgres 8.1, usuarios y grupos ya no son elementos diferenciados. Aunque se usa pgAdmin como interfaz con la base de datos, y este incluye un menú contextual para la definición de los roles, estos se definirán de forma manual mediante sentencias, tal como se ha venido haciendo en el presente trabajo de fin de grado.

La sentencia básica sería:

```
CREATE USER name [ [ WITH ] option [ ... ] ]
```

Inicialmente no disponemos de información de qué usuarios acceden a la base de datos IDMS actualmente. Habitualmente estos están definidos mediante la herramienta Resource Access Control Facility (abreviado RACF[1]). Asumiremos como buena práctica que se debe definir un usuario Admin técnico para realizar cualquier acción correctiva de datos, un usuario operacional que será al que se conecten las aplicaciones del sistema del almacén y que pueda hacer un limitado mantenimiento de datos, y finalmente un usuario de solo lectura que tendrá acceso tan solo a las vistas.

Nuestro usuario de tipo Admin (no confundir con el usuario root propio de PostgreSQL) no debería poder crear bases de datos nuevas, ni eliminar la existente. Por lo tanto definiremos el role de admin como:

```
CREATE role warehouse_admin WITH
      SUPERUSER
      NOCREATEDB
      CREATEROLE
      LOGIN
      CONNECTION LIMIT 1
      PASSWORD 'a123456';
```

El rol para acceso operacional, que limitaremos en el tiempo para mayor seguridad:

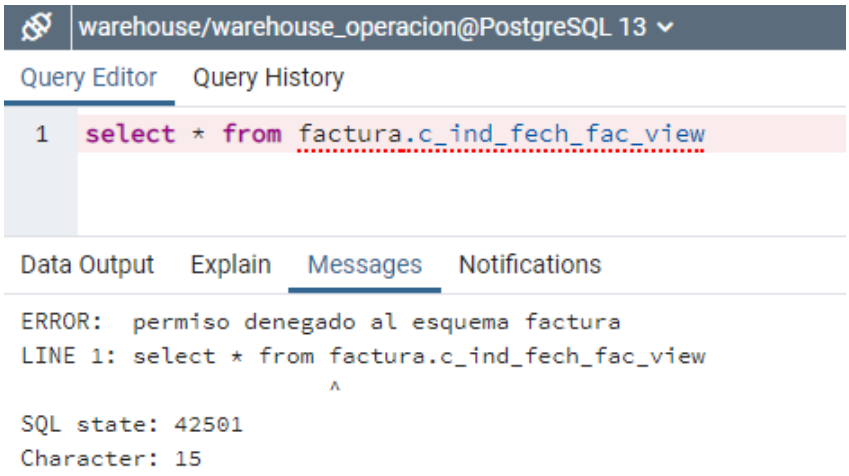
```
CREATE role warehouse_operacion WITH
      LOGIN
      PASSWORD 'a123456' VALID UNTIL '2023-12-31';
```

Y finalmente el usuario de solo lectura para consulta de datos que puede ser usado por una aplicación de usuario o de reporting:

```
CREATE role warehouse_lectura WITH
      LOGIN
      PASSWORD 'a123456' VALID UNTIL '2023-12-31';
```

1.https://www.ibm.com/docs/es/was-nd/8.5.5?topic=SSAW57_8.5.5/com.ibm.websphere.zseries.doc/ae/rsec_racftools.htm

A diferencia de warehouse_admin, que al haber sido definido como superuser tiene acceso a todos los elementos de la base de datos, ni warehouse_operacion ni warehouse_lectura tienen ningún tipo de permiso todavía, e intentar leer cualquier tabla o vista retornará error, mejorando así la seguridad.



9 Error de lectura por falta de autorización

El siguiente paso es por tanto asignar a cada uno los accesos que le correspondan, y esto se realiza en mediante la instruccion GRANT (o REVOKE para eliminar permisos).

Determinamos que warehouse_operacion debe tener acceso a todas las tablas, vistas, y funciones, excepto las tablas del esquema de diccionario que solo pueden ser mantenidas por el admin, por lo que definiremos inicialmente:

a) Que pueda conectar a la base de datos

```
GRANT CONNECT ON DATABASE warehouse TO warehouse_operacion;
```

b) Que pueda acceder completamente a los esquemas y sus tablas/vistas

```
GRANT ALL PRIVILEGES ON SCHEMA almacen TO warehouse_operacion;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA almacen TO warehouse_operacion;
GRANT ALL PRIVILEGES ON SCHEMA asociado TO warehouse_operacion;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA asociado TO warehouse_operacion;
GRANT ALL PRIVILEGES ON SCHEMA factura TO warehouse_operacion;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA factura TO warehouse_operacion;
GRANT ALL PRIVILEGES ON SCHEMA orden TO warehouse_operacion;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA orden TO warehouse_operacion;
```

