Titulación: GRADO INGENIERIA INFORMATICA Año Académico: 2019/2020 -- Curso: 2º Asignatura: Ficheros y Bases de Datos Título: Memoria Práctica 2 – Consultas, vistas y disparadores	uc3m uc3m | Universidad Carlos III de Madrid

Profesor:	**Miguel Ángel Peña Camacho**	**Grupo**	**80**
Alumno/a:	**Javier Diaz Leyva**	**NIA:**	**100383310**
Alumno/a:	**Alejandro Diaz García**	**NIA:**	**100383181**

A. Introducción

El principal objetivo de este trabajo es hacer uso del lenguaje de manipulación y control de datos de PL/SQL. Como pudimos aprender en la práctica anterior de la asignatura que se centró puramente en el diseño relacional de la base de datos y en su implementación con el lenguaje de definición de datos de SQL en Oracle. Para esta práctica partimos de una solución de diseño de la base de datos, con una serie de datos cargados sobre los cuales tendremos que realizar una serie de tareas como se van a mostrar en los siguientes apartados. Entre las tareas a realizar nos encontramos con:

1) Consultas en álgebra relacional como su implementación en SQL y pruebas para verificar su funcionamiento.
2) Diseño e implementación de vistas así como pruebas de su funcionamiento.
3) Desarrollo de tablas de históricos.
4) Diseño externo.
5) Disparadores.

B. Consultas

A continuación se van a explicar las consultas seleccionadas a desarrollar. En nuestro caso hemos seleccionado Tragicomic y Burden-user.

A. Tragicomic.

Consulta:"*protagonistas de películas que se enmarcan, al menos, en los géneros 'comedy' y 'drama' al mismo tiempo"*

a. Diseño en álgebra relacional.

i. T1: Realizamos una selección por el atributo *genre='comedy'* en la tabla *genres_movies*. Para obtener las películas que se enmarcan dentro del género de comedia.

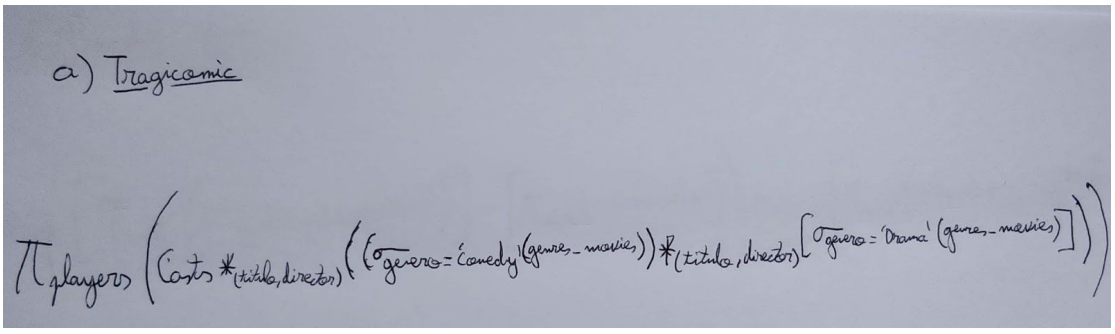
ii. T2: Realizamos una selección por el atributo *genre='drama'* en la tabla *genres_movies*. Para obtener las películas que se enmarcan dentro del género de drama.

iii. T3: Usamos la combinación natural o equijoin con *T1*T2* con los atributos *title* y *director*. Con esto obtenemos todas aquellas películas que tienen como género comedia y drama a la vez.

iv. T4: Usamos de nuevo la combinación natural o equijoin para la tabla *casts*, que es la que contiene a los actores que pertenecen al casting, es decir aquellos que son protagonistas y por otro lado en el equijoin a T3. Se resume *casts * T3* con los atributos *title* y *director*. Con esto obtenemos finalmente una tabla con aquellos actores que pertenecen a las películas con dichos géneros.

v. Por último realizamos una proyección sobre T4 con el atributo players para así obtener finalmente el nombre de todos aquellos actores que han pertenecido a películas con los géneros pedidos.

A continuación en álgebra relacional:



b. Implementación en SQL.

Entre los aspectos a destacar en la implementación se encuentran:

i. Hemos tenido que dividir en consultas más pequeñas para hacer más simple y fácil el manejo de la consulta. Todo esto gracias a WITH dentro del lenguaje SQL.

ii. Hemos decidido devolver los nombres de los actores como resultado de la query dado que es lo que se interpreta en el enunciado.

iii. Usamos DISTINCT para que no se repitan aquellos actores que han participado en más de una película con los criterios dados.

