



Universidad Nacional
de San Martín

Proyecto: Bases de Datos

Alumnos:

- *Mecozzi, Tamara Eleonor*
- *Rodríguez, Lucas G.*

Índice

Introducción.....	4
Parte 1: Análisis y Limpieza de Datos.....	5
Análisis de los Datos Existentes.....	5
Organización y Normalización de Datos.....	5
Estandarización y Separación de Campos.....	5
División de la Columna "Descripción" en Subcampos.....	6
Pasos de Limpieza y Organización Realizados (Data Preprocessing).....	6
Propuestas para Datos Incompletos o Faltantes.....	6
Parte 2: Selección y Justificación del Motor de Base de Datos.....	7
Selección y Justificación del Motor de Base de Datos.....	7
Comparación con una Alternativa No Relacional: MongoDB.....	7
Comparación entre PostgreSQL y MongoDB.....	8
Parte 3: Diseño de Modelo de Datos y Creación del Esquema.....	9
Diseño del Modelo Entidad-Relación.....	9
Entidades.....	9
Relaciones.....	9
Implementación del Esquema en SQL.....	10
Creación de la Base de Datos Inicial.....	10
Creación de tablas.....	11
Introducción a la creación de las tablas relacionales y claves FK.....	12
Crear las Tablas de Relaciones y determinación de las claves foránea.....	12
Verificar las Tablas.....	14
Diagrama Entidad-Relación (E-R).....	15
Modelo lógico y restricciones.....	17
Parte 4: Carga y Manipulación de Datos.....	18
Crear la tabla provisoria.....	18
Carga de los datos a las respectivas tablas.....	18
Consultas Básicas y Reportes.....	26
Punto A).....	27
Punto B).....	29
Punto C).....	30
Punto D).....	32
Consultas Avanzadas y Optimización.....	33
Punto A).....	33
Punto B).....	35
Parte 5: Procedimientos, Triggers y Automatización.....	37
Análisis Teórico de Procedimientos Almacenados.....	37
¿Qué es un procedimiento almacenado y para qué se usa?.....	37
Cómo utilizar el PROCEDURE en PostgreSQL.....	38
Ejemplo de caso de uso del procedure en PostgreSQL.....	44

Creación del schema y de las tablas que vamos a utilizar.....	44
Creado y carga de datos de la tabla alumnos.....	44
Creado y carga de datos de la tabla materias.....	44
Creado y carga de datos de la tabla inscripción.....	45
Creación del procedure.....	45
Utilización del procedure creado.....	45
Análisis Teórico de Triggers.....	46
¿Que es un trigger y para que se usa?.....	46
Ejemplo de caso de uso del trigger en PostgreSQL.....	48
Creación y carga de datos en la tabla alumnos.....	49
Creación de la tabla log que queremos utilizar.....	49
Creación de la función del trigger.....	50
Creación del trigger que vamos a utilizar.....	51
Ubicación de donde se nos crea el trigger y la función de este mismo.....	52
Ejemplos de caso de uso del trigger creado anteriormente.....	53
Propuesta de Estrategias de Automatización.....	56
Índices de optimización.....	56
Uso de vistas.....	57
Ejemplo de vista estática.....	58
Programación de Tareas.....	59
Principales Aplicaciones.....	59
Backups.....	60
Automatización de Backups.....	60
Comando para hacer un backup con la extension .backup.....	60
Comando para hacer un backup con la extensión .sql.....	61
Creación de la tabla alumnos y el cargado de los datos.....	61
Creación de la tabla materias y el cargado de los datos.....	62
Creación de la tabla inscripciones y cargado de los datos.....	62
Creación y uso de los índices de optimización.....	63
Creación y uso de la vista estándar.....	64
Conclusión Final.....	67
Bibliografía.....	69

Introducción

En el presente trabajo práctico, vamos a trabajar en el desarrollo de una base de datos que centralice y organice toda la información académica de los estudiantes de la Universidad Nacional de San Martín. Actualmente, la universidad cuenta con los datos de los alumnos dispersos y con poca facilidad de acceso. Por eso, nuestro objetivo es diseñar una solución que permita gestionar y analizar los datos de manera eficiente, contribuyendo a mejorar la experiencia educativa.

El proyecto se llevará a cabo en varias etapas:

1. **Análisis y Limpieza de Datos:** Vamos a trabajar en organizar y normalizar los datos iniciales, los cuales presentan inconsistencias y errores. Esto incluirá la separación de campos combinados, la estandarización de valores y la eliminación de duplicados.
2. **Selección del Motor de Base de Datos:** Mostraremos el motor de base de datos elegido, destacando sus ventajas para manejar consultas complejas, integridad de los datos y escalabilidad.
3. **Diseño del Modelo de Datos:** Elaboramos un Diagrama Entidad-Relación (DER) para definir cómo se relacionan las distintas partes de la información, como alumnos, materias, grupos, hobbies, entre otros.
4. **Carga y Manipulación de Datos:** Utilizaremos archivos CSV para cargar la información en las tablas creadas. También generamos consultas avanzadas que permitirán extraer datos.
5. **Procedimientos y Automatización:** Implementaremos procedimientos almacenados para realizar tareas como la inscripción de alumnos a materias de manera automática y eficiente.

Nuestro fin es contribuir en una base de datos que además de organizar la información, también permite analizar para tomar decisiones. Por ejemplo, mostraremos cómo identificar patrones en los intereses y comportamientos de los alumnos, lo que podría ayudar a la universidad a ajustar su oferta académica.

Por lo tanto, esta base de datos busca facilitar el acceso y la gestión de los datos de los alumnos, proporcionando herramientas analíticas que apoyen la planificación y toma de decisiones de la universidad.

Parte 1: Análisis y Limpieza de Datos

Análisis de los Datos Existentes

Para iniciar realizamos una inspección exhaustiva de los datos proporcionados en el archivo original para identificar campos clave y detectar información incompleta. En este análisis, detectamos los siguientes elementos:

- **Campos claves:** identificamos los más relevantes, como "Nombre", "Apellido", "Grupo", "Rol", entre otros.
- **Campos incompletos:** Se detectaron datos faltantes o incompletos en varios registros, especialmente en el campo "Descripción", donde la información estaba desorganizada.
- **Formatos inconsistentes:** Observamos variaciones en la forma en que se ingresaron ciertos datos, como abreviaciones o errores de tipo en nombres y localidades. Los nombres y apellidos estaban en un solo campo, lo que complicaba su análisis y requería separación y estandarización.

Con esta revisión inicial, se obtuvo una visión clara de los elementos a normalizar y los campos a dividir para mejorar la estructura y calidad de la base de datos.

Organización y Normalización de Datos

Para estructurar y limpiar los datos, se implementaron varios pasos utilizando Excel:

Estandarización y Separación de Campos

- **Separación de "Nombre" y "Apellido":** Dado que estos datos estaban combinados en un solo campo, utilizamos herramientas de Excel como “Texto en columnas” para dividirlos y asignarlos a columnas individuales.
- **Estandarización de "Roles" y "Grupos":** Se corrigieron los formatos de estos campos para garantizar la uniformidad en el archivo:
- **Estandarización de "Localidades" y "Materias":** Se eliminaron abreviaciones y formatos inconsistentes en nombres de localidades y materias. Para esto:
 - Usamos funciones de Excel como "Buscar y Reemplazar" (Ctrl + H) para identificar y corregir rápidamente los datos comunes.
 - Estandarizamos nombres completos de localidades y validamos la información.

División de la Columna "Descripción" en Subcampos

El campo "Descripción" contenía múltiples tipos de datos y se dividió en subcampos específicos.

Pasos de Limpieza y Organización Realizados (Data Preprocessing)

Normalización de Datos: Tras la separación de los datos, desarrollamos un código en Python para realizar una normalización automática. Este código identificó y estandarizó patrones comunes dentro de los datos divididos. Entre las acciones realizadas, se incluyeron:

- **Estandarización de términos** en la columna "Materias en curso" (por ejemplo, "Pack de 3", "Seminario", "Algoritmos 3").
- **Agrupación de términos en "Trabajo" y "Experiencia en BD"** para crear categorías homogéneas.
- **Uniformidad en las localidades** eliminando variaciones y abreviaturas.
- **Estandarización de Hobbies y Música/Series** según patrones comunes.

También decidimos sacar la columna series, ya que nos parecía irrelevante porque no estaba dentro del todo completa, como así también a los docentes de la tabla ya que en el enunciado del trabajo solo se nos pide ingresar los datos de los alumnos inscriptos en la universidad. Además de esto incorporaremos información ficticia para enriquecer los datos y permitir así un análisis más significativo.

Propuestas para Datos Incompletos o Faltantes

- **Campos Incompletos en Descripción:** Proponemos realizar una revisión detallada de los registros que carecen de información clave, como hobbies o ubicación. Se podrían asignar valores predeterminados en campos no críticos, para mantener la integridad de los datos.
- **Automatización en Casos de Gran Volumen de Datos:** Si tuviéramos un volumen de datos mayor, sería conveniente emplear técnicas automatizadas de limpieza de datos, como algoritmos de normalización, detección de patrones y modelos de procesamiento de lenguaje natural (NLP) para identificar y corregir inconsistencias. Estas herramientas serían adecuadas para gestionar grandes volúmenes de datos sin requerir una intervención manual exhaustiva.

Parte 2: Selección y Justificación del Motor de Base de Datos

Selección y Justificación del Motor de Base de Datos

Elegimos PostgreSQL como motor de base de datos relacional para este trabajo debido a sus capacidades avanzadas para gestionar datos estructurados y consultas complejas. Permite realizar consultas detalladas con SQL estándar y asegura que los datos estén organizados de manera precisa mediante claves foráneas y restricciones de integridad. Además, ofrece transacciones ACID y control de versiones, lo que protege la coherencia y precisión de los datos.

Está diseñado para optimizar el rendimiento en grandes volúmenes de datos mediante el uso de índices, que aceleran las consultas, y particiones que dividen las tablas en partes más pequeñas y manejables, agilizando el acceso a los datos. También es posible escalar su capacidad con servidores más potentes (escalabilidad vertical) y admitir la replicación de datos, permitiendo así distribuir la carga de trabajo entre varios servidores y mejorar la disponibilidad del sistema.

Para su implementación en producción, PostgreSQL requiere una infraestructura adecuada, lo cual puede generar costos adicionales, especialmente si se necesita alta disponibilidad o rendimiento. Aunque permite la replicación y el particionamiento, presenta ciertas limitaciones en la escalabilidad horizontal, ya que no está tan optimizado como algunas bases de datos no relacionales para dividir la carga entre Múltiples servidores, lo cual puede ser menos adecuado en proyectos donde el volumen de datos crece rápidamente.

Comparación con una Alternativa No Relacional: MongoDB

MongoDB es una base de datos no relacional basada en documentos, en la que cada documento almacena un registro independiente. Esta estructura permite gestionar datos sin necesidad de un esquema rígido, otorgando flexibilidad en el almacenamiento.

Entre sus ventajas, MongoDB no requiere un esquema fijo, lo que permite añadir o modificar campos con facilidad. También ofrece una escalabilidad horizontal mediante fragmentación , conveniente si se cuentan con un volumen de datos distribuidos en múltiples servidores.

Sin embargo, MongoDB tiene limitaciones importantes en términos de integridad y complejidad de consultas. Carece de integridad relacional, ya que no mantiene relaciones directas entre datos, lo cual puede dificultar la gestión de integridad en aplicaciones que requieren datos estructurados. Además, no permite realizar uniones complejas como las bases de datos relacionales, lo que limita su uso en aplicaciones que necesitan consultas detalladas entre datos relacionados.

Comparación entre PostgreSQL y MongoDB

Aspecto	PostgreSQL	MongoDB
Estructura	Datos estructurados con integridad relacional	Flexibilidad en el esquema, sin integridad referencial.
Escalabilidad	Escalabilidad vertical con opciones limitadas de horizontalidad	Escalabilidad horizontal avanzada mediante fragmentación
Consultas complejas	Soporta uniones y SQL avanzado	Consultas complejas limitadas, sin uniones
Integridad	Asegurada con transacciones ACID y claves foráneas	No asegura consistencia entre entidades
Flexibilidad	Los cambios de esquema requieren modificaciones	Estructura flexible sin cambios de esquema.
Redundancia de datos	Baja redundancia gracias a la normalización, pero requiere uniones para combinar datos relacionados.	Alta redundancia debido a la desnormalización, optimizada para lecturas rápidas pero con mayor mantenimiento en actualizaciones.

Parte 3: Diseño de Modelo de Datos y Creación del Esquema

Diseño del Modelo Entidad-Relación

Un Diagrama Entidad-Relación (E-R) es una representación visual de los datos que vamos a guardar en nuestra base de datos y de cómo estos datos están conectados entre sí.

En nuestro proyecto, el Diagrama E-R incluye las siguientes **entidades y relaciones**:

Entidades

- **Alumno:** almacena información de cada estudiante.
- **Materia:** guarda las materias que cursan los alumnos.
- **Grupo:** representa los grupos de estudio o trabajo a los que pertenecen los alumnos.
- **Rol:** define los diferentes roles que puede tener un alumno dentro de un grupo.
- **Hobbie:** guarda los pasatiempos o hobbies de los alumnos.
- **Mascota:** registra las mascotas que tienen los alumnos.
- **Trabajo:** indica el empleo o puesto en el que trabaja un alumno.
- **Localidad:** almacena las ubicaciones o lugares donde viven los alumnos.
- **Música:** guarda los géneros musicales que escuchan los alumnos.

Relaciones

- **Alumno-Materia:** llamada **alumno_materia**, indica qué materias cursa cada alumno.
- **Alumno-Grupo-Rol:** llamada **alumno_grupo_rol**, muestra que un alumno puede pertenecer a un grupo y tener un rol en él.
- **Alumno-Hobbie:** llamada **alumno_hobbie**, indica los hobbies de cada alumno.
- **Alumno-Música:** llamada **alumno_musica**, representa los géneros de música que escucha cada alumno.
- **Alumno-Mascota:** llamada **mascota**, indica qué mascotas tienen los alumnos. Relaciona las entidades Alumno y Mascota mediante la clave foránea **id_alumno**.
- **Alumno-Localidad:** en la tabla **alumno**, donde la clave foránea **id_localidad** conecta al alumno con la localidad.
- **Alumno-Trabajo:** en la tabla **alumno**, la clave foránea **id_trabajo** conecta al alumno con el trabajo.