Por otro lado, determinamos que el rol warehouse_lectura que puede ser utilizado por alguna aplicación externa solo pueda tener acceso de tipo SELECT y solo a las vistas, limitando así la visibilidad de datos y mejorando como se ha comentado la seguridad gracias a los roles, obteniendo:

a) Que pueda conectar con la base de datos

```
GRANT CONNECT ON DATABASE warehouse TO warehouse_lectura;
```

b) Que pueda acceder so lo a las vistas

```
GRANT USAGE  ON SCHEMA almacen TO warehouse_lectura;
GRANT SELECT ON TABLE almacen.inventario TO warehouse_lectura;
GRANT USAGE  ON SCHEMA asociado TO warehouse_lectura;
GRANT SELECT ON TABLE asociado.v_cliente_estadistica_mensual
```

```
            TO warehouse_lectura;
GRANT SELECT ON TABLE asociado.v_cliente_estadistica_anuaL
            TO warehouse_lectura;
GRANT USAGE  ON SCHEMA factura TO warehouse_lectura;
GRANT SELECT ON TABLE factura.c_ind_fech_fac_view  TO warehouse_lectura;
GRANT SELECT ON TABLE factura.v_ind_fech_fac_view  TO warehouse_lectura;
GRANT USAGE  ON SCHEMA orden TO warehouse_lectura;
GRANT SELECT ON TABLE orden.backorden_pendiente  TO warehouse_lectura;
GRANT SELECT ON TABLE orden.detalle_pedido   TO warehouse_lectura;
```

6. Test de rendimiento y mejoras

Una parte importante en el diseño de una base de datos es que el rendimiento sea óptimo, por lo que suele ser común definir índices con este objetivo. Ya se había comentado en la sección 4.6 que los índices/set definidos en IDMS, que como se muestra en el diagrama de Bachman para nuestro caso de estudio eran C_IND_FECH_FAC y V_IND_FECH_FAC, proporcionaban una forma de acceso filtrado y ordenado a la información, y que por tanto el funcionamiento de estos SET/índice en IDMS era conceptualmente diferente de PostgreSQL.

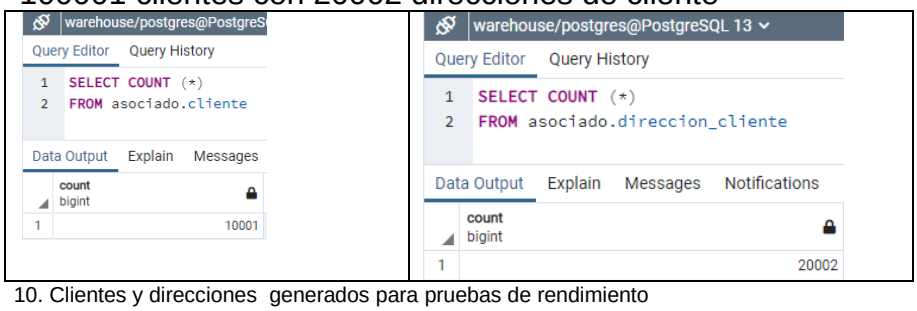
Por lo tanto, en este paso final abordaremos el proceso de definición de índices y mejora del rendimiento, y para ello tomaremos como ejemplo la vista orden.v_detalle_pedido. El proceso para el resto de vistas o tablas sería similar, quedando fuera del presente trabajo de fin de grado por limitaciones en la extensión del mismo.

6.1 Caso de ejemplo sobre vista factura.c_ind_fech_fac_view.

Para poder hacer un estudio del rendimiento necesitamos tener un cierto volumen de datos: en un caso real, dispondríamos de los datos de origen y podríamos hacer una carga y prueba sobre los mismos. Para poder simular esto de la mejor forma posible se ha creado una función que genere datos con un mínimo de calidad y que se adjuntará en el entregable. Puede consultarse en el fichero de generación de la bbdd.

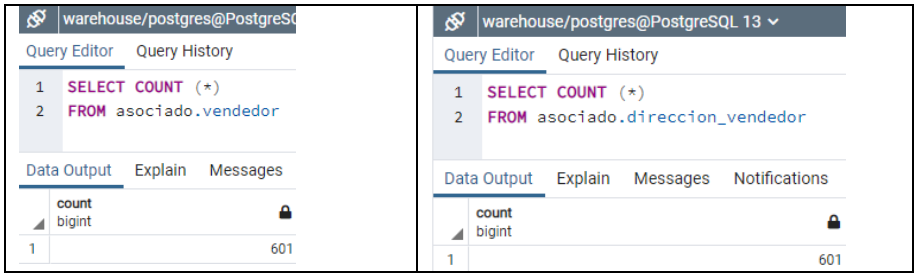
Tras ejecutarla, podemos ver que se han generado:

- 100001 clientes con 20002 direcciones de cliente



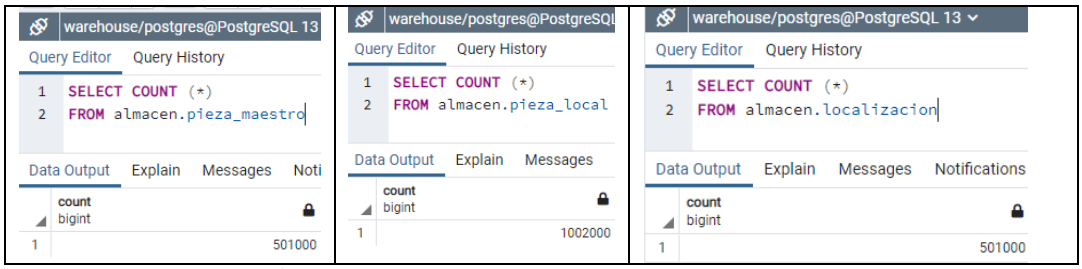
10. Clientes y direcciones generados para pruebas de rendimiento

- 601 vendedores con 601 direcciones de vendedor



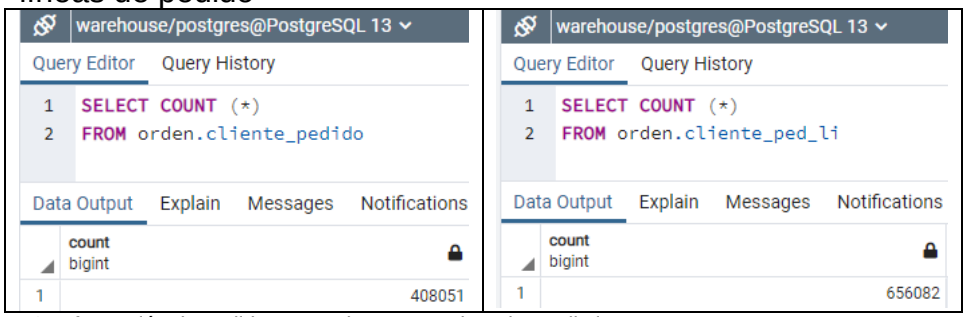
11. Vendedores y direcciones generados para pruebas de rendimiento

- 501000 partes con 1002000 versiones repartidas entre 501000 localizaciones en el almacén



12. Información de piezas generada para pruebas de rendimiento

- 408501 pedido (activos y cancelados) de cliente con 656082 líneas de pedido



13. Información de pedido generada para pruebas de rendimiento

- 8001 facturas de cliente, con un total de 328041 linea

14. Información de facturas generada para pruebas de rendimiento

Una vez completado este paso, analizaremos la vistas definidas y las consultas más comunes posibles sobre la misma. Para ello usaremos el comando EXPLAIN ANALYZE[1] de postgres que nos mostrará el plan seleccionado por el SGBD que incluye información de qué ruta o índices se ha ejecutado para obtener la información, así como los tiempos empleados.

Finalmente definiremos índices sobre los campos clave que identifiquemos y volveremos a ejecutar EXPLAIN ANALYZE para comparar si realmente ha habido una mejora

orden.v_detalle_pedido es una vista básica en las operaciones de consulta de pedidos de cliente. Su uso habitual podría ser:

- mostrar un listado completo de pedidos con factura activos para generar un informe para una auditoría
- consultar los pedidos pendientes de completar propios de un cliente
- o consultar el detalle de un pedido concreto para mostrarlo en pantalla

Si realizamos EXPLAIN ANALYZE sobre la definición inicial para estas 3 consultas se nos mostrará cómo el SGBD decide recuperar los datos y los tiempos, así obtendríamos:

```
1  -- 1)mostrar un listado completo de pedidos con factura activos para generar un informe para una auditoría
2  -- =>aquellos no cancelados, con factura informada
3  EXPLAIN ANALYZE
4  SELECT * FROM orden.v_detalle_pedido WHERE estado > '0' AND id_factura IS NOT NULL
5
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=45667.88..115655.49 rows=463381 width=127) (actual time=477.128..750.716 rows=640080 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (cost=44667.88..68317.39 rows=193075 width=127) (actual time=449.293..576.181 rows=213360 loops=3)
5	Hash Cond: (linea.id_pieza = pieza.id_pieza_gen)
6	-> Parallel Hash Join (cost=18244.13..32768.06 rows=193075 width=38) (actual time=200.356..270.883 rows=213360 loops=3)
7	Hash Cond: (linea.id_pedido = pedido.id_pedido)
8	-> Parallel Seq Scan on cliente_ped_li linea (cost=0.00..7756.05 rows=306705 width=22) (actual time=0.013..24.711 rows=245364 loops=3)
9	-> Parallel Hash (cost=15469.34..15469.34 rows=151103 width=24) (actual time=134.835..134.940 rows=106680 loops=3)
10	Buckets: 65536 Batches: 8 Memory Usage: 2752kB
11	-> Parallel Hash Join (cost=8949.58..15469.34 rows=151103 width=24) (actual time=82.578..117.522 rows=106680 loops=3)
12	Hash Cond: (factura.id_pedido_orig = pedido.id_pedido)
13	-> Parallel Seq Scan on c_factura_linea_pedido factura (cost=0.00..3612.59 rows=188259 width=8) (actual time=0.011..11.259 rows=106680 loops=3)
14	Filter: (id_factura IS NOT NULL)
15	-> Parallel Hash (cost=5600.38..5600.38 rows=192656 width=16) (actual time=51.721..51.721 rows=109347 loops=3)
16	Buckets: 131072 Batches: 8 Memory Usage: 2976kB
17	-> Parallel Seq Scan on cliente_pedido pedido (cost=0.00..5600.38 rows=192656 width=16) (actual time=0.015..33.614 rows=109347 loops=3)
18	Filter: ((estado)::text > '0'::text)
19	Rows Removed by Filter: 26670
20	-> Parallel Hash (cost=17535.00..17535.00 rows=417500 width=43) (actual time=105.732..105.732 rows=334000 loops=3)
21	Buckets: 65536 Batches: 32 Memory Usage: 3040kB
22	-> Parallel Seq Scan on pieza_local pieza (cost=0.00..17535.00 rows=417500 width=43) (actual time=0.169..47.422 rows=334000 loops=3)
23	Planning Time: 0.422 ms
24	Execution Time: 763.175 ms

15. EXPLAIN ANALYZE pedidos con factura sobre v_detalle_pedido pre-optimización

1.https://www.postgresql.org/docs/current/sql-explain.html

Donde vemos los *joins* de la vista en los 'Hash cond' y el escaneo secuencial, así como los costes de las condiciones. El tiempo invertido mostrado es un resultado medio tras ejecutarlo varias veces.