A continuación se muestra el script implementado en SQL de la consulta tragicomic, explicada con comentarios:

```sql
-- CONSULTA a TRAGICOMIC

WITH T1 AS(  -- Aqui obtenemos las peliculas cque pertenecen al genero comedia
SELECT * FROM genres_movies
WHERE genre='Comedy'
),
 T2 AS(  -- Aqui obtenemos las peliculas cque pertenecen al genero drama
SELECT *
FROM genres_movies
WHERE genre='Drama'
),
T3 AS(-- Resultado de las peliculas que pertenecen a los generos de comedia y drama
SELECT T1.TITLE,T1.DIRECTOR
FROM T1 INNER JOIN T2
    ON T1.title = T2.title AND T1.director = T2.director
),
T4 AS(
--Aqui obtenemos los actores que pertenecen a las peliculas obtenidas en las consult
as anteriores, no se especifica en ningun campo la distinccion de protagonista por l
o que sacamos los actores
SELECT *
FROM CASTS INNER JOIN T3
    ON casts.title = T3.title AND casts.director = T3.director
)
SELECT DISTINCT ACTOR FROM T4;
-- Sacamos solamente el campo de los nombres de los actores que cumplen con la condi
cion
```

c. Pruebas realizadas para demostrar su correcto funcionamiento.

Al ejecutar el Script en la base de datos tras haber ejecutado NEWcreation.sql y NEWload.sql que se nos han proporcionado hemos obtenido con esta consulta un total de 1.400 filas, es decir 1.400 actores/protagonistas que pertenecen a esa clase de películas.

i. Realizamos una prueba en la que insertamos un nuevo actor que haya realizado dos películas, un drama y una comedia, y comprobamos que la consulta ha aumentado en uno el número de actores.

ii. Realizamos una prueba en la que insertamos un nuevo actor que solo haya realizado una película que es un drama, y comprobamos que la consulta no aumenta el número de actores.

iii. Realizamos una prueba parecida a la anterior pero en vez de comprobar con un drama, comprobamos con un actor que sólo ha realizado una comedia

iv. Realizamos una prueba en la que añadimos un actor que no ha actuado en ningún drama y ninguna comedia y comprobamos que no ha aumentado el número de actores de la consulta.

v. Todas las pruebas se han comportado como se esperaba.

B. Burden-user

Consulta: *"usuarios que, llevando más de seis meses registrados, aún no pertenecen ni han pertenecido a ningún club, y tampoco tienen ni han tenido nunca contrato"*

a. Diseño en álgebra relacional.

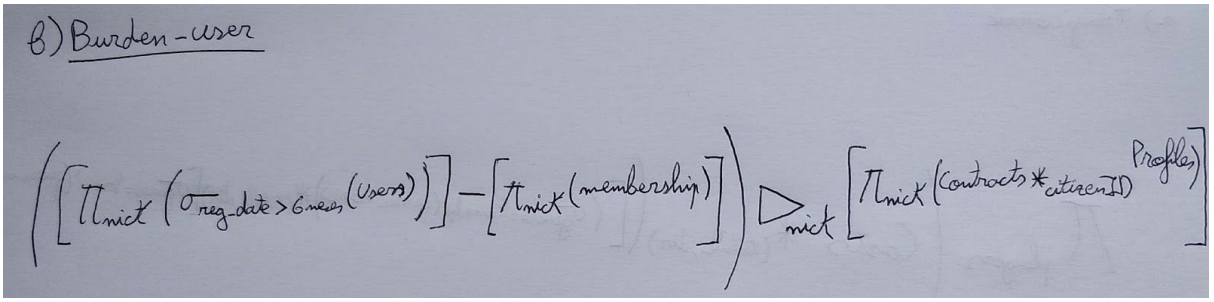
i. T1: Obtenemos los *users* que llevan más de seis meses registrados haciendo una selección de los usuarios cuyo *reg_date* es mayor que 6 meses.

ii. T2: Usamos una diferencia natural entre T1 con el *nick* de *membership*, para obtener los *users* con más de 6 meses de registro que no han formado parte de ningún club.

iii. T3: Hacemos una combinación natural entre *contracts* y *profiles* con el atributo *citizenID* y de esto hacemos una proyección de *nick,* para obtener así los nicks que han tenido o tienen un contrato.

iv. T4: Hacemos una anti-combinación izquierda entre T2 y T3 para obtener aquellos *users* que no han tenido ni tienen ningún *contract* y llevan registrados más de 6 meses.

A continuación en álgebra relacional:



b. Implementación en SQL.

Entre los aspectos a destacar en la implementación se encuentran:

i. Entendemos que nos piden los *nicks* de los usuarios que cumplen las especificaciones, de tener que devolver el *user* completo podríamos hacer una combinación natural del *T4* con usuarios con el atributo *nick*.

ii. Utilizamos *add_months* para restar meses a la fecha actual (*current_date*), le restamos dos meses que son los que especifica el enunciado que tiene que ser la antigüedad de los *users*. Y hacemos que los que se hayan registrado antes de la fecha actual menos seis meses son los que formarán T1.

iii. Y ahora seguimos la estructura especificada en el diseño relacional, en T2 tenemos los users que tienen una antigüedad mayor que seis meses y que no han pertenecido a ningún club, en T3 tenemos a los users que

tienen o han tenido un contrato, y en T4 hacemos una anti-combinación entre T2 y T3. Posteriormente nos quedamos con T4.