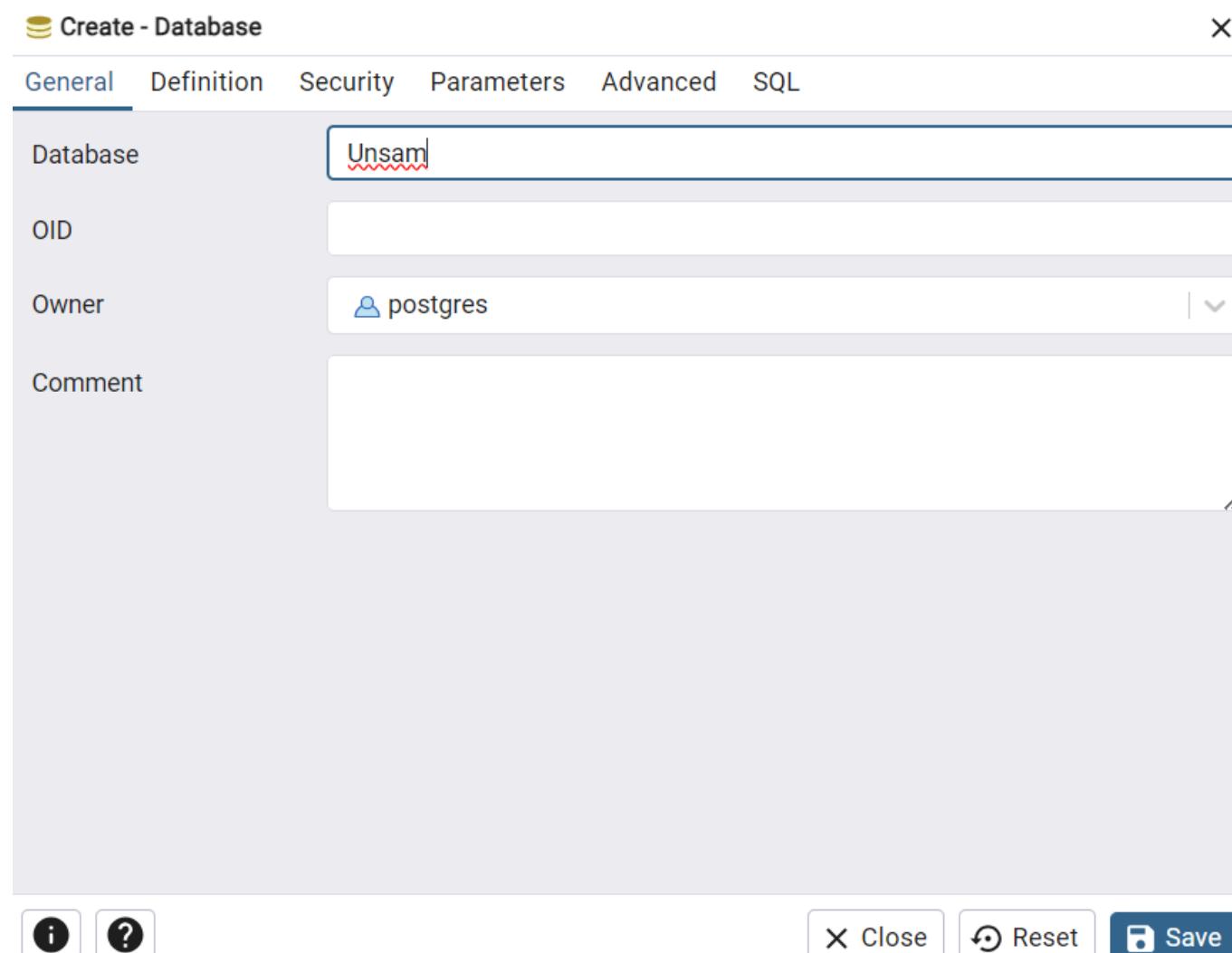
Hemos elegido estas entidades y relaciones siguiendo un proceso llamado normalización, con el objetivo de organizar los datos, sin duplicaciones ni errores. Con esto buscamos llegar a la tercera forma normal, lo cual significa que:

1. Cada tipo de información tiene su propia tabla.
2. Las tablas se conectan solo cuando es necesario.
3. Las relaciones de muchos a muchos usando tablas intermedias.

Implementación del Esquema en SQL

Creación de la Base de Datos Inicial

Antes de proceder con la creación de las tablas y estructuras de datos, es necesario establecer la base de datos principal en PostgreSQL, que será utilizada como el contenedor para toda la información del proyecto. En este caso, hemos creado una base de datos llamada "**Unsam**", asignada al usuario `postgres` como propietario.



Create - Database

General Definition Security Parameters Advanced SQL

Database: `Unsam`

OID:

Owner: `postgres`

Comment:

i ? × Close ↻ Reset 💾 Save

Creación de tablas

Usamos el comando CREATE TABLE para definir cada tabla con sus respectivas columnas y claves primarias.

```
CREATE TABLE alumno (
    id_alumno SERIAL PRIMARY KEY,
    nombre VARCHAR(50),
    apellido VARCHAR(50) NOT NULL,
    dni VARCHAR(15),
    email VARCHAR(100),
    experiencia_bd BOOLEAN,
    experiencia_bd_relacional BOOLEAN
);

CREATE TABLE grupo (
    id_grupo SERIAL PRIMARY KEY,
    nombre_grupo VARCHAR(100) NOT NULL
);

CREATE TABLE rol (
    id_rol SERIAL PRIMARY KEY,
    nombre_rol VARCHAR(100) NOT NULL
);

CREATE TABLE mascota (
    id_mascota SERIAL PRIMARY KEY,
    nombre_mascota VARCHAR(100) NOT NULL
);

CREATE TABLE hobbie (
    id_hobbie SERIAL PRIMARY KEY,
    nombre_hobbie VARCHAR(100) NOT NULL UNIQUE
);

CREATE TABLE trabajo (
    id_trabajo SERIAL PRIMARY KEY,
    descripcion VARCHAR(100),
    nombre_puesto VARCHAR(50)
);
```

```
CREATE TABLE musica (
    id_musica SERIAL PRIMARY KEY,
    nombre_musica VARCHAR(100) NOT NULL
);

CREATE TABLE materia (
    id_materia SERIAL PRIMARY KEY,
    nombre_materia VARCHAR(100) NOT NULL
);

CREATE TABLE localidad (
    id_localidad SERIAL PRIMARY KEY,
    nombre_localidad VARCHAR(50) NOT NULL
);
```

Introducción a la creación de las tablas relacionales y claves FK

Para continuar con nuestro proyecto, hemos implementado el manejo de las relaciones de muchos a muchos entre entidades, usaremos tablas intermedias.

Las claves foráneas fueron declaradas explícitamente usando REFERENCES, para asegurar la dependencia.

Además, utilizamos la cláusula ON DELETE SET NULL para evitar la pérdida de información que podría ser necesaria más adelante, por ejemplo, para estadísticas o análisis históricos. En lugar de eliminar completamente los datos relacionados, los dejamos como NULL para mantener un registro parcial de las asociaciones, incluso por ejemplo si un alumno, grupo o rol se elimina, entre otros.

Crear las Tablas de Relaciones y determinación de las claves foránea

Para reflejar las relaciones entre los alumnos y las entidades de localidad y trabajo, se añadieron las columnas id_localidad e id_trabajo a la tabla alumno, estableciendo claves foráneas hacia las tablas localidad y trabajo, respectivamente.

```
ALTER TABLE alumno
ADD COLUMN id_trabajo INT,
ADD CONSTRAINT fk_trabajo FOREIGN KEY (id_trabajo) REFERENCES trabajo(id_trabajo);

ALTER TABLE alumno
ADD COLUMN id_localidad INT,
ADD CONSTRAINT fk_localidad FOREIGN KEY (id_localidad) REFERENCES localidad(id_localidad);
```

Agregamos la columna id_alumno a la tabla mascota para identificar al dueño de cada mascota. Se aplicó la relación entre mascota y alumno.

```
ALTER TABLE mascota
ADD COLUMN id_alumno INT,
ADD CONSTRAINT fk_mascota_alumno FOREIGN KEY (id_alumno) REFERENCES alumno(id_alumno) ON DELETE SET NULL;
```

Para manejar la relación entre alumnos y las materias que cursan, se creó esta tabla intermedia. Las claves foráneas id_alumno y id_materia están referenciadas a las tablas alumno y materia, respectivamente.

```
CREATE TABLE alumno_materia (
    id_alumno INT,
    id_materia INT,
    PRIMARY KEY (id_alumno, id_materia),
    CONSTRAINT fk_alumno FOREIGN KEY (id_alumno) REFERENCES alumno(id_alumno) ON DELETE SET NULL,
    CONSTRAINT fk_materia FOREIGN KEY (id_materia) REFERENCES materia(id_materia) ON DELETE SET NULL
);
```

Generamos la tabla alumno_grupo_id e incluimos las claves foráneas id_alumno, id_grupo y id_rol, asegurando la integridad referencial. Se utiliza **ON DELETE SET NULL** para evitar que la eliminación de un grupo o rol cause la eliminación encadenada de los alumnos.

```
CREATE TABLE alumno_grupo_rol (
    id_alumno INT,
    id_grupo INT,
    id_rol INT,
    PRIMARY KEY (id_alumno, id_grupo),
    CONSTRAINT fk_alumno_gr FOREIGN KEY (id_alumno) REFERENCES alumno(id_alumno) ON DELETE SET NULL,
    CONSTRAINT fk_grupo FOREIGN KEY (id_grupo) REFERENCES grupo(id_grupo) ON DELETE SET NULL,
    CONSTRAINT fk_rol FOREIGN KEY (id_rol) REFERENCES rol(id_rol) ON DELETE SET NULL
);
```

También generamos la tabla alumno_hobbie para identificar los hobbies asociados a cada alumno.

```
CREATE TABLE alumno_hobbie (
    id_alumno INT,
    id_hobbie INT,
    PRIMARY KEY (id_alumno, id_hobbie),
    CONSTRAINT fk_alumno FOREIGN KEY (id_alumno) REFERENCES alumno(id_alumno) ON DELETE SET NULL,
    CONSTRAINT fk_hobbie FOREIGN KEY (id_hobbie) REFERENCES hobbie(id_hobbie) ON DELETE SET NULL
);
```

La tabla alumno_musica, esta tabla intermedia conecta a los alumnos con los géneros musicales que escuchan.

```
CREATE TABLE alumno_musica (
    id_alumno INT,
    id_musica INT,
    PRIMARY KEY (id_alumno, id_musica),
    CONSTRAINT fk_alumno FOREIGN KEY (id_alumno) REFERENCES alumno(id_alumno) ON DELETE SET NULL,
    CONSTRAINT fk_musica FOREIGN KEY (id_musica) REFERENCES musica(id_musica) ON DELETE SET NULL
);
```

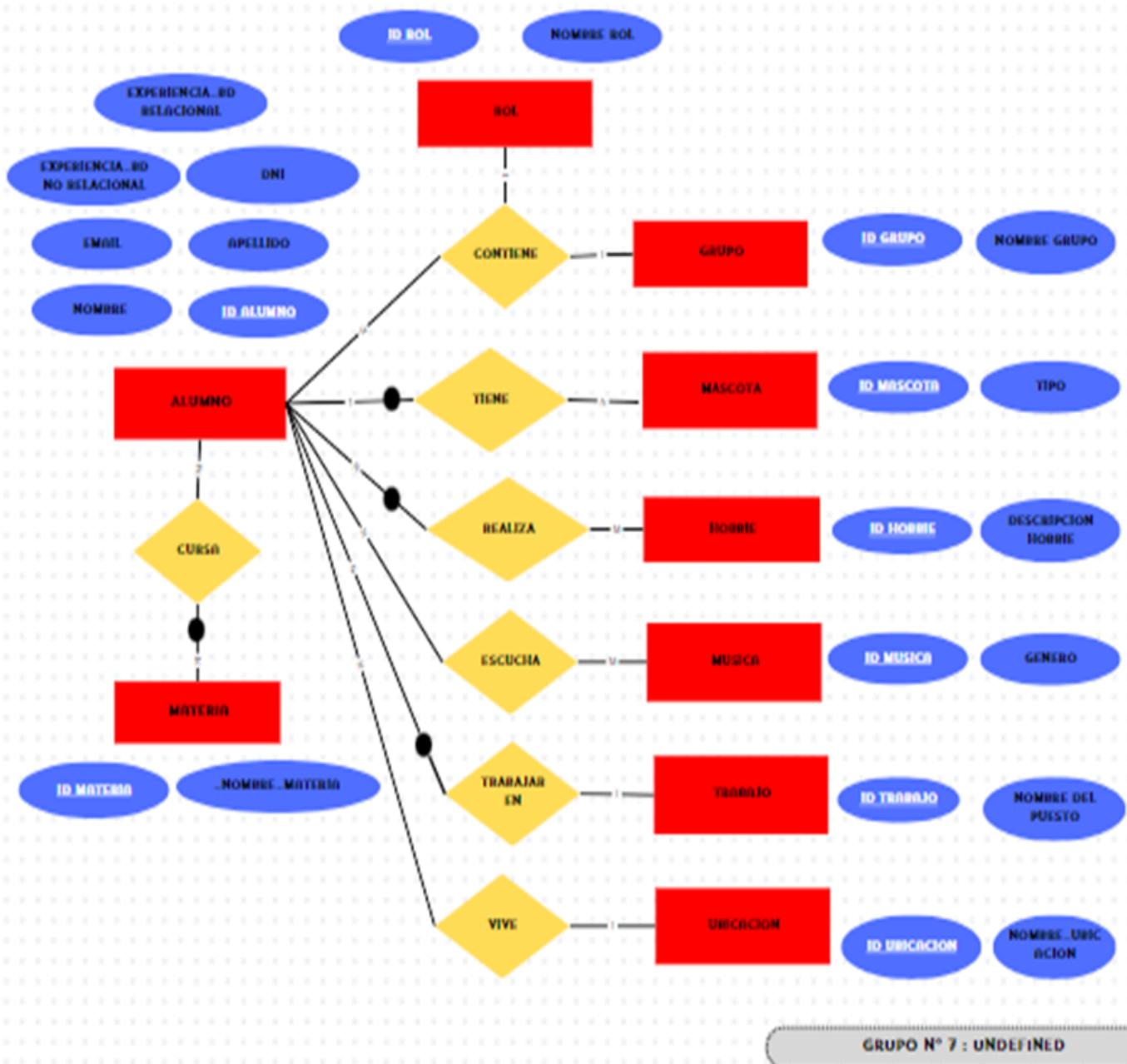
Verificar las Tablas

En psql, ejecutamos el comando \d tp.* para ver la creación de cada tabla y sus relaciones

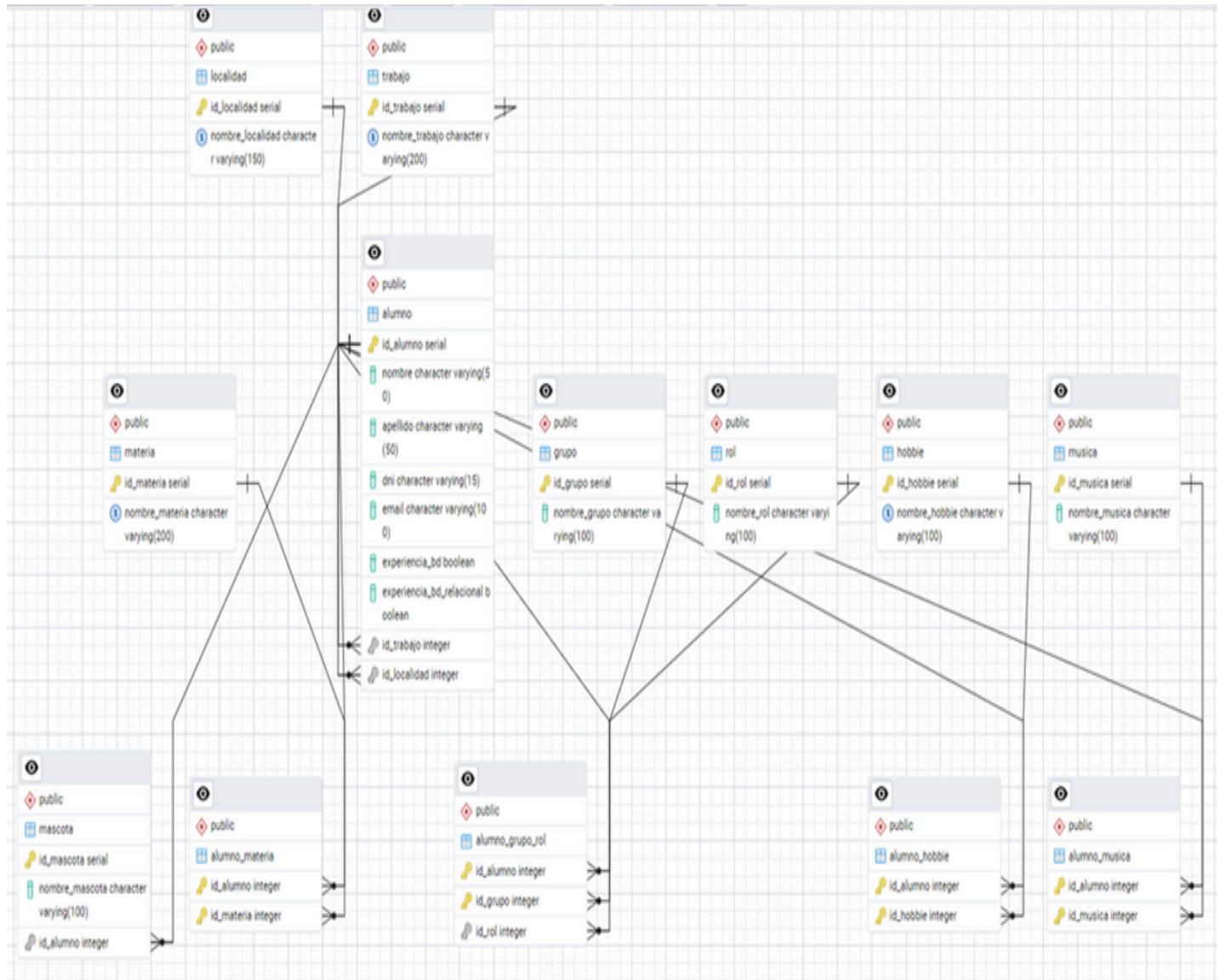
Listado de relaciones			
Esquema	Nombre	Tipo	Dueño
tp	alumno	tabla	postgres
tp	alumno_grupo_rol	tabla	postgres
tp	alumno_hobbie	tabla	postgres
tp	alumno_materia	tabla	postgres
tp	alumno_musica	tabla	postgres
tp	grupo	tabla	postgres
tp	hobbie	tabla	postgres
tp	localidad	tabla	postgres
tp	mascota	tabla	postgres
tp	materia	tabla	postgres
tp	musica	tabla	postgres
tp	rol	tabla	postgres
tp	trabajo	tabla	postgres
(13 filas)			

Diagrama Entidad-Relación (E-R)

El siguiente DER fue diseñado en el canva para representar las entidades y relaciones según nuestras decisiones de diseño, basadas en los requisitos del sistema y en las mejores prácticas de normalización.