```
1  -- 2)consultar los pedidos pendientes de completar propios de un cliente
2  -- =>aquellos en estado 1=activo, filtrado por un cliente escogido al azar
3  EXPLAIN ANALYZE
4  SELECT * FROM orden.v_detalle_pedido WHERE estado = '1' AND cif = 'A00002000';
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=7205.91..13434.97 rows=74 width=127) (actual time=35.929..56.708 rows=82 loops=1)
2	-> Nested Loop (cost=7205.49..13395.01 rows=74 width=38) (actual time=35.921..56.583 rows=82 loops=1)
3	-> Hash Right Join (cost=7205.06..12975.58 rows=41 width=20) (actual time=35.904..56.496 rows=41 loops=1)
4	Hash Cond: (factura.id_pedido_orig = pedido.id_pedido)
5	-> Seq Scan on c_factura_linea_pedido factura (cost=0.00..4930.40 rows=320040 width=8) (actual time=0.018..11.206 rows=320040 loops=1)
6	-> Hash (cost=7204.55..7204.55 rows=41 width=16) (actual time=28.932..28.963 rows=41 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 10kB
8	-> Gather (cost=1000.00..7204.55 rows=41 width=16) (actual time=8.316..28.948 rows=41 loops=1)
9	Workers Planned: 1
10	Workers Launched: 1
11	-> Parallel Seq Scan on cliente_pedido pedido (cost=0.00..6200.45 rows=24 width=16) (actual time=7.384..17.631 rows=21 loops=2)
12	Filter: (((estado)::text = '1'::text) AND ((cif)::text = 'A00002000'::text))
13	Rows Removed by Filter: 204005
14	-> Index Scan using cliente_ped_li_pk on cliente_ped_li linea (cost=0.42..10.21 rows=2 width=22) (actual time=0.001..0.002 rows=2 loops=41)
15	Index Cond: (id_pedido = pedido.id_pedido)
16	-> Index Scan using pieza_local_pk on pieza_local pieza (cost=0.42..0.53 rows=1 width=43) (actual time=0.001..0.001 rows=1 loops=82)
17	Index Cond: (id_pieza_gen = linea.id_pieza)
18	Planning Time: 0.358 ms
19	Execution Time: 56.738 ms

16. EXPLAIN ANALYZE pedidos pendientes de un cliente sobre v_detalle_pedido pre-optimización

En esta ocasión podemos ver el uso de los índices propios por las claves primarias definidas, por ejemplo en la línea 14.

```
13  -- 3) consultar el detalle de un pedido concreto para mostrarlo en pantalla
14  -- => consulta por número de pedido
15  EXPLAIN ANALYZE
16  SELECT * FROM orden.v_detalle_pedido WHERE id_pedido = '892';
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=1001.27..5118.92 rows=2 width=127) (actual time=0.234..25.213 rows=2 loops=1)
2	-> Nested Loop (cost=1000.85..5102.02 rows=2 width=38) (actual time=0.227..25.201 rows=2 loops=1)
3	-> Nested Loop Left Join (cost=1000.42..5091.79 rows=1 width=20) (actual time=0.216..25.189 rows=1 loops=1)
4	Join Filter: (pedido.id_pedido = factura.id_pedido_orig)
5	-> Index Scan using cliente_pedido_pk on cliente_pedido pedido (cost=0.42..8.44 rows=1 width=16) (actual time=0.006..0.008 rows=1 loops=1)
6	Index Cond: (id_pedido = 892)
7	-> Gather (cost=1000.00..5083.34 rows=1 width=8) (actual time=0.208..25.177 rows=1 loops=1)
8	Workers Planned: 1
9	Workers Launched: 1
10	-> Parallel Seq Scan on c_factura_linea_pedido factura (cost=0.00..4083.24 rows=1 width=8) (actual time=0.018..8.119 rows=1 loops=2)
11	Filter: (id_pedido_orig = 892)
12	Rows Removed by Filter: 160020
13	-> Index Scan using cliente_ped_li_pk on cliente_ped_li linea (cost=0.42..10.21 rows=2 width=22) (actual time=0.010..0.010 rows=2 loops=1)
14	Index Cond: (id_pedido = 892)
15	-> Index Scan using pieza_local_pk on pieza_local pieza (cost=0.42..8.44 rows=1 width=43) (actual time=0.003..0.003 rows=1 loops=2)
16	Index Cond: (id_pieza_gen = linea.id_pieza)
17	Planning Time: 0.202 ms
18	Execution Time: 25.239 ms

17. EXPLAIN ANALYZE de datos de un pedido concreto sobre v_detalle_pedido pre-optimización

Se observa que en este caso usa los índices de la clave primaria de pedido y su participación en el join (línea 4 y 5) del estado.

Si deseamos optimizar las consultas debemos intentar encontrar qué campos de la definición de la vista son clave para el filtrado como para las *joins* con otras tablas. En nuestro caso recordemos que la tabla de factura puede no contener dato relacionado:

```
CREATE VIEW orden.v_detalle_pedido AS
SELECT pedido.id_pedido, pedido.cif, pedido.estado, factura.id_factura,
       linea.num_linea, pieza.id_pieza, pieza.descripcion, linea.cantidad,
       linea.coste_neto_unidad * linea.cantidad AS "total neto",
       linea.coste_bruto_unidad * linea.cantidad AS "total bruto"
FROM orden.cliente_ped_li linea, almacen.pieza_local pieza,
     orden.cliente_pedido pedido
      LEFT JOIN factura.c_factura_linea_pedido factura
      ON pedido.id_pedido = factura.id_pedido_orig
WHERE linea.id_pedido = pedido.id_pedido
  and linea.id_pieza = pieza.id_pieza_gen;
```

Observamos que muchas de las consultas están orientadas a un cliente concreto, por lo que el campo cif sería un candidato ya que además no forma parte del índice de la clave primaria. Por lo tanto, definimos el siguiente índice y repetimos las consultas:

```
CREATE INDEX IF NOT EXISTS   ON orden.cliente_pedido (cif);
```

```
1  -- 1)mostrar un listado completo de pedidos con factura activos para generar un informe para una auditoría
2  -- =>aquellos no cancelados, con factura informada
3  EXPLAIN ANALYZE
4  SELECT * FROM orden.v_detalle_pedido WHERE estado > '0' AND id_factura IS NOT NULL
5
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Gather (cost=45667.88..115655.49 rows=463381 width=127) (actual time=509.231..767.278 rows=640080 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Hash Join (cost=44667.88..68317.39 rows=193075 width=127) (actual time=517.679..630.829 rows=213360 loops=3)
5	Hash Cond: (linea.id_pieza = pieza.id_pieza_gen)
6	-> Parallel Hash Join (cost=18244.13..32768.06 rows=193075 width=38) (actual time=188.948..253.628 rows=213360 loops=3)
7	Hash Cond: (linea.id_pedido = pedido.id_pedido)
8	-> Parallel Seq Scan on cliente_ped_li linea (cost=0.00..7756.05 rows=306705 width=22) (actual time=0.014..25.058 rows=245364 loops=3)
9	-> Parallel Hash (cost=15469.34..15469.34 rows=151103 width=24) (actual time=124.034..124.035 rows=106680 loops=3)
10	Buckets: 65536 Batches: 8 Memory Usage: 2752kB
11	-> Parallel Hash Join (cost=8949.58..15469.34 rows=151103 width=24) (actual time=76.120..107.946 rows=106680 loops=3)
12	Hash Cond: (factura.id_pedido_orig = pedido.id_pedido)
13	-> Parallel Seq Scan on c_factura_linea_pedido factura (cost=0.00..3612.59 rows=188259 width=8) (actual time=0.009..9.131 rows=106680 loops=3)
14	Filter: (id_factura IS NOT NULL)
15	-> Parallel Hash (cost=5600.38..5600.38 rows=192656 width=16) (actual time=50.261..50.261 rows=109347 loops=3)
16	Buckets: 131072 Batches: 8 Memory Usage: 2976kB
17	-> Parallel Seq Scan on cliente_pedido pedido (cost=0.00..5600.38 rows=192656 width=16) (actual time=0.019..31.927 rows=109347 loops=3)
18	Filter: ((estado)::text > '0'::text)
19	Rows Removed by Filter: 26670
20	-> Parallel Hash (cost=17535.00..17535.00 rows=417500 width=43) (actual time=194.539..194.540 rows=334000 loops=3)
21	Buckets: 65536 Batches: 32 Memory Usage: 3008kB
22	-> Parallel Seq Scan on pieza_local pieza (cost=0.00..17535.00 rows=417500 width=43) (actual time=0.230..48.747 rows=334000 loops=3)
23	Planning Time: 0.385 ms
24	Execution Time: 779.042 ms