PostgreSQL tiene la ventaja de que después de las creaciones de las tablas podemos crear el DER. Este diagrama muestra cómo se implementa realmente el modelo de datos en el sistema y refleja los nombres y relaciones exactas de las tablas en la base de datos.



Modelo lógico y restricciones

Modelo Lógico Relacional

alumno (**id_alumno**, nombre, apellido, email, experiencia_bd_relacional,

experiencia_bd_no_relacional, DNI, **id_trabajo**, **id_localidad**)

materia (**id_materia**, nombre_materia)

CURSA(**id_materia**, **id_alumno**)

rol(**id_rol**, nombre_rol)

grupo(**id_grupo**, nombre_grupo)

Contiene(**id_alumno**, **id_grupo**, **id_rol**)

mascota(**id_mascota**, tipo, **id_alumno**)

hobbie(**id_hobbie**, descripcion_hobbie)

REALIZA(**id_hobbie**, **id_alumno**)

musica(**id_musica**, Genero)

Escucha (**id_musica**, **Id_alumno**)

trabajo (**id_trabajo**, nombre_del_puesto)

localidad(**ID_localidad**,NOMBRE_localidad)

Restricciones

alumno.**id_alumno** puede no estar en mascota.**id_alumno**

alumno.**id_trabajo** puede ser nulo

ALUMNO_ID.**ALUMNO** PUEDE NO ESTAR EN REALIZA.ID_ALUMNO

materia.**id_materia** puede no estar en cursa.**id_materia**

Las claves primarias están resaltadas en rojo para facilitar su identificación, mientras que las claves foráneas están marcadas en celeste, destacando las relaciones entre las tablas.

Parte 4: Carga y Manipulación de Datos

Crear la tabla provisoria

Antes de cargar los datos a PostgreSQL, transformamos el archivo original en formato Excel a CSV, ya que este formato es más adecuado para trabajar en el entorno de PostgreSQL. Posteriormente, creamos una tabla provisoria que coincide con la estructura del archivo CSV, permitiendo así una carga eficiente y organizada de los datos.

```
CREATE TABLE info_generica (
    dni INT,
    apellido VARCHAR(100),
    nombre VARCHAR(100),
    email VARCHAR(150),
    grupo VARCHAR(100),
    rol VARCHAR(100),
    materia VARCHAR(200),
    trabajo VARCHAR(200),
    experiencia_bd TEXT,
    experiencia_bd_relacional TEXT,
    localidad VARCHAR(100),
    mascotas VARCHAR(100),
    musica VARCHAR(100),
    hobbie VARCHAR(100)
);
```

Luego utilizamos el comando COPY para cargar los datos del archivo CSV a la tabla creada.

```
COPY info_generica
FROM 'C:/Program Files/PostgreSQL/17/Unsam-tp.csv'
DELIMITER ';'
CSV HEADER;
```

Carga de los datos a las respectivas tablas

Vamos a proceder a cargar los datos a las tablas que ya creamos, comenzamos con alumnos.

Como se detectaron varios valores NULL en la columna dni en la tabla info_generica, se decidió eliminar la restricción de unicidad (UNIQUE) de la columna dni en la tabla alumno. Adicionalmente, se observaron registros donde el campo nombre estaba vacío o con valores NULL, por lo que también se eliminó la restricción NOT NULL en esta columna. Si bien estas decisiones no son lo más adecuado desde el punto de vista de diseño y calidad de datos, se realizaron de esta manera con el objetivo de poder cargar los datos de forma completa y evitar errores que impedían el proceso de carga inicial.

Las columnas experiencia_bd y experiencia_bd_relacional, se convirtieron a valores booleanos (TRUE o FALSE) usando la cláusula CASE.

```
INSERT INTO alumno (nombre, apellido, dni, email, experiencia_bd, experiencia_bd_relacional)
SELECT
    nombre,
    apellido,
    dni,
    email,
    CASE WHEN experiencia_bd = 'Con experiencia en BD' THEN TRUE ELSE FALSE END AS experiencia_bd,
    CASE WHEN experiencia_bd_relacional = 'Con experiencia en BD' THEN TRUE ELSE FALSE END AS experiencia_bd_relacional
FROM info_generica;

ALTER TABLE alumno
ADD COLUMN id_trabajo INT,
ADD CONSTRAINT fk_trabajo FOREIGN KEY (id_trabajo) REFERENCES trabajo(id_trabajo);

UPDATE alumno
SET id_trabajo = subquery.id_trabajo
FROM (
    SELECT
        a.id_alumno AS id_alumno,
        t.id_trabajo AS id_trabajo
    FROM alumno a
    JOIN info_generica ig ON CAST(ig.dni AS VARCHAR) = a.dni
    JOIN trabajo t ON ig.trabajo = t.nombre_trabajo
) subquery
WHERE alumno.id_alumno = subquery.id_alumno;
```

Tratamos de eliminar si existen duplicados y que quede una única combinación.

```
DELETE FROM alumno
WHERE ctid NOT IN (
    SELECT MIN(ctid)
    FROM alumno
    GROUP BY id_alumno, id_trabajo
);
```

	Id_alum	nombre	apellido	dni	email	experiencia_bd	experiencia_b	id_trabajo	id_localidad
	[PK] integer	character varying (50)	character varying (50)	character varying (15)	character varying (100)	boolean	boolean	integer	integer
32	5	Rodrigo Nicolas	Pavon Gomez	40663606	rodrigopavongomez@gmail.com	false	true	6	15
33	7	[null]	Dragonetti	38601662	mdragonetti@estudiantes.unsam.edu.ar	true	true	6	12
34	8	Juan Ignacio	Caceffo	44547586	jicaceffo@estudiantes.unsam.edu.ar	true	true	2	17
35	9	[null]	Gibelli	37984582	julilibelli@gmail.com	true	true	6	3
36	79	Matias	Caballero	35093145	msebacaballero@gmail.com	false	true	16	16
37	80	David	Pazos	36594617	davidgpazos@gmail.com	false	false	11	3
38	82	Fabrizio	Signorello	43441575	fsignorello@estudiantes.undam.edu.ar	true	true	6	20
39	84	Andrs Elias	Simonini	42997600	aesimonini@estudiantes.unsam.edu.ar	false	false	14	12
40	86	Emiliano	Ferretti	35205248	emifieferretti@gmail.com	true	true	7	14
41	88	[null]	Perez	41555134	amperez@estudiantes.unsam.edu.ar	false	false	12	20
42	91	Delfina	Borrelli	41548103	dborrelli@estudiantes.unsam.edu.ar	false	false	12	15
43	94	Pedro	Geraghty	42101048	pedrogeraghty82@gmail.com	false	false	3	6
44	96	Diego	Lentz	36791436	diqoolentz@gmail.com	false	true	5	1

En el caso de las tablas grupo, rol, trabajo , localidad se ejecutó lo siguiente:

```
INSERT INTO grupo (nombre_grupo)
SELECT DISTINCT grupo
FROM info_generica
WHERE grupo IS NOT NULL;
```

	id_grupo [PK] integer	nombre_grupo character varying (50)
1	1	1 - Datamasters
2	2	2 - Nullpointer
3	3	3 - Enrutados
4	4	4 - Mandarina
5	5	5 - Mcteam
6	6	6 - Okupas
7	7	7 - Undefined
8	8	8 - Droptable
9	9	9 - Dreamteam

```
INSERT INTO rol (nombre_rol)
SELECT DISTINCT rol
FROM info_generica
WHERE rol IS NOT NULL;
```

	id_rol [PK] integer	nombre_rol character varying (100)
1	1	Team Leader
2	2	Representante
3	3	Organizador
4	4	Supervisor
5	5	Lider Técnico

```
INSERT INTO trabajo (nombre_trabajo)
SELECT DISTINCT trabajo
FROM info_generica
WHERE trabajo IS NOT NULL;
```

	id_trabajo [PK] integer	nombre_trabajo character varying (200)
1	1	Comercio
2	2	Ingeniero en Datos
3	3	Oficinista
4	4	Soporte IT
5	5	Administrativo
6	6	Desarrollador
7	7	Vendedor
8	8	Independiente
9	9	Programador
10	10	Sin especificar
11	11	Seguridad
12	12	No trabaja actualmente
13	13	Repartidor
14	14	IA
15	15	Front end
16	16	Mecanico

```
INSERT INTO localidad (nombre_localidad)
SELECT DISTINCT localidad
FROM info_generica
WHERE localidad IS NOT NULL;
```

	id_localidad [PK] integer	nombre_localidad character varying (150)
1	1	Chilavert
2	2	Saavedra
3	3	Belgrano
4	4	Palermo
5	5	El Palomar
6	6	Villa Devoto
7	7	Jose Leonsuarez
8	8	Villa Urquiza
9	9	Boedo
10	10	Ciudad Jardin
11	11	San Andres
12	12	San Martín
13	13	Santos Lugares
14	14	Sin especificar
15	15	Laferriere
16	16	General Pacheco
17	17	Ciudad Autónoma de Buenos Aires
18	18	Villa Bosch
19	19	Villa Ballester
20	20	Escobar

Asociamos cada alumno con una localidad, transformando (nombre_localidad) en su correspondiente clave foránea (id_localidad).

```

UPDATE alumno
SET id_localidad = subquery.id_localidad
FROM (
    SELECT
        a.id_alumno AS id_alumno,
        l.id_localidad AS id_localidad
    FROM alumno a
    JOIN info_generica ig ON CAST(ig.dni AS VARCHAR) = a.dni
    JOIN localidad l ON ig.localidad = l.nombre_localidad
) subquery
WHERE alumno.id_alumno = subquery.id_alumno;
  
```

Identificamos a los alumnos con sus mascotas, permitiendo que aquellos sin una mascota tengan el valor 'Sin mascotas'.

```

INSERT INTO mascota (nombre_mascota, id_alumno)
SELECT
    COALESCE(ig.mascotas, 'Sin mascotas') AS nombre_mascota,
    a.id_alumno
FROM alumno a
LEFT JOIN info_generica ig ON CAST(ig.dni AS VARCHAR) = a.dni;
  
```

Para sacar los id_duplicados

```

DELETE FROM mascota
WHERE ctid NOT IN (
    SELECT MIN(ctid)
    FROM mascota
    GROUP BY nombre_mascota, id_alumno
);
  
```

	id_mascota [PK] integer	nombre_mascota character varying (100)	id_alumno integer
1	1	Sin mascotas	1
2	2	Perro(s)	2
3	4	Perro(s)	3
4	6	Perro(s)	4
5	7	Perro(s)	5
6	9	Perro(s)	6
7	11	Perro(s)	7
8	12	Perro(s)	8
9	13	Sin mascotas	9
10	15	Sin mascotas	10
11	17	Sin mascotas	11
12	18	Sin mascotas	12
13	20	Sin mascotas	13
14	22	Gato(s)	14

Incorporamos los datos sobre géneros musicales en la tabla de música, asegurando que no haya valores duplicados (DISTINCT). Aplicamos el mismo procedimiento con la tabla hobbie y la tabla materia.

```
INSERT INTO musica (nombre_musica)
SELECT DISTINCT
    COALESCE(musica, 'Sin género musical') AS nombre_musica
FROM info_generica;
```

```
INSERT INTO hobbie (nombre_hobbie)
SELECT DISTINCT
    COALESCE(hobbie, 'Sin hobbie') AS nombre_hobbie
FROM info_generica;
```

```
INSERT INTO materia (nombre_materia)
SELECT DISTINCT materia
FROM info_generica
WHERE materia IS NOT NULL;
```

Eliminamos inconsistencias en la nomenclatura de las materias, para facilitar el análisis y consultas posteriores.

```

UPDATE info_generica
SET materia = 'Bases de Datos'
WHERE materia ILIKE '%Bases de Datos%';

UPDATE info_generica
SET materia = 'Algoritmos 3'
WHERE materia ILIKE '%Algoritmos 3%';

UPDATE info_generica
SET materia = 'Seminario de Programación Concurrente'
WHERE materia ILIKE '%Seminario%'
OR materia ILIKE '%Concurrente%';

UPDATE info_generica
SET materia = 'CASO'
WHERE materia ILIKE '%CASO%';
  
```

	id_materia [PK] integer	nombre_materia character varying (100)
1	2	Algoritmos 3
2	3	CASO
3	4	Seminario de Programación Concurrente
4	1	Bases de Datos

Ahora insertamos los datos en las tablas relacionales:

```

INSERT INTO alumno_materia (id_alumno, id_materia)
SELECT DISTINCT
    a.id_alumno,
    m.id_materia
FROM info_generica ig
JOIN alumno a ON CAST(ig.dni AS VARCHAR) = a.dni
JOIN materia m ON ig.materia = m.nombre_materia;

INSERT INTO alumno_grupo_rol (id_alumno, id_grupo, id_rol)
SELECT DISTINCT
    a.id_alumno,
    g.id_grupo,
    r.id_rol
FROM info_generica ig
JOIN alumno a ON CAST(ig.dni AS VARCHAR) = a.dni
JOIN grupo g ON ig.grupo = g.nombre_grupo
JOIN rol r ON ig.rol = r.nombre_rol;
  
```

```
INSERT INTO alumno_hobbie (id_alumno, id_hobbie)
SELECT DISTINCT
    a.id_alumno,
    h.id_hobbie
FROM info_generica ig
JOIN alumno a ON CAST(ig.dni AS VARCHAR) = a.dni
JOIN hobbie h ON ig.hobbie = h.nombre_hobbie;

INSERT INTO alumno_musica (id_alumno, id_musica)
SELECT DISTINCT
    a.id_alumno,
    m.id_musica
FROM info_generica ig
JOIN alumno a ON CAST(ig.dni AS VARCHAR) = a.dni
JOIN musica m ON ig.musica = m.nombre_musica;
```

Consultas Básicas y Reportes

Para realizar esta parte del trabajo práctico, utilizaremos:

WITH (Common Table Expressions, CTEs) para estructurar subconsultas intermedias y facilitar su reutilización.

Usamos diferentes tipos de **JOIN**, para relacionar las tablas y obtener datos precisos y consistentes.

La función **ROW_NUMBER()**, que nos ayudará a clasificar los resultados según criterios específicos, como el número de inscripciones o el conteo de hobbies por localidad. Esto nos permitirá destacar los valores más representativos, como la materia más popular en cada grupo o el hobby más común en cada localidad.

Las cláusulas **WHERE** y funciones de ventana como **COUNT()** y **RANK()**, aseguran la precisión de los resultados.

La herramienta **PARTITION BY** nos va a permitir dividir el conjunto de datos en particiones lógicas basadas en valores de una o más columnas.

Con **CASE**, vamos a evaluar condiciones y retornar diferentes valores dependiendo de si esas condiciones se cumplen.

RANK asigna un rango a cada fila dentro de una partición definida, será combinado con **PARTITION BY** ya que nos facilitará hacer el análisis específico por secciones del conjunto de datos.

Punto A)

En esta consulta, el objetivo es identificar los hobbies más comunes en cada localidad y el número total de alumnos que lo realizan. Utilizamos una “**Expresión Común de Tabla**” (**CTE**) la llamamos **hobbie_ranking**, para organizar los datos necesarios para el análisis de una manera estructurada y reutilizable.

Dentro de la **hobbie_ranking**, los datos están agrupados por localidad y hobby utilizando la cláusula **GROUP BY**. También usamos la función de agregación **COUNT**, para contar el número de alumnos (id_alumnos) asociados a cada combinación de localidad y hobby.

Además, se hace uso de la función **RANK()**, para calcular un rango de cada hobby dentro de una localidad, ordenando los resultados por el número total de alumnos en orden descendente. La cláusula **PARTITION BY** que utilizamos dentro de la función nos **ASEGURAMOS** que el cálculo de los rangos se realice de forma independiente para cada localidad. Al asignar el rango 1 al hobby con más alumnos en cada localidad, se identifica fácilmente el hobby más popular en cada área.

Se filtran los datos generados en la CTE mediante la cláusula **WHERE rank_hobby = 1**, para incluir únicamente los hobbies con el rango 1, es decir, los más practicados. Luego, los resultados se ordenan en orden descendente utilizando **ORDER BY** basado en la cantidad total de alumnos, destacando las localidades con los hobbies más populares en términos de participación. Por último, se utiliza la cláusula **LIMIT 10** para limitar la salida a las 10 localidades con mayor participación.

```

WITH hobbie_ranking AS (
  SELECT
    l.nombre_localidad AS Localidad,
    h.nombre_hobbie AS "Hobby más Común",
    COUNT(ah.id_alumno) AS "Total Alumnos",
    RANK() OVER (PARTITION BY l.nombre_localidad ORDER BY COUNT(ah.id_alumno) DESC) AS rank_hobbie
  FROM alumno a
  JOIN localidad l ON a.id_localidad = l.id_localidad
  JOIN alumno_hobbie ah ON a.id_alumno = ah.id_alumno
  JOIN hobbie h ON ah.id_hobbie = h.id_hobbie
  GROUP BY l.nombre_localidad, h.nombre_hobbie
)
SELECT Localidad, "Hobby más Común", "Total Alumnos"
FROM hobbie_ranking
WHERE rank_hobbie = 1
ORDER BY "Total Alumnos" DESC
LIMIT 10;
  
```

	localidad character varying (150)	Hobby más Común character varying (100)	Total Alumnos bigint
1	Sin especificar	Videojuegos	7
2	Villa Ballester	Deportes	7
3	San Martín	Deportes	6
4	Villa Bosch	Videojuegos	4
5	Ciudad Jardín	Deportes	3
6	Ciudad Jardín	Videojuegos	3
7	Santos Lugares	Sin especificar	3
8	Escobar	Deportes	3
9	Ciudad Autónoma de Buenos Aires	Sin especificar	3
10	Escobar	Lectura	3

Punto B)

Generamos un reporte que muestra la información clave sobre los alumnos, incluyendo su nombre, apellido, la cantidad de materias en curso y su experiencia previa en bases de datos relacionales y no relacionales.

Primero seleccionamos los datos necesarios de la tabla **alumno**, incluyendo las columnas nombre y apellido para identificar a cada estudiante de manera clara. Además, utilizamos la relación entre las tablas alumno y materia a través de la tabla intermedia **alumno_materia** para calcular la cantidad de materias en las que está inscrito cada alumno. Este cálculo se realizó con la función de agregación **COUNT**, que permite contar las filas correspondientes a cada alumno, y agrupamos usando la cláusula **GROUP BY**.

Habíamos determinado con valores boléanos la experiencia previa en bases de datos, tanto relacionales como no relacionales. Pero para determinar que en vez de que nos muestre true o false, decidimos que nos muestre si o no para poder hacer esto empleamos la expresiones condicionales (**CASE**). Estas expresiones transforman los valores booleanos de las columnas **experiencia_bd_relacional** y **experiencia_bd** en respuestas legibles ("Sí" o "No").

Los datos se ordenan en orden descendente utilizando la cláusula **ORDER BY**, lo que prioriza a los alumnos con mayor cantidad de materias en curso. Se incluyó un límite de cinco registros mediante la cláusula **LIMIT 5**. Esto significa que el reporte muestra exclusivamente los cinco alumnos con mayor carga de materias. Interpretamos que nos piden obtener la información de los alumnos más activos en la UNSAM y facilita el análisis de su perfil en relación con su experiencia previa, el nivel técnico.

SELECT

```

a.nombre AS Alumno,
a.apellido AS Apellido,
COUNT(am.id_materia) AS "Materias en curso",
CASE WHEN a.experiencia_bd_relacional THEN 'Sí' ELSE 'No' END AS "Experiencia BD Relacional",
CASE WHEN a.experiencia_bd THEN 'Sí' ELSE 'No' END AS "Experiencia BD No Relacional"
FROM alumno a
LEFT JOIN alumno_materia am ON a.id_alumno = am.id_alumno
GROUP BY a.id_alumno, a.nombre, a.apellido, a.experiencia_bd_relacional, a.experiencia_bd
ORDER BY "Materias en curso" DESC
LIMIT 5;
  
```

	alumno character varying (50) 	apellido character varying (50) 	Materias en curso bigint 	Experiencia BD Relacional text 	Experiencia BD No Relacional text 
1	Tom s	Neiro	3	Sí	No
2	Cristian	Lomas	3	Sí	No
3	Lautaro	Cuellar	3	No	No
4	Federico	Virgilio	3	Sí	No
5	Lautaro	Cuellar	3	No	No

Punto C)

Para evitar posibles sesgos en los resultados, optamos por excluir la materia "Bases de Datos", ya que se imparte como parte central de este curso y todos los estudiantes la cursan, lo que podría distorsionar las interpretaciones estadísticas.

A través de esta exclusión, buscamos centrarnos en las materias adicionales que representan las preferencias específicas de cada grupo, permitiendo un análisis más objetivo y significativo. Ordenamos los resultados por el número de alumnos inscritos, limitando la visualización a los cinco registros más representativos, en orden descendente, para destacar las materias con mayor participación.

```

WITH materia_inscripciones AS (
  SELECT
    g.nombre_grupo AS grupo,
    m.nombre_materia AS materia,
    COUNT(am.id_alumno) AS alumnos_inscritos,
    COUNT(am.id_alumno) * 100.0 / SUM(COUNT(am.id_alumno)) OVER (PARTITION BY g.id_grupo) AS porcentaje_grupo
  FROM
    alumno_materia am
  INNER JOIN
    materia m ON am.id_materia = m.id_materia -- Relación con 'materia'
  INNER JOIN
    alumno_grupo_rol agr ON am.id_alumno = agr.id_alumno
  INNER JOIN
    grupo g ON agr.id_grupo = g.id_grupo
  WHERE
    m.nombre_materia != 'Bases de Datos' -- Excluir "Bases de Datos"
  GROUP BY
    g.id_grupo, g.nombre_grupo, m.nombre_materia
),

materia_mas_popular AS (
  SELECT
    grupo,
    materia,
    alumnos_inscritos,
    porcentaje_grupo,
    ROW_NUMBER() OVER (PARTITION BY grupo ORDER BY alumnos_inscritos DESC) AS rank
  FROM
    materia_inscripciones
)
SELECT
  grupo,
  materia,
  alumnos_inscritos,
  porcentaje_grupo
FROM
  materia_mas_popular
WHERE
  rank = 1
ORDER BY
  alumnos_inscritos DESC;
  
```

	grupo character varying (100)	materia character varying (100)	alumnos_inscritos bigint	porcentaje_grupo numeric
1	5 - MCTeam	Algoritmos 3	5	71.4285714285714286
2	6 - okupas	Seminario de Programación Concurrente	5	55.555555555555555556
3	9 - DreamTeam	Algoritmos 3	4	66.666666666666666667
4	3 - Enrutados	Algoritmos 3	4	50.00000000000000000000
5	7 - undefined	Seminario de Programación Concurrente	3	50.00000000000000000000

Punto D)

En este punto, el reporte que hicimos identifica a los alumnos con experiencia significativa en bases de datos relacionales, no relacionales o actividades tecnológicas, reflejando su rol dentro del grupo de estudio.

Para esto, hicimos el uso de una consulta SQL que combina información de varias tablas. Se trabajó con la tabla **alumno**, la cual almacena los datos personales y experiencia previa de los alumnos, junto con las tablas relacionadas **alumno_grupo_rol** y **rol**, que con esta tabla identificamos los roles asignados. Realizamos una **unión (LEFT JOIN)** entre estas tablas para asegurar que todos los alumnos con experiencia relevante fueran incluidos, incluso si no tenían un rol asignado explícitamente. Para los casos sin rol, se aplicó la función **COALESCE**, asignando el valor "Sin especificar" en la columna correspondiente.

Además, utilizamos **condicionales con CASE** para transformar los valores booleanos de las columnas **experiencia_bd_relacional** y **experiencia_bd** en respuestas legibles como "Sí" o "No". Esto hace que el reporte sea más comprensible para los usuarios finales.

La consulta se optimizó para priorizar a los alumnos con experiencia en bases de datos relacionales, seguidos de aquellos con experiencia en bases de datos no relacionales. Esto se logró utilizando un **ORDER BY** que ordena los resultados en base a estas experiencias. Por último, se incluyó una cláusula **LIMIT** para restringir el reporte a los cinco primeros resultados, asegurando que solo los alumnos más destacados fueran mostrados.

```
SELECT
    CONCAT(a.nombre, ' ', a.apellido) AS Alumno,
    CASE
        WHEN a.experiencia_bd_relacional THEN 'Sí'
        ELSE 'No'
    END AS "Experiencia BD Relacional",
    CASE
        WHEN a.experiencia_bd THEN 'Sí'
        ELSE 'No'
    END AS "Experiencia BD No Relacional",
    COALESCE(r.nombre_rol, 'Sin especificar') AS Rol
FROM alumno a
LEFT JOIN alumno_grupo_rol agr ON a.id_alumno = agr.id_alumno
LEFT JOIN rol r ON agr.id_rol = r.id_rol
WHERE a.experiencia_bd_relacional OR a.experiencia_bd
ORDER BY
    a.experiencia_bd_relacional DESC,
    a.experiencia_bd DESC,
    Rol ASC
LIMIT 5;
```

alumno text	Experiencia BD Relacional text	Experiencia BD No Relacional text	rol character varying
Juan Ignacio Caceffo	Sí	Sí	Lider Tecnico
Gibelli	Sí	Sí	Organizador
Jotallan Calvetti	Sí	Sí	Organizador
Mariel Nadine Kovinchich	Sí	Sí	Representante
Fabrizio Signorello	Sí	Sí	Representante

Consultas Avanzadas y Optimización

Punto A)

Para realizar este análisis, se combinaron datos de las tablas de alumnos, localidades y materias inscritas. En particular, utilizamos la tabla localidad para relacionar a cada alumno con su zona, y la tabla alumno_materia para determinar cuántas materias cursa cada alumno. Luego, calculamos el promedio de materias por localidad utilizando la función de agregación **AVG** aplicada sobre el conteo de materias inscritas (**COUNT**), redondeando los resultados a una cifra decimal para mayor claridad. Además, contamos el número total de alumnos únicos en cada localidad con la función **COUNT(DISTINCT)**, lo que asegura que cada alumno se considere solo una vez en el cálculo. Finalmente, los resultados se ordenaron en función del promedio de materias inscritas, destacando las localidades con mayor carga académica.

Los resultados que obtuvimos los ordenamos en función del promedio de materias inscritas , destacando aquellas localidades con mayor carga académica. Nos permite observar por ejemplo, en localidades como **San Martín, Villa Ballester**, los alumnos tienen un promedio más alto de materias inscritas , lo que podría indicar una mayor carga académica o mayor acceso a la oferta educativa. En contraste, en **Ciudad Autonoma de Buenos Aires**, el promedio de materias inscritas es significativamente menor, lo que podría reflejar limitaciones en la accesibilidad o una menor demanda académica en esa zona.

```

SELECT
    l.nombre_localidad AS localidad,
    ROUND(AVG(COUNT(am.id_materia)) OVER (PARTITION BY l.nombre_localidad), 1) AS "Promedio de materias",
    COUNT(DISTINCT a.id_alumno) AS "Total Alumnos"
FROM
    alumno a
JOIN
    localidad l ON a.id_localidad = l.id_localidad
JOIN
    alumno_materia am ON a.id_alumno = am.id_alumno
GROUP BY
    l.nombre_localidad
ORDER BY
    "Promedio de materias" DESC;
  
```

	localidad character varying (150)	Promedio de materias numeric	Total Alumnos bigint
1	Sin especificar	18.0	7
2	San Martín	18.0	8
3	Villa Ballester	13.0	5
4	Santos Lugares	7.0	3
5	Escobar	6.0	2
6	Belgrano	4.0	2
7	Villa Bosch	4.0	2
8	Villa Urquiza	3.0	1
9	Chilavert	3.0	1
10	Ciudad Autónoma de Buenos Aires	3.0	1

Punto B)

Desarrollamos una consulta avanzada para identificar el potencial mentor de cada alumno basándonos en sus intereses comunes y experiencia en bases de datos (relacional y no relacional).

Primero, en el **CTE** (intereses_comunes), identificamos las relaciones entre alumnos y mentores potenciales. Aquí vinculamos las tablas alumno, alumno_hobbie, y hobbie para encontrar intereses compartidos entre alumnos y mentores. Luego, con el uso de **CASE**, determinamos si los mentores tienen experiencia en bases de datos relacionales y no relacionales, lo que permite identificar qué mentores tienen más habilidades técnicas en ambas áreas.

Posteriormente, en el segundo **CTE** (mejor_mentor), agrupamos los datos por alumno y mentor, clasificando a los mentores según el número de intereses compartidos. Utilizamos la función **ROW_NUMBER()** para asignar un rango y seleccionar al mentor más relevante para cada alumno. Además, aplicamos un cálculo de prioridad mediante un **CASE** para establecer un orden en el que se prioricen los mentores con experiencia en ambas áreas (Sí en ambas categorías), seguidos por aquellos con experiencia parcial.

Aplicamos **DISTINCT** para eliminar filas duplicadas, asegurándonos de que cada mentor-alumno aparezca solo una vez en el resultado. La columna de prioridad, calculada previamente, se incluye explícitamente en la selección y en el ordenamiento para garantizar que los resultados están organizados según el criterio definido.