18. EXPLAIN ANALYZE de pedidos sobre v_detalle_pedido post-optimización

Como esperábamos, en este caso no ha habido cambio en el planificador pues el campo usado en el índice no es parte de las *joins* ni las condiciones.

```
1  -- 2)consultar los pedidos pendientes de completar propios de un cliente
2  -- =>aquellos en estado 1=activo, filtrado por un cliente escogido al azar
3  EXPLAIN ANALYZE
4  SELECT * FROM orden.v_detalle_pedido WHERE estado = '1' AND cif = 'A00002000';
```

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=10.80..6239.86 rows=74 width=127) (actual time=6.920..28.285 rows=82 loops=1)
2	-> Nested Loop (cost=10.38..6199.90 rows=74 width=38) (actual time=6.912..28.159 rows=82 loops=1)
3	-> Hash Right Join (cost=9.95..5780.47 rows=41 width=20) (actual time=6.895..28.075 rows=41 loops=1)
4	Hash Cond: (factura.id_pedido_orig = pedido.id_pedido)
5	-> Seq Scan on c_factura_linea_pedido factura (cost=0.00..4930.40 rows=320040 width=8) (actual time=0.007..11.458 rows=320040 loops=1)
6	-> Hash (cost=9.44..9.44 rows=41 width=16) (actual time=0.038..0.039 rows=41 loops=1)
7	Buckets: 1024 Batches: 1 Memory Usage: 10kB
8	-> Index Scan using v_detalle_pedido_idx on cliente_pedido pedido (cost=0.42..9.44 rows=41 width=16) (actual time=0.024..0.032 rows=41 loops...
9	Index Cond: ((cif)::text = 'A00002000'::text)
10	Filter: ((estado)::text = '1'::text)
11	Rows Removed by Filter: 10
12	-> Index Scan using cliente_ped_li_pk on cliente_ped_li linea (cost=0.42..10.21 rows=2 width=22) (actual time=0.001..0.002 rows=2 loops=41)
13	Index Cond: (id_pedido = pedido.id_pedido)
14	-> Index Scan using pieza_local_pk on pieza_local pieza (cost=0.42..0.53 rows=1 width=43) (actual time=0.001..0.001 rows=1 loops=82)
15	Index Cond: (id_pieza_gen = linea.id_pieza)
16	Planning Time: 0.327 ms
17	Execution Time: 28.315 ms

19. EXPLAIN ANALYZE pedidos pendientes de un cliente sobre v_detalle_pedido post-optimización

En esta ocasión sí puede apreciarse que prácticamente se ha dividido los tiempos a la mitad, de 56 ms a 28 ms, y cómo se usa el índice en la línea 8 con su condición antes de aplicar el filtrado de estado, por lo que este sería un caso válido de optimización candidato.

```
10 -- 3) consultar el detalle de un pedido concreto para mostrarlo en pantalla
11 -- => consulta por número de pedido
12 EXPLAIN ANALYZE
13 SELECT * FROM orden.v_detalle_pedido WHERE id_pedido = '892';
```

Data Output | Explain | Messages | Notifications

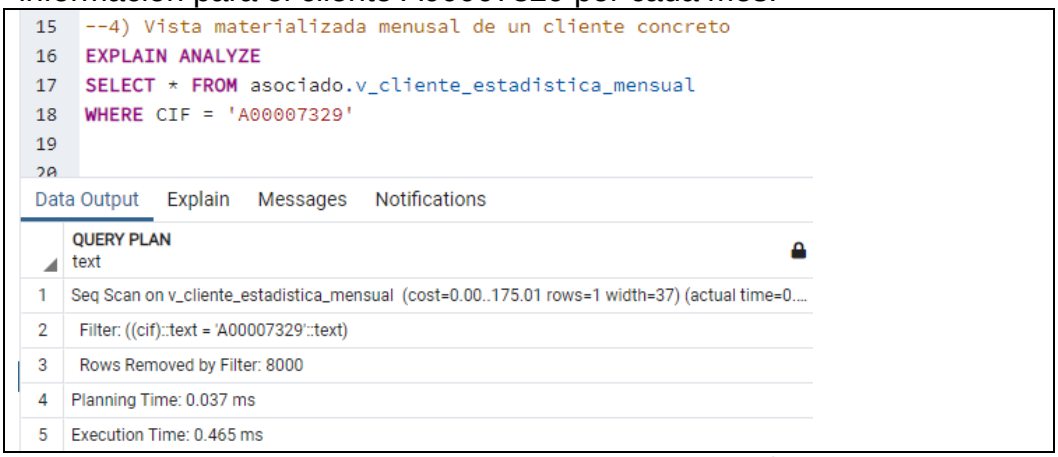
	QUERY PLAN text
1	Nested Loop (cost=1001.27..5118.92 rows=2 width=127) (actual time=0.245..24.026 rows=2 loops=1)
2	-> Nested Loop (cost=1000.85..5102.02 rows=2 width=38) (actual time=0.239..24.017 rows=2 loops=1)
3	-> Nested Loop Left Join (cost=1000.42..5091.79 rows=1 width=20) (actual time=0.231..24.008 rows=1 loops=1)
4	Join Filter: (pedido.id_pedido = factura.id_pedido_orig)
5	-> Index Scan using cliente_pedido_pk on cliente_pedido pedido (cost=0.42..8.44 rows=1 width=16) (actual time=0.006..0.008 rows=1 loops=1)
6	Index Cond: (id_pedido = 892)
7	-> Gather (cost=1000.00..5083.34 rows=1 width=8) (actual time=0.224..23.997 rows=1 loops=1)
8	Workers Planned: 1
9	Workers Launched: 1
10	-> Parallel Seq Scan on c_factura_linea_pedido factura (cost=0.00..4083.24 rows=1 width=8) (actual time=0.019..7.106 rows=1 loops=2)
11	Filter: (id_pedido_orig = 892)
12	Rows Removed by Filter: 160020
13	-> Index Scan using cliente_ped_li_pk on cliente_ped_li linea (cost=0.42..10.21 rows=2 width=22) (actual time=0.007..0.008 rows=2 loops=1)
14	Index Cond: (id_pedido = 892)
15	-> Index Scan using pieza_local_pk on pieza_local pieza (cost=0.42..8.44 rows=1 width=43) (actual time=0.002..0.003 rows=1 loops=2)
16	Index Cond: (id_pieza_gen = linea.id_pieza)
17	Planning Time: 0.199 ms
18	Execution Time: 24.052 ms