Limitamos los primeros 5 registros con **LIMIT**, mostrando a los alumnos con su mentor más relevante, el interés compartido que los conecta

```

WITH intereses_comunes AS (
  SELECT
    a.id_alumno AS id_alumno,
    CONCAT(a.nombre, ' ', a.apellido) AS "Alumno",
    CONCAT(m.nombre, ' ', m.apellido) AS "Potencial Mentor",
    h_a.nombre_hobbie AS "Interés Común",
    CASE
      WHEN m.experiencia_bd_relacional THEN 'Sí'
      ELSE 'No'
    END AS "Experiencia Mentor BD Relacional",
    CASE
      WHEN m.experiencia_bd THEN 'Sí'
      ELSE 'No'
    END AS "Experiencia Mentor BD No Relacional",
    COUNT(h_a.id_hobbie) AS intereses_compartidos
  FROM alumno a
  JOIN alumno_hobbie ah_a ON a.id_alumno = ah_a.id_alumno
  JOIN hobbie h_a ON ah_a.id_hobbie = h_a.id_hobbie
  JOIN alumno m ON m.id_alumno <> a.id_alumno
  JOIN alumno_hobbie ah_m ON m.id_alumno = ah_m.id_alumno
  JOIN hobbie h_m ON ah_m.id_hobbie = h_m.id_hobbie
  AND h_a.id_hobbie = h_m.id_hobbie
  WHERE m.experiencia_bd_relacional OR m.experiencia_bd
  GROUP BY a.id_alumno, m.id_alumno, h_a.nombre_hobbie, a.nombre, a.apellido, m.nombre, m.apellido, m.experiencia_bd
),
mejor_mentor AS (
  SELECT
    id_alumno,
    "Alumno",
    "Potencial Mentor",
    MIN("Interés Común") AS "Interés Común",
    "Experiencia Mentor BD Relacional",
    "Experiencia Mentor BD No Relacional",
    ROW_NUMBER() OVER (
      PARTITION BY id_alumno ORDER BY intereses_compartidos DESC
    ) AS mentor_rank,
    CASE
      WHEN "Experiencia Mentor BD Relacional" = 'Sí' AND "Experiencia Mentor BD No Relacional" = 'Sí' THEN 1
      WHEN "Experiencia Mentor BD Relacional" = 'Sí' THEN 2
      WHEN "Experiencia Mentor BD No Relacional" = 'Sí' THEN 3
      ELSE 4
    END AS prioridad
  FROM intereses_comunes
  GROUP BY id_alumno, "Alumno", "Potencial Mentor", "Experiencia Mentor BD Relacional", "Experiencia Mentor BD No Relacional", intereses_compa
)
SELECT DISTINCT
  "Alumno",
  "Potencial Mentor",
  "Interés Común",
  "Experiencia Mentor BD Relacional",
  "Experiencia Mentor BD No Relacional",
  prioridad
FROM mejor_mentor
WHERE mentor_rank = 1
ORDER BY prioridad, "Alumno"
  
```

	Alumno text	Potencial Mentor text	Interés Común text	Experiencia Mentor BD Relacional text	Experiencia Mentor BD No Relacional text	prioridad integer
1	Dragonetti	Gibelli	Lectura	Sí	Sí	1
2	Perez	Gibelli	Lectura	Sí	Sí	1
3	Pugliese	Jotallan Calvetti	Videojuegos	Sí	Sí	1
4	Ruina	Emiliano Ferretti	Escuchar Musica	Sí	Sí	1
5	Alan Exarchos	Jotallan Calvetti	Videojuegos	Sí	Sí	1

Parte 5: Procedimientos, Triggers y Automatización

Análisis Teórico de Procedimientos Almacenados

¿Qué es un procedimiento almacenado y para qué se usa?

Un procedimiento almacenado es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Es un objeto que se crea con la sentencia **CREATE PROCEDURE** y se invoca con la sentencia **CALL**. Un procedimiento puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida.

El propósito de utilizar **PROCEDURE** es producir una mejora en la legibilidad y eficiencia del código, así como también un mejor control de errores, ya que este nos permite hacer una estructura que aglutina una serie de instrucciones con el fin de evitar tener código repetido.

Los beneficios que podemos encontrar a la hora de utilizar un procedimiento almacenado son los siguientes:

Características	Detalle
Modularidad	Al encapsular el código dentro de los procedimientos, puede dividir la lógica compleja en fragmentos manejables, lo que facilita su comprensión y mantenimiento.
Reutilización	Los procedimientos almacenados se pueden llamar desde múltiples lugares de su aplicación, promoviendo la reutilización del código y reduciendo la duplicación.
Rendimiento	Al ejecutar procedimientos almacenados en el lado del servidor, se minimiza la sobrecarga de la red y mejora el rendimiento general.
Integridad de datos	Los procedimientos almacenados pueden hacer cumplir reglas comerciales y restricciones de integridad, garantizando que los datos sean consistentes y válidos.

En los procedimientos almacenados podemos tener tres tipos de parámetros:

- Entrada: Se indican poniendo la palabra reservada **IN** delante del nombre del parámetro. Estos parámetros no pueden cambiar su valor dentro del procedimiento, es decir, cuando el procedimiento finalice estos parámetros tendrán el mismo valor que tenían cuando se hizo la llamada al procedimiento. En programación sería equivalente al paso por valor de un parámetro.
- Salida: Se indican poniendo la palabra reservada **OUT** delante del nombre del parámetro. Estos parámetros cambian su valor dentro del procedimiento. Cuando se hace la llamada al procedimiento empiezan con un valor inicial y cuando finaliza la ejecución del procedimiento pueden terminar con otro valor diferente. En programación sería equivalente al paso por referencia de un parámetro.
- Entrada/Salida: Es una combinación de los tipos **IN** y **OUT**. Estos parámetros se indican poniendo la palabra reservada **IN/OUT** delante del nombre del parámetro.

Cómo utilizar el PROCEDURE en PostgreSQL

En el caso de este tp vamos a explicar el **PROCEDURE** de postgresSQL ya que es el motor de base de datos utilizado.

En PostgresSQL vamos a poder crear un nuevo procedimiento utilizando **CREATE PROCEDURE** o también podemos utilizar **CREATE OR REPLACE PROCEDURE** que esto nos va a permitir crear un nuevo procedimiento o reemplazar una definición existente. Para que el usuario pueda definir un procedimiento este debe de tener el privilegio de utilizar **USAGE**. Si a este procedimiento a la hora de crearlo se le asigna un nombre de schema, se va a crear en el schema especificado, pero sino a este no se le asigna un nombre de schema en la hora de creación se va a asignar a el schema actual. El nombre de un nuevo **PROCEDURE** no debe de coincidir con el nombre de un procedimiento ya creado o con una función ya existente con los mismos tipos de argumentos de entrada en el mismo schema. Sin embargo, los procedimientos y funciones de diferentes tipos de argumentos pueden compartir un nombre, pero esto puede generar una sobrecarga al sistema.

Para reemplazar la definición actual de un **PROCEDURE** existente, se debe utilizar el **CREATE OR REPLACE PROCEDURE**. Pero cuando vamos a utilizar este comando no va a ser posible cambiar el nombre o los tipos de argumentos de un procedimiento de esta manera, ya que haciendo esto lo que estaríamos realizando es un nuevo procedimiento y distinto al que queremos reemplazar. Cuando se utiliza el comando anteriormente mencionado, la propiedad y los permisos del procedimiento a reemplazar no serán cambiados. Para poder utilizar el **CREATE OR REPLACE PROCEDURE** el usuario debe de ser el propietario de este procedimiento a reemplazar (esto también incluye ser miembro del rol de propietario).

A todas las demás propiedades del **PROCEDURE** se les asignan los valores especificados o implícitos en el comando.

El usuario que crea el **PROCEDURE** se convierte en el propietario del procedimiento. Para poder crear un procedimiento, este usuario debe de tener privilegios de **USAGE** sobre los tipos de argumentos.

Para facilitar la comprensión de sus principales características y usos, a continuación, realizamos una tabla que resume los aspectos clave de los procedimientos almacenados:

Aspecto	Descripción
Definición	Conjunto de instrucciones SQL almacenadas en la base de datos, invocables mediante CALL.
Propósito	Mejorar la legibilidad del código, evitar redundancias, centralizar la lógica de negocio y optimizar la gestión de datos.
Tipos de Parámetros	<ul style="list-style-type: none"> - IN: Entrada (no modificable). - OUT: Salida (valor modificado por el procedimiento). - INOUT: Entrada y salida.
Beneficios	Modularidad, reutilización, rendimiento y garantía de integridad de datos.
Creación	Utiliza CREATE PROCEDURE o CREATE OR REPLACE PROCEDURE para definir o actualizar un procedimiento.
Invocación	Los procedimientos se ejecutan con la sentencia CALL junto con los parámetros requeridos.
Lenguajes Soportados	PostgresSQL admite lenguajes como PL/pgSQL, SQL, y lenguajes definidos por el usuario.
Control de Privilegios	Los procedimientos pueden ejecutarse con permisos del invocador (SECURITY INVOKER) o del creador (SECURITY DEFINER).
Limitaciones	Requieren diseño cuidadoso, pueden ser específicos del motor de base de datos y su uso excesivo puede complicar la migración a otros motores.

Esta tabla proporciona una visión de los procedimientos almacenados, destacando las consideraciones más importantes en su implementación.

A continuación explicaremos los beneficios específicos de su uso, vamos a mostrar dos maneras de crear un PROCEDURE en postgresSQL:

Opción 1:

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$  
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;
```

Opción 2:

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
BEGIN ATOMIC
    INSERT INTO tbl VALUES (a);
    INSERT INTO tbl VALUES (b);
END;
```

Después para hacer uso de este **PROCEDURE** deberíamos utilizar el comando **CALL**, A continuación vamos a dejar un ejemplo de uso de este comando:

```
CALL insert_data(1, 2);
```

Para poder utilizar el **PROCEDURE** vamos a disponer el uso de varios parámetros que a continuación se va a detallar cada uno de ellos:

- name: el name (opcionalmente calificado por el schema) del procedimiento a crear.
- argmode: El modo de un argumento: **IN**, **OUT**, **INOUT** o **VARIADIC**. Si se omite, el valor predeterminado es **IN**.
- argtype: Los tipos de datos de los argumentos del procedimiento(opcionalmente calificados por el schema), si estos ya están definidos. Los tipos de argumentos pueden ser de tipo base, composite, o domain types, o a su vez estos pueden hacer referencia al tipo de una columna de la tabla. Dependiendo del lenguaje utilizado, también se puede permitir especificar “pseudotipos”, como **cstring**. Los pseudotipos indican que el tipo de argumento real no está especificado de forma completa o se encuentra fuera del conjunto de tipos de datos SQL ordinarios. Se puede hacer referencia al tipo de una columna escribiendo **table_name.column_name%TYPE**, esta función nos va ayudar a veces a que un procedimiento sea independiente de los cambios en la definición de una tabla.

- default_expr: Una expresión que se utilizará como valor predeterminado si no se especifica el parámetro. La expresión debe poder convertirse en el tipo de argumento del parámetro. Todos los parámetros de entrada que siguen a un parámetro con un valor predeterminado también deben tener valores predeterminados.
- lang_name: El nombre del lenguaje en el que se implementa el procedimiento. Puede ser SQL, C, interno o el nombre de un lenguaje de procedimiento definido por el usuario, por ejemplo., plpgsql. El valor predeterminado es SQL si se especifica sql_body. No se recomienda escribir el nombre entre comillas simples y es necesario que coincidan las mayúsculas y minúsculas.
- TRANSFORM { FOR TYPE type_name } [....]: Se deben aplicar las listas que transforman una llamada al procedimiento. Las transformaciones se hacen entre tipos SQL y tipos de datos específicos del lenguaje; consulte CREATE TRANSFORM. Las implementaciones de lenguaje procedural generalmente tienen conocimiento codificado de los tipos integrados, por lo que no es necesario incluirlos aquí. Si una implementación de lenguaje procedural no sabe cómo manejar un tipo y no se proporciona ninguna transformación, recurrirá a un comportamiento predeterminado para convertir tipos de datos, pero esto depende de la implementación.
- [EXTERNAL] SECURITY INVOKER y [EXTERNAL] SECURITY DEFINER: SECURITY INVOKER indica que el procedimiento se ejecutará con los privilegios del usuario que lo llama. Ese es el valor predeterminado. SECURITY DEFINER especifica que el procedimiento se ejecutará con los privilegios del usuario que lo posee. La palabra clave EXTERNAL está permitida para la conformidad con SQL, pero es opcional ya que, a diferencia de SQL, esta característica se aplica a todos los procedimientos, no solo a los externos. Un procedimiento SECURITY DEFINER no puede ejecutar instrucciones de control de transacciones (por ejemplo, COMMIT y ROLLBACK, según el lenguaje).

Estos parámetros anteriormente pueden ser configurados con las cláusulas mencionadas a continuación:

- value: La cláusula SET hace que el parámetro de configuración especificado se establezca en el valor especificado cuando se ingresa al procedimiento y luego se restablece a su valor anterior cuando el procedimiento sale. SET FROM CURRENT guarda el valor del parámetro que es actual cuando se ejecuta CREATE PROCEDURE como el valor que se aplicará cuando se ingresa al procedimiento. Si se adjunta una cláusula SET a un procedimiento, los efectos de un comando SET LOCAL ejecutado dentro del procedimiento para la misma variable se limitan al procedimiento: el valor anterior del parámetro de configuración aún se restaura al salir del procedimiento. Sin embargo, un comando SET común (sin LOCAL) anula la cláusula SET, de manera similar a como lo haría con un comando SET LOCAL anterior: los efectos de dicho comando persistirán después de salir del procedimiento, a menos que se revierta la transacción actual. Si se adjunta una cláusula SET a un procedimiento, ese

procedimiento no puede ejecutar instrucciones de control de transacciones (por ejemplo, COMMIT y ROLLBACK, según el lenguaje).

- definition: Una constante de tipo cadena que define el procedimiento; el significado depende del lenguaje. Puede ser un nombre de procedimiento interno, la ruta a un archivo de objeto, un comando SQL o texto en un lenguaje procedural. A menudo es útil usar dollar quoting para escribir la cadena de definición del procedimiento, en lugar de la sintaxis habitual de comillas simples. Sin las dollar quoting, cualquier comilla simple o barra invertida en la definición del procedimiento debe ser escapada aplicándolas.
- obj_file y link_symbol: Esta forma de la cláusula AS se utiliza para procedimientos de lenguaje C que se pueden cargar dinámicamente cuando el nombre del procedimiento en el código fuente de lenguaje C no es el mismo que el nombre del procedimiento SQL. La cadena obj_file es el nombre del archivo de biblioteca compartida que contiene el procedimiento C compilado y se interpreta como para el comando LOAD. La cadena link_symbol es el símbolo de enlace del procedimiento, es decir, el nombre del procedimiento en el código fuente de lenguaje C. Si se omite el símbolo de enlace, se supone que es el mismo que el nombre del procedimiento SQL que se está definiendo. Cuando las llamadas CREATE PROCEDURE repetidas hacen referencia al mismo archivo de objeto, el archivo solo se carga una vez por sesión. Para descargar y volver a cargar el archivo (quizás durante el desarrollo), se debe iniciar una nueva sesión.
- sql_body: El cuerpo de un procedimiento SQL de LANGUAGE. Debe ser un bloque

```
BEGIN ATOMIC
statement;
statement;
...
statement;
END
```

Esto es similar a escribir el cuerpo del procedimiento como una constante de tipo cadena (ver la definición anterior), pero hay algunas diferencias: esta forma solo funciona para LANGUAGE SQL, mientras que la forma de cadena constante funciona para todos los lenguajes. Esta forma se analiza (o se interpreta) en el momento en que se define el procedimiento, mientras que la forma de cadena constante se analiza en el momento de la ejecución; por lo tanto, esta forma no puede soportar tipos de argumentos polimórficos y otras construcciones que no se pueden resolver al momento de la definición del procedimiento. Esta forma rastrea las dependencias entre el procedimiento y los objetos usados en el cuerpo del procedimiento, por lo que el comando DROP ... CASCADE funcionará correctamente, mientras que la forma que usa

literales de cadena podría dejar procedimientos colgantes (sin referencias). Finalmente, esta forma es más compatible con el estándar SQL y con otras implementaciones de SQL.

Ejemplo de caso de uso del procedure en PostgreSQL

Creación del schema y de las tablas que vamos a utilizar

```
CREATE SCHEMA procedimiento;
```

Para poder llevar a cabo esto, primero tendríamos que realizar el creado de las tablas que vamos a utilizar y el cargado de estas mismas, en este ejemplo, vamos a utilizar la tabla alumnos que nos va a servir para guardar la información del alumno, la tabla materia que nos va a servir para guardar la información de las materias que se dan en la universidad y la tabla inscripciones, que seria una tabla intermedia entre la tabla de alumnos y la tabla materia, que nos va a servir para guardar la información de las inscripciones a las materias realizadas en la universidad

Creado y carga de datos de la tabla alumnos

```
CREATE TABLE procedimiento.alumnos (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(100),
    apellido VARCHAR(100)
);

INSERT INTO procedimiento.alumnos(nombre,apellido) VALUES
('Juan', 'Martinez'),
('Pedro', 'Garcia'),
('Luis', 'Gimenez');
```

Creado y carga de datos de la tabla materias

```
CREATE TABLE procedimiento.materias (
    id SERIAL PRIMARY KEY,
    nombre_materia VARCHAR(100)
);
```

```
INSERT INTO procedimiento.materias(nombre_materia) VALUES
('Algoritmos 3'),
('Base de datos'),
('Seminario'),
('Redes');
```

Creado y carga de datos de la tabla inscripción

```
CREATE TABLE procedimiento.inscripciones (
    id SERIAL PRIMARY KEY,
    alumno_id INTEGER REFERENCES programacion.alumnos(id),
    materia_id INTEGER REFERENCES programacion.materias(id)
);
```

Al principio la tabla de inscripciones se va a encontrar vacía ya que vamos a ingresar los datos dentro de ésta a través del procedure que vamos a crear a continuación.

Creación del procedure

Para poder llevar a cabo esto primero tendríamos que crear el **PROCEDURE** que en este caso lo vamos a llamar `realizar_inscripcion`, que como parámetro va a recibir el id del alumno y el id de la materia en la que se quiere inscribir. Este tipo de parámetros se van a definir como tipo de entrada. Una vez pasados estos parámetros los vamos a insertar dentro de la tabla inscripciones.

```
CREATE PROCEDURE realizar_incripcion(IN alumno_id INT, IN materia_id INT)
LANGUAGE SQL --donde se define que lenguaje estamos utilizando
BEGIN ATOMIC
    INSERT INTO procedimiento.inscripciones(alumno_id,materia_id)
    VALUES (alumno_id, materia_id);
END;
```

Utilización del procedure creado

Para después utilizar este procedimiento almacenado tendríamos que utilizar la función **CALL** `realizar_incripcion(alumno_id, materia_id)` donde `alumno_id` sería el alumno que quiere inscribirse a la materia y `materia_id` sería la materia donde se quiere inscribir.

```
CALL realizar_incripcion(id_alumno,id_materia)
```

Como ejemplo de uso vamos a utilizar a un alumno cuyo id seria 3, se quiere inscribir a la materia de base de datos, cuyo id en nuestro caso seria el numero 2. Entonces como parámetro en la

Llamada al procedure realizar_inscripcion le tendríamos que pasar el id 3 del alumno y el id de la materia que en nuestro caso seria 2 ya que el alumno se quiere inscribir a base de datos.

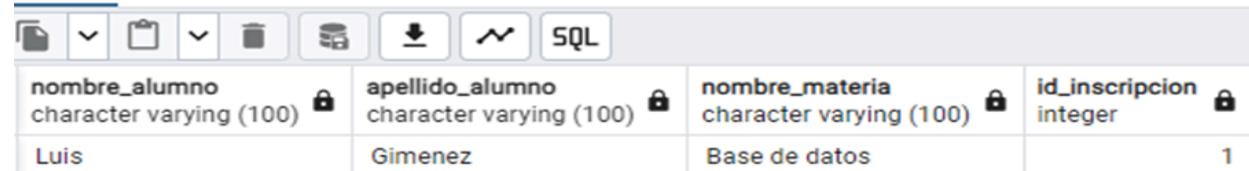
```
CALL realizar_inscripcion(3,2);
```

Haciendo la siguiente consulta podemos comprobar si la inscripción del alumno con id 3 a la materia de base de datos fue realizada con éxito.

```

SELECT
    a.nombre AS nombre_alumno,
    a.apellido AS apellido_alumno,
    m.nombre_materia AS nombre_materia,
    i.id AS id_inscripcion
FROM
    procedimiento.inscripciones i
JOIN procedimiento.alumnos a ON i.alumno_id = a.id
JOIN procedimiento.materias m ON i.materia_id = m.id;
```

Output Messages Notifications



nombre_alumno	apellido_alumno	nombre_materia	id_inscripcion
character varying (100)	character varying (100)	character varying (100)	integer
Luis	Gimenez	Base de datos	1

La ventaja de utilizar este tipo de procedimiento es que cada vez que queramos hacer una nueva inscripción de un alumno a una materia solo tendríamos que llamar a el procedimiento realizar_inscripcion, lo que esto nos va ahorrar repetición de código ya que los datos al utilizar esto se van a agregar de una manera más automatizada produciendo una mejor eficiencia como así también un mejor manejo de errores ya que los datos no se van a tener que ingresar manualmente. Si no utilizamos esto cada vez que queramos inscribir a un alumno a una materia tendríamos que usar la siguiente línea de comando para cada alumno del sistema que se quiera inscribir a una materia.

```
INSERT INTO tp.alumno_materia (id_alumno, id_materia)
VALUES (1, 2);
```

Análisis Teórico de Triggers

¿Que es un trigger y para que se usa?

Un trigger o disparador es un objeto que se asocia con tablas y se almacena en la base de datos. Su nombre se deriva por el comportamiento que presentan en su funcionamiento, ya que se ejecutan cuando sucede algún evento sobre las tablas a las que se encuentra asociado. Los

eventos que hacen que se ejecute un trigger son las operaciones de inserción (INSERT), borrado (DELETE) o actualización (UPDATE), ya que modifican los datos de una tabla.

Como tipos de triggers podemos encontrar los triggers DML que van a reaccionar ante las operaciones de INSERT, DELETE o UPDATE. También podemos encontrar los tipo de triggers DDL que van a reaccionar ante los comandos CREATE, ALTER y DROP. Como así también existen los triggers que van a reaccionar al evento de LOGON. Pero los que más se suelen utilizar son los triggers de DML, así que vamos a abarcar más en explicar estos.

La principal función de los trigger es contribuir a mejorar la gestión de la base de datos. Gracias a ellos muchas operaciones se pueden realizar de forma automática, sin necesidad de intervención humana, lo que permite ahorrar mucho tiempo.

Otra de sus funciones es aumentar la seguridad e integridad de la información. Esto lo consiguen gracias a la programación de restricciones o requerimientos de verificación que permiten minimizar los errores y sincronizar la información.

Por otra parte, entre sus principales ventajas es que todas estas funciones se pueden realizar desde la propia base de datos, es decir, no es necesario recurrir a lenguajes externos de programación.

Los triggers DML tienen dos tipos:

- FOR o AFTER [INSERT, UPDATE, DELETE]: Estos tipos de Triggers se ejecutan después de completar la instrucción de disparo (inserción, actualización o eliminación).
- INSTEAD OF [INSERT, UPDATE, DELETE]: A diferencia del tipo FOR (AFTER), los Triggers INSTEAD OF se ejecutan en lugar de la instrucción de disparo. En otras palabras, este tipo de trigger reemplaza la instrucción de disparo. Son de gran utilidad en los casos en los que es necesario tener integridad referencial entre bases de datos.

Se puede definir cuando se va a ejecutar la acción del trigger, siempre y cuando estos cambios no violen las restricciones de este, utilizando las siguientes combinaciones:

- Before statement: Antes de ejecutar la sentencia de disparo.
- Before row: Antes de modificar cada fila afectada por la sentencia de disparo, y antes de chequear las restricciones de integridad apropiadas.
- After statement: Despues de ejecutar la sentencia de disparo, y despues de chequear las restricciones de integridad apropiadas.
- After row: Despues de modificar cada fila afectada por la sentencia de disparo y posiblemente aplicando las restricciones de integridad apropiadas .

Existen dos tipos de disparadores que se clasifican según la cantidad de ejecuciones a realizar:

- Row Triggers (o Disparadores de fila): son aquellas que se ejecutarán cada vez que se llama al disparador desde la tabla asociada al trigger.
- Statement Triggers (o Disparadores de secuencia): son aquellos que sin importar la cantidad de veces que se cumpla con la condición, su ejecución es única.

Para poder realizar la creación de un trigger tenemos que seguir la siguiente estructura para su funcionamiento:

1. Se produce una llamada de activación al código que se ha de ejecutar.
2. Aplica las restricciones necesarias para poder realizar la acción, por ejemplo, una determinada condición o una nulidad.
3. Una vez verificadas las restricciones, se ejecuta la acción, en base a las instrucciones recibidas en el primer punto.

A continuación daremos como se tendría que crear un trigger utilizando postgresSQL, ya que este es el motor de base de datos que utilizamos para el tp.

```
CREATE TRIGGER check_update
BEFORE UPDATE ON accounts
FOR EACH ROW
EXECUTE FUNCTION check_account_update();
```

Algunos ejemplos de uso de los triggers que vimos en la cursada son:

- Mantener un Historial de Cambios (Auditoría): Cuando necesitamos llevar un registro de todos los cambios que ocurren en una tabla, para poder auditar la información.
- Validar Datos Antes de Guardarlos: Los triggers se usan para validar datos antes de almacenarlos, asegurando que cumplan con las reglas de negocio, como evitar retiros por encima del saldo disponible en un sistema bancario.
- Sincronizar Cambios en Tablas Relacionadas: Cuando necesitamos asegurarnos de que los cambios en una tabla se reflejan automáticamente en otras tablas relacionadas.
- Restringir Operaciones No Permitidas: Cuando queremos prevenir operaciones no permitidas en la base de datos.

Ejemplo de caso de uso del trigger en PostgreSQL

Para dar el ejemplo práctico vamos a ahondar en el caso de uso de una auditoría, como ejemplo, tomaremos la actualización de la información de un alumno en la universidad, ya que nosotros no lo implementamos en nuestro trabajo, vamos a hacer un ejemplo desde cero.

Creación y carga de datos en la tabla alumnos

Primero tendríamos que crear la entidad alumno que va a contener los atributos que queremos actualizar

```
CREATE TABLE public.alumnos(
    id_alumno SERIAL PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    apellido VARCHAR(50) NOT NULL,
    dni VARCHAR(15) UNIQUE,
    email VARCHAR(100)
);
```

A continuación generamos unos alumnos para insertar dentro de la tabla public.alumnos en nuestro caso

```
INSERT INTO public.alumnos (nombre,apellido,dni,email) VALUES
('Pepe','Manolo','1234','example1@mail.com'),
('Juan','Cruz','5678','example2@mail.com'),
('Pedro','Carilo','1427','example3@gmail.com');
```

Una vez creados los alumnos y metidos dentro de la tabla public.alumnos hacemos un listado de la tabla public.alumnos para ver si los alumnos fueron correctamente creados

14 SELECT * FROM public.alumnos

Data Output Messages Notifications

	id_alumno [PK] integer	nombre character varying (50)	apellido character varying (50)	dni character varying (15)	email character varying (100)
1	1	Pepe	Manolo	1234	example1@mail.com
2	2	Juan	Cruz	5678	example2@mail.com
3	3	Pedro	Carilo	1427	example3@gmail.com

Creación de la tabla log que queremos utilizar

Una vez verificado si fueron creados con éxito los alumnos, creamos la función del trigger que nos va a permitir un registro de log para cuando se ejecute un comando de **INSERT**, **UPDATE** o **DELETE** en la tabla de alumnos.

Para implementar un sistema de registro de logs efectivo, primero será necesario crear una tabla específica para almacenar estos registros. Esta tabla servirá para documentar cada cambio

realizado, proporcionando un historial detallado de las operaciones. Los campos de la tabla de logs incluirán:

- Fecha: la fecha y hora en que se ejecutó la operación.
- Usuario: el usuario que realizó el cambio.
- Tipo de operación: especificará la acción realizada (por ejemplo, actualización, inserción o eliminación).
- ID del alumno: el identificador único del alumno que fue modificado.
- Campo modificado: el nombre del campo que fue actualizado.
- Valor antiguo: el valor previo del campo antes de la actualización.
- Valor nuevo: el valor actualizado del campo.

Este esquema asegurará un registro completo y detallado de los cambios, facilitando el seguimiento y la auditoría de las operaciones realizadas en la base de datos.

```

16 CREATE TABLE public.log_alumnos (
17   id_log SERIAL PRIMARY KEY,
18   fecha_cambio TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
19   usuario TEXT DEFAULT CURRENT_USER,
20   operacion TEXT,
21   id_alumno INTEGER,
22   campo_modificado TEXT,
23   valor_anterior TEXT,
24   valor_nuevo TEXT
25 );
26
27 SELECT * FROM public.log_alumnos

```



	<code>id_log</code> [PK] integer	<code>fecha_cambio</code> timestamp without time zone	<code>usuario</code> text	<code>operacion</code> text	<code>id_alumno</code> integer	<code>campo_modificado</code> text	<code>valor_anterior</code> text	<code>valor_nuevo</code> text
--	-------------------------------------	--	------------------------------	--------------------------------	-----------------------------------	---------------------------------------	-------------------------------------	----------------------------------

Creación de la función del trigger

A continuación, se crea una función que registra automáticamente los cambios en la tabla log_alumnos. Esta función será utilizada por un trigger para capturar las modificaciones realizadas en los campos relevantes de la tabla de alumnos. Cuando se detecta un cambio, la función registra la operación en la tabla de logs, indicando qué campo fue modificado, junto con el valor antiguo y el nuevo.

```

CREATE OR REPLACE FUNCTION fn_log_cambios_alumno()
RETURNS TRIGGER AS $$ 
BEGIN

IF NEW.nombre != OLD.nombre THEN
    INSERT INTO public.log_alumnos (operacion, id_alumno, campo_modificado, valor_anterior, valor_nuevo)
    VALUES ('UPDATE', OLD.id_alumno, 'nombre', OLD.nombre, NEW.nombre);
END IF;

IF NEW.apellido != OLD.apellido THEN
    INSERT INTO public.log_alumnos (operacion, id_alumno, campo_modificado, valor_anterior, valor_nuevo)
    VALUES ('UPDATE', OLD.id_alumno, 'apellido', OLD.apellido, NEW.apellido);
END IF;

IF NEW.dni != OLD.dni THEN
    INSERT INTO public.log_alumnos (operacion, id_alumno, campo_modificado, valor_anterior, valor_nuevo)
    VALUES ('UPDATE', OLD.id_alumno, 'dni', OLD.dni, NEW.dni);
END IF;

IF NEW.email != OLD.email THEN
    INSERT INTO public.log_alumnos (operacion, id_alumno, campo_modificado, valor_anterior, valor_nuevo)
    VALUES ('UPDATE', OLD.id_alumno, 'email', OLD.email, NEW.email);
END IF;

RETURN NEW;
END;
$$ LANGUAGE plpgsql;
  
```

Esta función realiza lo siguiente:

- Compara los valores antiguos y nuevos de los campos específicos (nombre, apellido, dni y email) para verificar si hubo cambios.
- Si se detecta una modificación, inserta un registro en log_alumnos con detalles como la operación realizada, el identificador del alumno, el campo modificado, el valor anterior y el nuevo valor.
- El uso de RETURN NEW asegura que la operación que activó el trigger continúe ejecutándose correctamente.