20. EXPLAIN ANALYZE de datos de un pedido concreto sobre v_detalle_pedido post-optimización

Para la consulta por pedido directo vemos que el SGBD sigue determinando que la forma más rápida es mediante la clave primaria, lo cual es lógico ya que es el valor proporcionado en la clave.

6.2 Caso de ejemplo sobre vista asociado.v_cliente_estadistica_mensual.

Otro posible caso de estudio de optimización sería sobre las vistas materializadas: a diferencia de las vistas normales, éstas sí permiten definir índices. Supongamos el caso en que se desea saber la información para el cliente A00007329 por cada mes.



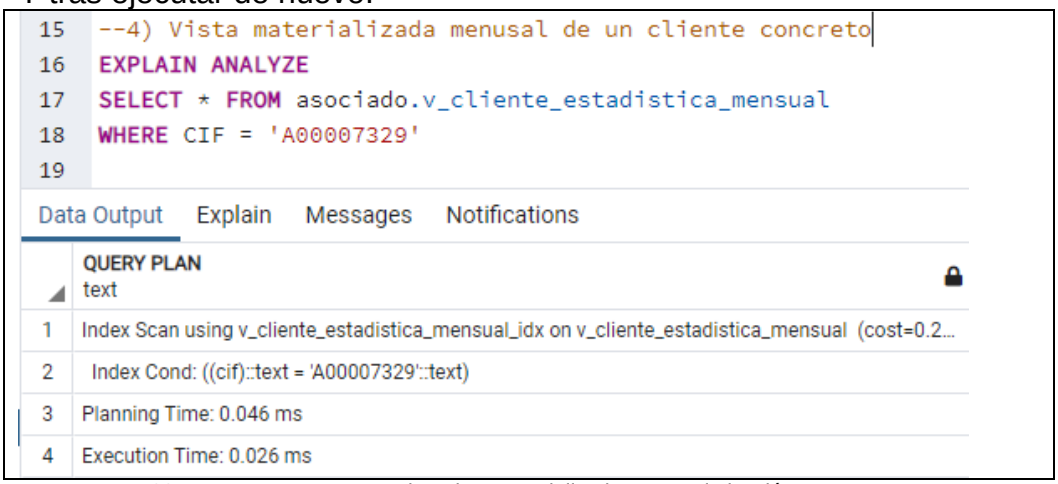
21. EXPLAIN ANALYZE sobre vista materializada pre-optimización

Como era de esperar, se escanea toda la vista y se filtra por el campo de la WHERE.

Definamos ahora un índice específico sobre dicho atributo:

```
CREATE INDEX IF NOT EXISTS v_cliente_estadistica_mensual_idx ON asociado.v_cliente_estadistica_mensual(cif);
```

Y tras ejecutar de nuevo:



22. EXPLAIN ANALYZE sobre vista materializada post-optimización

Vemos un incremento drástico en el rendimiento, pasando de 0.465ms a 0.026ms, por lo que podría ser un índice candidato. Pero deberíamos también preguntarnos hasta qué punto debemos definir índices, pues

estos también ocupan espacio en las bases de datos y tienen un mantenimiento asociado. En el ejemplo anterior sobre la estadística mensual no solo se trata de una tabla pequeña por lo que por muy grande que sea la mejora, el beneficio es mínimo, sino que al escoger un campo único, el cif, el índice tendrá el tamaño máximo.

Para completar este último paso de la migración, se procedería de forma similar con todas las vistas creadas, así como posibles consultas más comunes.

7. Conclusiones

Considero que el resultado final cumple correctamente con los objetivos definidos: hemos partida de una base de datos IDMS inicial bastante alejada del modelo relacional, y hemos identificado una serie de normas genéricas, reglas y procesos, que tras ser aplicadas de forma iterativa nos ha permitido obtener una base de datos relacional PostgreSQL.