Creación del trigger que vamos a utilizar

Una vez creada la función que se encargará de registrar los cambios, procedemos a definir el trigger que se encargará de ejecutar automáticamente tras cada modificación en la tabla alumnos. Este trigger se activará después de cada operación de actualización, garantizando que cualquier cambio en los datos de un alumno se registre en la tabla de logs.

```

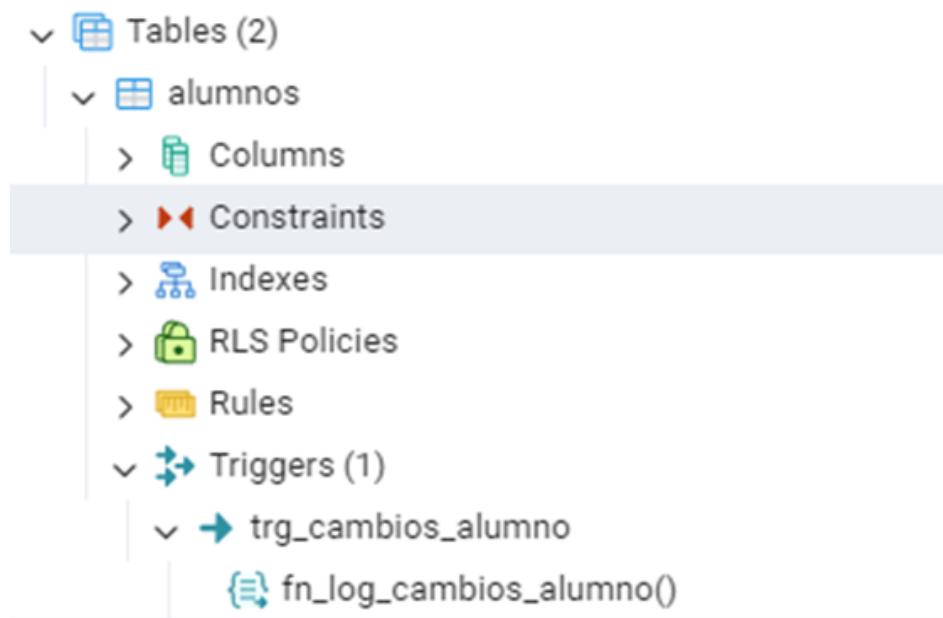
CREATE TRIGGER trg_cambios_alumno
AFTER UPDATE ON alumnos
FOR EACH ROW
EXECUTE FUNCTION fn_log_cambios_alumno();
  
```

Este trigger realiza lo siguiente:

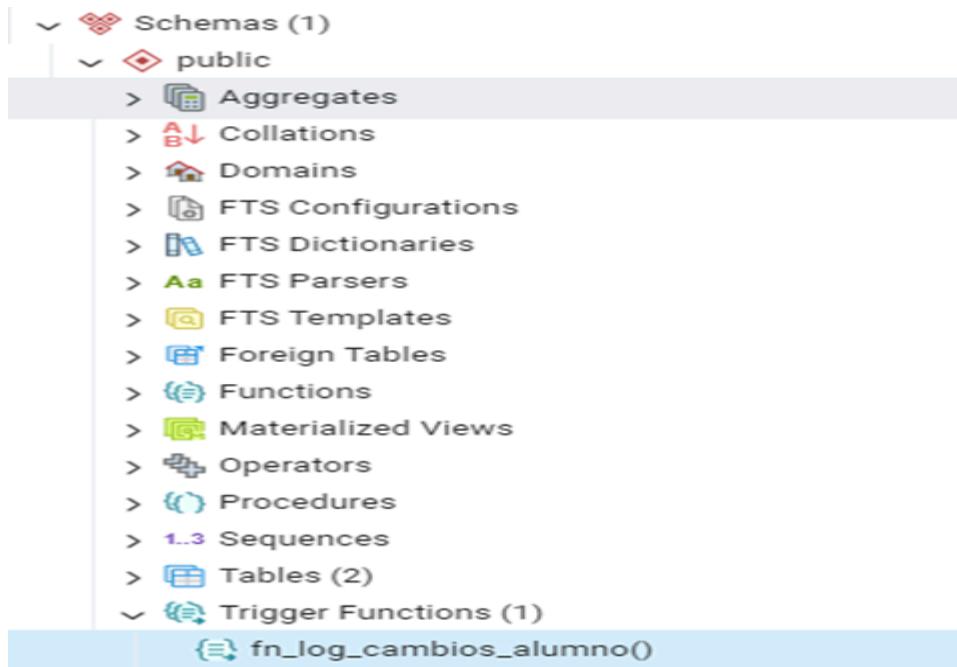
- AFTER UPDATE: Especifica que el trigger se ejecutará después de que se haya realizado una operación de actualización en la tabla de alumnos.
- FOR EACH ROW: Indica que el trigger se ejecutará una vez por cada fila que se modifique.
- EXECUTE FUNCTION fn_log_cambios_alumno(): Llama a la función fn_log_cambios_alumno, la cual registrará los detalles de los cambios en la tabla log_alumnos.

Ubicación de donde se nos crea el trigger y la función de este mismo

El trigger se nos va a crear en la pestaña de triggers de la tabla alumnos en nuestro caso



Y la función del trigger la podremos encontrar dentro de la carpeta triggers functions del schema creado(en nuestro ejemplo el schema es public).



Ejemplos de caso de uso del trigger creado anteriormente

A continuación, daremos dos ejemplos de caso de uso de este trigger creado. El primer ejemplo va ahondar en la actualización de un solo campo de un alumno, mientras que en el segundo caso vamos a ahondar en la actualización de múltiples campos de un alumno.

Supongamos que para el primer ejemplo el alumno de ID 1, viene hacia a nosotros diciendo que tuvo que cambiar el mail por el cual se había registrado a nuestro sistema por un desperfecto técnico. En la tabla creada, nuestro alumno con ID 1 sería Pepe Manolo, entonces lo que tendríamos que hacer es actualizar el campo de mail de este mismo. Para ello vamos a ejecutar el siguiente comando.

```
UPDATE public.alumnos
SET email = 'pepe.manolo@nuevoemail.com'
WHERE id_alumno = 1;
```

Este comando nos va a permitir hacer:

- UPDATE: el UPDATE nos va a permitir elegir qué tabla queremos actualizar, en nuestro caso sería public.alumnos.
- SET: el SET nos va a permitir que campo de nuestra tabla queremos actualizar, en nuestro primer ejemplo solo sería el campo de email.
- WHERE: Con el WHERE vamos a poder elegir a través del id del alumno, que alumno deseamos actualizar, en este ejemplo sería el alumno cuyo id es 1.

Ahora para poder ver si se realizó correctamente la actualización, vamos a realizar la siguiente consulta a la tabla de public.log_alumnos para ver si esta actualización del campo del mail se registró correctamente en el sistema.

```
SELECT
*
FROM log_alumnos
WHERE id_alumno = 1 --Nos permite elegir el registro de cambios de un alumno en particular
ORDER BY fecha_cambio DESC;
```

Output Messages Notifications

id_log [PK] integer	fecha_cambio timestamp without time zone	usuario text	operacion text	id_alumno integer	campo_modificado text	valor_anterior text	valor_nuevo text
1	2024-11-17 01:01:41.016827	postgres	UPDATE	1	email	example1@mail.com	pepe.manolo@nuevoemail.com

```
SELECT * FROM alumnos WHERE id_alumno = 1;
```

Output Messages Notifications

id_alumno [PK] integer	nombre character varying (50)	apellido character varying (50)	dni character varying (15)	email character varying (100)
1	Pepe	Manolo	1234	pepe.manolo@nuevoemail.com

Entonces como podemos observar en las anteriores dos imágenes, el cambio de mail del alumno cuyo ID es 1 fue realizado con éxito.

Para el segundo ejemplo supongamos que el alumno con ID 2, se tuvo que cambiar el apellido ya que este se caso y quiere tener el apellido del esposo/a, como así también se tuvo que cambiar el mail con el cual se había registrado en nuestro sistema ya que este se olvido su contraseña de ingreso a esta cuenta. En nuestra tabla el alumno cuyo ID es 2 sería Juan Cruz, entonces lo que nosotros deberíamos de realizar es la actualización de los campos de apellido y mail de este alumno. Para poder llevar a cabo esto, tendremos que utilizar la siguiente consulta.

```
UPDATE alumnos
SET
    apellido = 'Quintero',
    email = 'juan.quintero@email.com'
WHERE id_alumno = 2;
```

Para poder ver si se actualizó con éxito los cambios de los campos, vamos a ejecutar las mismas consultas que realizamos en el punto anterior pero esta vez nuestro id_alumno seria 2.

```
SELECT
*
FROM log_alumnos
WHERE id_alumno = 2
ORDER BY fecha_cambio DESC;
```

Output Messages Notifications

id_log [PK] integer	fecha_cambio timestamp without time zone	usuario text	operacion text	id_alumno integer	campo_modificado text	valor_anterior text	valor_nuevo text
2	2024-11-17 01:48:36.808555	postgres	UPDATE	2	apellido	Cruz	Quintero
3	2024-11-17 01:48:36.808555	postgres	UPDATE	2	email	example2@mail.com	juan.quintero@email.com

```
SELECT * FROM alumnos WHERE id_alumno = 2;
```

Output Messages Notifications

id_alumno [PK] integer	nombre character varying (50)	apellido character varying (50)	dni character varying (15)	email character varying (100)
2	Juan	Quintero	5678	juan.quintero@email.com

Entonces una vez realizados los ejemplos, nuestra tabla de registros de logs nos tendría que quedar de esta manera.

```
SELECT
*
FROM log_alumnos
ORDER BY fecha_cambio DESC;
```

Output Messages Notifications

id_log [PK] integer	fecha_cambio timestamp without time zone	usuario text	operacion text	id_alumno integer	campo_modificado text	valor_anterior text	valor_nuevo text
2	2024-11-17 01:48:36.808555	postgres	UPDATE	2	apellido	Cruz	Quintero
3	2024-11-17 01:48:36.808555	postgres	UPDATE	2	email	example2@mail.com	juan.quintero@email.com
1	2024-11-17 01:01:41.016827	postgres	UPDATE	1	email	example1@mail.com	pepe.manolo@nuevoemail.com

La principal ventaja de utilizar este tipo de trigger es que nos permite mantener un registro detallado y completo de las actualizaciones realizadas en los datos de los alumnos. Este registro proporciona información valiosa, como el usuario que efectuó el cambio, la fecha y hora en que se realizó, el alumno afectado, y el campo específico que fue modificado. Esta trazabilidad es crucial para detectar y analizar posibles errores del sistema, así como para auditar y supervisar las operaciones de manera eficaz, mejorando la transparencia y la seguridad de los datos.

Propuesta de Estrategias de Automatización

La automatización de bases de datos abarca el uso de herramientas y técnicas para realizar tareas repetitivas de manera programada y eficiente. Esto va desde la creación y configuración inicial de bases de datos hasta la gestión continua de actualizaciones, migraciones de datos y monitoreo del rendimiento. Al automatizar estos procesos, se liberan recursos humanos para tareas más creativas y estratégicas, al tiempo que se minimizan los riesgos de fallos y se aumenta la consistencia y la confiabilidad de los datos.

A continuación pasaremos a explicar algunas técnicas básicas de automatización que se utilizan en base de datos.

Índices de optimización

Un índice en una base de datos es una estructura que permite acelerar el acceso a los datos en las tablas, ayudando a optimizar las consultas, reduciendo el tiempo de búsqueda de datos y mejorando la eficiencia del sistema.

Los índices no cambian la información almacenada en la base de datos, pero facilitan su recuperación. Sin embargo, al agregar índices, también se aumenta el costo de las operaciones de escritura (inserciones, actualizaciones y eliminaciones), ya que estos deben mantenerse actualizados.

Hay diferentes tipos de índices que se pueden utilizar en SQL, pero los más comunes son:

- Índice de Clave Primaria: Se crea automáticamente cuando se define una clave primaria en una tabla. Garantiza la unicidad de los valores en la columna de clave primaria y acelera la búsqueda por ese campo.
- Índice de Clave Externa: Se crea automáticamente cuando se define una clave externa (foreign key) en una tabla. Acelera las operaciones de join entre tablas relacionadas.
- Índice No Agrupado: Se crea explícitamente en una o varias columnas seleccionadas por el desarrollador. Mejora la velocidad de búsqueda de los datos, pero no altera el orden físico de los registros en la tabla.

En PostgreSQL podemos encontrar los siguientes tipos de índices que nos va a ayudar a realizar las consultas con una mayor optimización

1. Índice B-Tree (Árbol B):

Este es el tipo de índice más común y el predeterminado en PostgreSQL. Es ideal para operaciones de igualdad (=) y rango (<, >, BETWEEN).

- **Ejemplo:** Buscar todos los alumnos cuyo DNI tenga un valor específico.

2. Índice Hash:

Optimizado para búsquedas de igualdad. Aunque no es tan versátil como el índice B-Tree, puede ser útil en columnas donde solo se realizan consultas de igualdad.

- **Ejemplo:** Buscar alumnos por correo electrónico exacto.

3. Índice GiST y GIN:

Diseñados para datos complejos, como búsquedas textuales o espaciales.

- **Ejemplo:** Usar GiST para buscar alumnos dentro de una región geográfica (ubicación).

4. Índice de Árbol R:

Especializado en datos espaciales, como puntos o coordenadas.

- **Ejemplo:** Almacenar ubicaciones y realizar búsquedas dentro de un área específica.

5. Índices compuestos y únicos:

- Los índices compuestos combinan varias columnas para optimizar búsquedas que dependen de múltiples criterios.
- Los índices únicos garantizan que no haya valores duplicados en una columna específica.

A continuación daremos ejemplos de uso de los índices de optimización en PostgreSQL

Si usamos la tabla alumno en la base de datos, y queremos optimizar las búsquedas de alumnos por su número de DNI. En este caso, podemos utilizar un índice B-Tree.

```
CREATE INDEX idx_dni ON alumno (dni);
```

También se podría aplicar en el caso de email

```
CREATE UNIQUE INDEX idx_email ON alumno (email);
```

También podríamos hacer el uso incide mixto para acelerar la búsqueda de inscripción de alumno y materia:

```
CREATE INDEX idx_alumno_materia ON alumno_materia (id_alumno, id_materia);
```

Uso de vistas

Una vista en una base de datos es una tabla virtual basada en el resultado de una consulta SQL. Aunque no almacena datos por sí misma, permite a los usuarios interactuar con ella como si fuera

una tabla real. Las vistas son especialmente útiles para simplificar consultas complejas, garantizar la seguridad de los datos y proporcionar un acceso estructurado a la información.

En PostgreSQL, las vistas pueden ser **estáticas** o **materializadas**:

- **Vistas Estáticas:** Son dinámicas y reflejan siempre la información actual de las tablas subyacentes. Sin embargo, no mejoran el rendimiento, ya que ejecutan la consulta subyacente cada vez que se acceden.
- **Vistas Materializadas:** Almacenan el resultado de la consulta, lo que mejora el rendimiento en consultas repetitivas, pero requieren una actualización manual para reflejar cambios en los datos subyacentes.

A continuación veremos ejemplos de las vistas

Ejemplo de vista estática

Continuando con nuestro caso del trabajo práctico si consideramos las tablas como alumno, alumno_materia, materia, y queremos mostrar una lista consolidada de alumnos inscritos en cada materia.

Podríamos crear la siguiente vista:

```
CREATE VIEW vista_alumnos_materias AS
SELECT
    a.nombre AS nombre_alumno,
    a.apellido AS apellido_alumno,
    m.nombre_materia AS materia
FROM
    alumno a
JOIN
    alumno_materia am ON a.id_alumno = am.id_alumno
JOIN
    materia m ON am.id_materia = m.id_materia;
```

Para poder acceder a ella consultaremos de la siguiente manera:

```
SELECT * FROM vista_alumnos_materias;
```

Ejemplo de vista Materializada

Si deseamos consultar datos de manera frecuente y estos no cambian con regularidad, podemos utilizar una vista materializada. Estas vistas no solo permiten acceder rápidamente a datos previamente almacenados, mejorando el rendimiento en consultas repetitivas, sino que también ofrecen la posibilidad de ser actualizadas o sobrescritas.

```
CREATE MATERIALIZED VIEW vista_alumnos_materias_mat AS
SELECT
    a.nombre AS nombre_alumno,
    a.apellido AS apellido_alumno,
    m.nombre_materia AS materia
FROM
    alumno a
JOIN
    alumno_materia am ON a.id_alumno = am.id_alumno
JOIN
    materia m ON am.id_materia = m.id_materia;
```

Esto se logra mediante el comando REFRESH MATERIALIZED VIEW, el cual sobrescribe los datos de la vista con información actualizada desde las tablas. Esto es particularmente necesario en escenarios donde los datos cambian de forma periódica, pero no lo suficiente como para justificar el uso de una vista dinámica.

```
REFRESH MATERIALIZED VIEW vista_alumnos_materias_mat;
```

Programación de Tareas

En PostgreSQL, las tareas automáticas no están gestionadas directamente por el motor de la base de datos, sino que se realizan mediante herramientas externas o extensiones, como **pg_cron** o **cron** del sistema operativo. Estas herramientas permiten ejecutar comandos SQL o scripts en intervalos definidos.

Principales Aplicaciones

1. **Backups Automáticos:** Realizar copias de seguridad periódicas para proteger la base de datos ante fallos o pérdidas de datos.
2. **Actualización de Vistas Materializadas:** Refrescar vistas materializadas para que reflejen los datos actuales.
3. **Monitoreo y Mantenimiento:** Tareas como limpieza de logs, actualización de índices o eliminación de datos antiguos.
4. **Integraciones:** Sincronización con otras bases de datos o sistemas externos.

Backups

Realizar un backup de la base de datos es una práctica esencial para garantizar la seguridad de los datos y la recuperación ante fallos. Sin un backup adecuado, cualquier pérdida o corrupción de datos podría resultar en la pérdida irreversible de información crítica. La realización de los backups nos pueden llegar a servir para las siguientes cosas:

- Protección contra fallos del sistema: En el caso de que ocurra un error en el sistema, como un corte de energía, una corrupción de archivos o un fallo de hardware, el backup permite restaurar los datos a su estado previo. Esto ayuda a minimizar el impacto en las operaciones de la empresa.
- Prevención ante errores humanos: Los errores humanos, como la eliminación accidental de datos o modificaciones incorrectas, son comunes. Un backup actualizado permite restaurar los datos a su estado anterior, reduciendo el riesgo de pérdida debido a este tipo de errores.
- Recuperación ante ataques: En el caso de un ataque cibernético (como un ransomware) que comprometa la base de datos, tener una copia de seguridad fuera del sistema afectado permite restaurar los datos sin ceder a las demandas de los atacantes.
- Cumplimiento de normativas: Muchas organizaciones deben cumplir con normativas y estándares de seguridad que exigen tener copias de seguridad periódicas de la información almacenada en sus bases de datos, como en el caso de regulaciones de privacidad de datos.
- Facilita la migración y mantenimiento: Los backups también son útiles cuando se necesita realizar mantenimiento o actualizaciones en la base de datos, ya que permiten una forma de "seguridad" para revertir cualquier cambio en caso de que algo salga mal durante el proceso.

Automatización de Backups

El objetivo es programar un proceso que realice una copia de seguridad de la base de datos todas las noches a las 2:00 AM.

```
0 2 * * * pg_dump -U usuario -h host -p puerto nombre_de_base_de_datos > /ruta/donde/guardar/backup_$(date +%%F).sql
```

Herramienta: pg_dump

Comando para hacer un backup con la extensión .backup

```
postgres=# pg_dump -U postgres -d tp_base_datos -F  
c -f /backups/tp_base_datos.backup
```

-U: Usuario de la base de datos.

-d: Nombre de la base de datos.

-F c: Especifica el formato del archivo de backup (c: custom).

-f: Ruta y nombre del archivo de backup.

Comando para hacer un backup con la extensión .sql

```
postgres=# pg_dump -U usuario -h host -p puerto nombre_de_base_de_datos > backup.sql
```

-U: Especifica el usuario de la base de datos que tiene permisos para realizar el backup.

-h: Define el host o servidor donde se encuentra la base de datos.

-p: Especifica el puerto de conexión.

nombre_de_base_de_datos: El nombre de la base de datos que se desea respaldar.

> backup.sql: Redirige la salida del comando a un archivo de tipo SQL (con extensión .sql), que contendrá todos los datos de la base de datos.

A continuación daremos un ejemplo desde cero de cómo tendríamos que utilizar los índices, vistas y backups en nuestro proyecto de PostgreSQL

Ejemplo de caso de uso de lo anteriormente explicado en PostgreSQL

Lo primero que tendríamos que hacer es la creación de las tablas, en este caso vamos a hacer un ejemplo de una universidad donde tendremos la tabla alumnos, la tabla materias que serían las que están en la universidad, como así la tabla inscripciones que esta contendrá qué alumnos están inscriptos a la materia.

Creación de la tabla alumnos y el cargado de los datos

```
CREATE TABLE programacion.alumnos (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(100),
    apellido VARCHAR(100),
    fecha_nacimiento DATE
);
```

```
INSERT INTO programacion.alumnos (nombre, apellido, fecha_nacimiento)
VALUES
('Juan', 'Pérez', '2000-04-15'),
('Ana', 'García', '1999-12-01'),
('Luis', 'Martínez', '2001-08-20');
```

Creación de la tabla materias y el cargado de los datos

```
CREATE TABLE programacion.materias (
    id SERIAL PRIMARY KEY,
    nombre_materia VARCHAR(100),
    profesor VARCHAR(100)
);
```

```
INSERT INTO programacion.materias (nombre_materia, profesor)
VALUES
('Matemática', 'Profesor A'),
('Historia', 'Profesor B'),
('Física', 'Profesor C'),
('Redes', 'Profesor D'),
('Base de datos', 'Profesor E');
```

Creación de la tabla inscripciones y cargado de los datos

```
CREATE TABLE programacion.inscripciones (
    id SERIAL PRIMARY KEY,
    alumno_id INTEGER REFERENCES programacion.alumnos(id),
    materia_id INTEGER REFERENCES programacion.materias(id),
    fecha_inscripcion DATE
);
```

```
INSERT INTO programacion.inscripciones (alumno_id, materia_id, fecha_inscripcion)
VALUES
(1, 1, '2023-01-10'),
(2, 2, '2023-01-12'),
(3, 3, '2023-01-15'),
(1, 2, '2024-05-13'),
(2, 4, '2024-04-10'),
(3, 5, '2024-08-2');
```

Creación y uso de los índices de optimización

Una vez hechas las tablas lo que haremos es crear los índices de optimización para realizar de una manera más optimizada las consultas de la tabla de inscripciones ya sea a través del id del alumno o a través del id de la materia.

```
CREATE INDEX idx_alumno_id ON programacion.inscripciones (alumno_id);
CREATE INDEX idx_materia_id ON programacion.inscripciones (materia_id);
```

El índice idx_alumno_id se va a hacer uso cuando queremos saber en cuáles materias está inscrito un alumno en específico, esta consulta lo haremos a base del id de este mismo de la siguiente manera

```
SELECT
    a.nombre AS nombre_alumno,
    m.nombre_materia AS nombre_materia,
    i.fecha_inscripcion
FROM
    programacion.inscripciones i
JOIN programacion.alumnos a ON i.alumno_id = a.id
JOIN programacion.materias m ON i.materia_id = m.id
WHERE
    i.alumno_id = 1;
```

Output Messages Notifications

nombre_alumno	nombre_materia	fecha_inscripcion
character varying (100) 	character varying (100) 	date 
Juan	Matemática	2023-01-10
Juan	Historia	2024-05-13

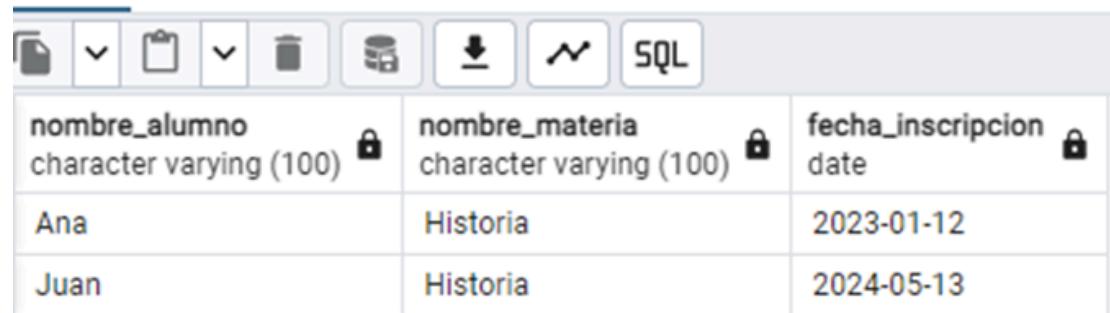
En este caso quisimos saber en cuáles materia está inscripto cuyo id es 1, si usted quiere hacer una consulta sobre otro alumno debería reemplazar el id que está definido en la línea i.alumno_id

El índice idx_materia_id se va a utilizar cuando queramos hacer uso de una consulta en la tabla de inscripciones sobre cuáles alumnos están inscritos en una materia específica a través de su id. Esta consulta la tendremos que realizar de la siguiente manera

```

SELECT
    a.nombre AS nombre_alumno,
    m.nombre_materia AS nombre_materia,
    i.fecha_inscripcion
FROM
    programacion.inscripciones i
JOIN programacion.alumnos a ON i.alumno_id = a.id
JOIN programacion.materias m ON i.materia_id = m.id
WHERE
    i.materia_id = 2;
  
```

Output Messages Notifications



nombre_alumno	nombre_materia	fecha_inscripcion
Ana	Historia	2023-01-12
Juan	Historia	2024-05-13

En este caso quisimos saber qué alumnos se habían inscripto en la materia cuyo id es 2, si usted quiere hacer una consulta sobre otro alumno debería reemplazar el id que está definido en la línea i.materia_id

Creación y uso de la vista estándar

Para el ejemplo de la vista vamos a crear una vista estándar para que en esta misma se nos muestre un registro de los alumnos de la universidad que en esta nos va a salir el apellido y nombre del alumno como así también en qué materias está inscripto y la fecha en la cual se inscribió.

Para la creación de esta misma debemos de hacer la siguiente consulta

```

CREATE VIEW programacion.vista_alumnos_materia AS
SELECT
    a.nombre,
    a.apellido,
    m.nombre_materia,
    i.fecha_incripcion
FROM
    programacion.inscripciones i
JOIN programacion.alumnos a ON i.alumno_id = a.id
JOIN programacion.materias m ON i.materia_id = m.id;
  
```

Una vez creada la vista, la manera de poder acceder a los datos de esta vista seria de la siguiente forma

```
SELECT * FROM programacion.vista_alumnos_materia;
```

Output Messages Notifications

nombre character varying (100)	apellido character varying (100)	nombre_materia character varying (100)	fecha_incripcion date
Juan	Pérez	Matemática	2023-01-10
Ana	García	Historia	2023-01-12
Juan	Pérez	Historia	2024-05-13
Luis	Martínez	Física	2023-01-15
Ana	García	Redes	2024-04-10
Luis	Martínez	Base de datos	2024-08-02

Esta vista nos va a permitir acceder más rápido y de una manera más optimizada a la información en general de la universidad, ya que si no hacemos uso de esta misma, cuando queramos visualizar la información deberíamos de ejecutar la siguiente consulta cada vez que queramos acceder a esta.

```
SELECT
    a.nombre,
    a.apellido,
    m.nombre_materia,
    i.fecha_inscripcion
FROM
    programacion.inscripciones i
JOIN programacion.alumnos a ON i.alumno_id = a.id
JOIN programacion.materias m ON i.materia_id = m.id;
```

Para poder realizar el backup de nuestra base de datos, lo que debemos de implementar es el comando que dimos como ejemplo anteriormente .

Conclusión Final

En este proyecto logramos desarrollar una base de datos que responde a las necesidades de la Universidad Nacional de San Martín para gestionar, centralizar y analizar la información académica de sus estudiantes. Desde los primeros pasos de análisis y limpieza de datos hasta la implementación de procedimientos avanzados, este proyecto refleja una solución.

En el análisis de datos, abordamos problemas de inconsistencia en la información inicial, logrando estandarizar y normalizar los datos para obtener una estructura ordenada. A través del Diagrama Entidad-Relación (DER), permitió definir claramente las relaciones entre las entidades clave, como alumnos, materias, grupos y localidades, entre otras.

La selección de PostgreSQL como motor de base de datos demostró ser una decisión acertada. Este motor nos permitió manejar consultas. Durante la implementación, diseñamos un esquema eficiente que facilita la organización y análisis de la información.

En la etapa de consultas y análisis, implementamos consultas avanzadas utilizando funciones, lo que nos permitió obtener resultados significativos. Por ejemplo, identificamos materias populares por grupo, hobbies dominantes por localidad y patrones académicos generales que podrían informar la planificación estratégica de la universidad. También excluimos datos como "Bases de Datos", reconociendo su influencia directa en este curso, para evitar sesgos en los análisis.

En cuanto a la automatización y optimización, desarrollamos procedimientos almacenados y diseñamos triggers hipotéticos para tareas críticas, como inscripciones automáticas y auditorías.

Consideramos que con este proyecto la universidad ahora cuenta con una herramienta estratégica que facilita la gestión diaria, como también va ayudar a la toma de decisiones informadas. Esto incluye la posibilidad de ajustar la oferta académica, identificar tendencias en las preferencias de los estudiantes y optimizar recursos en función de datos reales.

En conclusión, este trabajo práctico no solo responde a los desafíos actuales de la universidad, sino que también marca un punto de partida hacia una gestión de datos más innovadora, eficiente y orientada a las necesidades de los estudiantes y la institución. Es decir, considerándolos como dos pilares que no se pueden descuidar: el alumno y la universidad.

Si bien este proyecto se enfocó en los estudiantes, un análisis similar para los docentes sería de gran valor. Es importante reconocer que actualmente, la institución realiza encuestas al final de cada cursada para que el alumno evalúe al docente y el dictado de la materia. Consideramos que si bien son importantes estas encuestas, no son suficientes por sí solas. Centralizar información sobre los profesores, como las materias que imparten, su desempeño, el feedback de los estudiantes, su disponibilidad, intereses y trayectorias profesionales, podría ofrecer una visión más completa y estratégica de la dinámica académica. De esta manera, sería posible considerar a estudiantes, docentes y materias como un conjunto integrado, lo cual podría mejorar la planificación de la universidad.

Bibliografía

PostgreSQL. (s.f.). PostgreSQL. <https://www.postgresql.org/>

MongoDB.(s.f.).MongoDB: The Developer Data Platform. <https://www.mongodb.com/>

colaboradores de Wikipedia. (2024, 30 de Septiembre). PostgreSQL. Wikipedia, La Enciclopedia Libre.<https://es.wikipedia.org/wiki/PostgreSQL>

W3Schools.com. (s.f.). PostgreSQL Insert Data.

https://www.w3schools.com/postgresql/postgresql_insert_into.php

Python. (s.f.). 3.13.0 Documentation. <https://docs.python.org/3/index.html>

Google Colab. (s.f.). <https://colab.research.google.com/#scrollTo=Wf5KrEb6vrkR>

Internet Archive. (2020, 3 de Abril). Fundamentos De Bases De Datos (5a. Ed.) - Silberschatz : Free Download, Borrow, and Streaming : Internet Archive.

<https://archive.org/details/fundamentosdebasesdedatos5a.ed.abrahamsilberschatzhenryf.korths.sudarshan/page/n848/mode/1up>

postgresql. (s.f.). Aprendizaje postgresql. <https://riptutorial.com/Download/postgresql-es.pdf>

CastorDoc.(s.f.). How to use stored procedures in PostgreSQL?.

https://www-castordoc-com.translate.goog/how-to/how-to-use-stored-procedures-in-postgresql?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq#:~:text=They%20allow%20you%20to%20encapsulate,easier%20to%20understand%20and%20maintain

Hernández, J. J. S. (s.f.). Unidad 12. Triggers, procedimientos y funciones en MySQL.

<https://josejuansanchez.org/bd/unidad-12-teoria/index.html>

PostgreSQL Documentation. (2024, November 14). CREATE PROCEDURE.

<https://www.postgresql.org/docs/17/sql-createprocedure.html>

Tablado, F. (2020, 22 de Octubre). ¿Qué es un trigger en una base de datos? Ayuda Ley Protección Datos. <https://ayudaleyprotecciodnados.es/bases-de-datos/trigger/>

colaboradores de Wikipedia. (2022, 7 de diciembre). Trigger (base de datos). Wikipedia, La Enciclopedia Libre.

[https://es.wikipedia.org/wiki/Trigger_\(base_de_datos\)#:~:text=Un%20trigger%20o%20disparador%20es,las%20que%20se%20encuentra%20asociado.](https://es.wikipedia.org/wiki/Trigger_(base_de_datos)#:~:text=Un%20trigger%20o%20disparador%20es,las%20que%20se%20encuentra%20asociado.)

Latam, A. (2023, November 15). Qué es y cómo usar un trigger en SQL. Alura.

<https://www.aluracursos.com/blog/que-es-y-como-trigger-en-sql>