Esto ha sido en parte posible debido a una buena planificación inicial. Sí es cierto que en la segunda fase se infravaloró el coste en tiempo para realizar el diseño inicial, en parte impactado por los problemas con la licencia de VISIO, aplicando las medidas de contingencia definidas se pudo paliar parte del impacto, aunque se tuvo que reducir los atributos de las tablas al número mínimo de campos para cumplir con su función. Otro factor que no se tuvo en cuenta inicialmente fue la extensión del trabajo, que al estar limitado a 90 páginas se ha tenido que recortar alguna explicación y no se ha podido realizar tantas pruebas y mejoras de rendimiento como inicialmente estaba planificado. Por lo demás, las estimaciones fueron bastante próximas a la realidad.

Como conclusión final, creo que el trabajo ha sido muy interesante y enriquecedor, presenta al lector un sistema de bases de datos del que muchos no habrán oído hablar y que puede parecer ‘arcaico’ pero que aun esta presente en grandes compañías, y ofrece de forma razonada una serie de pasos y una metodología que, salvando las distancia entre SGBD, puede aplicarse a cualquier migración.

Respecto a futuras líneas de trabajo, varias opciones son:

- Continuar con las mejoras de rendimiento, especialmente para los datos de vendedor que no pudieron ser probados.
- Ampliar atributos y tablas para dar cobertura a otros requisitos
- Implementar un sistema de migración de datos a tablas de histórico.
- Mejoras en los triggers y funciones
- Realizar un diseño para postgres en el Cloud
- Integrar la base de datos con diferentes herramientas de monitorización como puede ser Grafana
- Y por supuesto, fuera del ámbito de las bases de datos una posible continuación futura sería desarrollar una aplicación sobre nuestra base de datos que de soporte a los diferentes roles definidos.

8. Glosario

- SGBD: sistema gestor de bases de datos.

- RECORD: es la definición de una entidad en IDMS, pudiendo ser de tipo CALC si cada tupla se identifica mediante una clave lógica o VIA, que son accesibles desde la CALC. Ambos son accesibles conociendo la dirección física real.

- SET: es una definición lógica de la relación entre 2 RECORD, en que uno hace de propietario, y otro de miembro.

- ERP: se trata de un conjunto de aplicaciones con el objetivo de cubrir las necesidades de los procesos de negocio de una empresa y los objetivos estratégicos.

- SaaS: acrónimo de 'Software as a Service', modelo en que las empresas pagan por el uso de los programas en lugar de ser propietarias del mismo.

- Proceso Batch: en entornos Mainframe, aquellos procesos ejecutados desde un Job de forma programada.

- Proceso Online: en entornos Mainframe, aquellos procesos ejecutados a consecuencia de una acción u evento.

- Forma normal: conjunto de criterios para determinar el grado de dependencias, redundancia, consistencia e integridad de una tabla.

- Triggers: procesos que ante un evento ejecutan una funciona almacenada en la base de datos.

- Diagrama de Bachmann: forma de representar las relaciones entre RECORDS y SETS, así como todas sus características.

- SQL: acrónimo de Structured Query Language.

9. Bibliografía

- Date, C.J. *Introducción a los sistemas de bases de datos*. Séptima edición, Prentice Hall, 2001.

- Karasz, Peter. *IDMS History. The Origins of IDMS*. http://www.manmrk.net [en línea] Recuperado el 10 de marzo de 2023 de http://www.manmrk.net/tutorials/database/IDMS/IDMSHistory.htm

- *CODASYL*. https://es.dbpedia.org/ [en línea] Recuperado el 09 de marzo de 2023 de https://es.dbpedia.org/page/CODASYL

- SAS Institute Inc. *IDMS Essentials*. [en línea] Recuperado el 02 de marzo de 2023 de https://www.sfu.ca/sasdoc/sashtml/idms/z0376933.htm

- Broadcom. *SETS*. https://techdocs.broadcom.com/ [En línea] Recuperado el 08 de abril de 2023 de https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/programming/navigational-dml-programming-reference/database-concepts/sets.html

- Broadcom. *RECORD Statement*. https://techdocs.broadcom.com/ [En línea] Recuperado el 10 de abril de 2023 de https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms/19-0/administrating/idms-database/schema-statements/record-statement.html

- Embarcadero. *Developing a Set of Rules (Designing Databases)*. https://docwiki.embarcadero.com/InterBase/2020/en/Main_Page [En línea] Recuperado el 29 de Febrero de 2023 de https://docwiki.embarcadero.com/InterBase/2020/en/Developing_a_Set_of_Rules_(Designing_Databases)

- IBM. *Normalización para evitar redundancias*. https://www.ibm.com/ [En línea] Recuperado el 05 de abril de 2023 de https://www.ibm.com/docs/es/db2-for-zos/11?topic=modeling-normalization-in-database-design

10. Anexos

Los anexos pueden encontrarse en la carpeta correspondiente de la entrega.