A continuación se muestra el script implementado en SQL de la consulta burden-user, explicada con comentarios:

```
-- CONSULTA a BURDEN-USER

WITH T1 AS(  -- Aqui obtenemos los usuarios con mas de seis de registro
SELECT nick
FROM users
WHERE reg_date<= ADD_MONTHS(current_date, -6)
),

T2 AS(
-- Aqui tenemos los usuarios con mas de 6 meses que no ha pertenecido ni pertence
n a ningun club
SELECT nick
FROM T1
WHERE nick NOT IN (SELECT nick FROM membership)
),

T3 AS(  -- Aqui cogemos lo perfiles que tienen o han tenido un contrato
SELECT nick
FROM profiles
WHERE citizenid IN (SELECT citizenid FROM contracts)
),

T4 AS(  --Aqui hacemos una anti-combinacion para obtener el resultado
SELECT nick
FROM T2
WHERE nick NOT IN (SELECT nick FROM T3)
)
SELECT * FROM T4;
```

c. Pruebas realizadas para demostrar su correcto funcionamiento.

i. Realizamos una prueba en la que añadimos un usuario con más de seis meses de registro, que no haya pertenecido a ningún club ni ha tenido ningún contrato y comprobamos que se ha sumado uno al resultado de la consulta.

ii. Realizamos un prueba de un usuario que sin tener seis meses de registro, pero que no haya pertenecido a ningún club ni ha tenido ningún contrato y comprobamos que no se ha sumado nada al resultado de la consulta.

iii. Comprobamos que con un usuario de más de seis meses, que haya pertenecido a un club, y que no ha tenido ningún contrato, no suma nada a la consulta.

iv. Comprobamos que un usuario con más de seis meses, que nunca haya pertenecido a un club, pero que ha tenido un contrato, no suma nada a la consulta.

v. Todas las pruebas se han comportado como se esperaba.

C. FilmMaster

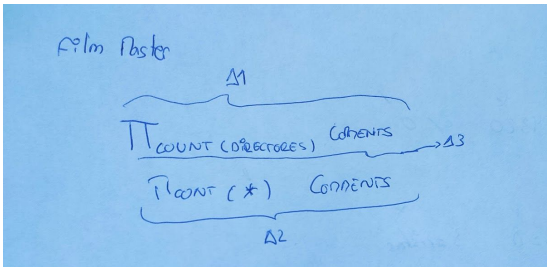
Consulta: "director cuyas películas reciben más comentarios (de media)."

a. Diseño en álgebra relacional.

i. A1: contamos los comentarios en los que sale cada director.

ii. A2: contamos los comentarios totales.

iii. Hacemos la división de A1/A2 y así obtenemos la media de cada director respecto a su aparición en los comentarios.



b. Implementación en SQL.

Entre los aspectos a destacar en la implementación se encuentran:

i. Usamos A1 para contar el número de comentarios que recibe cada director. Para ello utilizamos la función count.

ii. Creamos A2 con la intención de contar todos los comentarios existentes.

iii. En A3 realizamos la división entre A1 y A3 para obtener la media de comentarios que recibe cada director. También ordenamos la consulta de tal manera que el que mayor media tenga, salga el último en la consola y por tanto este al final, por lo que será el primero que verá el usuario.

```sql
--CONSULTA C FILMASTER
WITH A1 AS( --Numero de comentarios de cada director
SELECT
    comments.director,
    COUNT(*) AS DIR_COM
FROM
    comments
GROUP BY
    comments.director
ORDER BY
    DIR_COM
),
A2 AS( --Numero de comentarios totales
SELECT
    COUNT(*) AS DIR_COM2
FROM
    comments
),
A3 AS( --Division entre ambos para obtener la media
SELECT
    A1.director, A1.DIR_COM/A2.DIR_COM2 AS result
FROM
    A1, A2
ORDER BY
    result
)
SELECT * FROM A3;
```

c. Pruebas realizadas para demostrar su correcto funcionamiento.

i. Para el realizamiento de las pruebas en esta consultado hemos optado por hacer manualmente los cálculos del director más comentado que es Woody Allen, que aparece en 723 comentarios, y hay un total de 121716 comentarios. Por lo que la media en la que aparece Woody Allen en los comentarios es de 0,00594, que corresponde a lo que nos aparece en la consulta.

C. Vistas

Incluye una subsección por cada vista que desarrolles (al menos una, si bien puedes resolver más como trabajo opcional). Para cada vista, debes exponer:

D. su diseño en álgebra relacional
E. su implementación en SQL
F. Pruebas: no sólo se debe comprobar que la vista está bien definida (como una consulta), sino también la funcionalidad de la vista: consulta (índica el número de filas del resultado), inserción, borrado y modificación (es necesario establecer qué operaciones resuelve el gestor y qué operaciones no; pero no es necesario implementar las que el gestor no resuelva).

A continuación se va a mostrar el diseño y la implementación de las vista:

A. Captain Aragna

Vista:*"Usuarios que no comentan sus propuestas. Se proporciona el nombre de usuario (nickname), y el porcentaje de sus propuestas que no han comentado, ordenado de mayor a menor porcentaje"* . El enunciado propuesto tiene cierta ambigüedad pero entendemos que lo que se pide es sobre los usuarios que crean propuestas y no comentan sobre sus propias propuestas. Tras esta aclaración procedemos al diseño y la implementación:

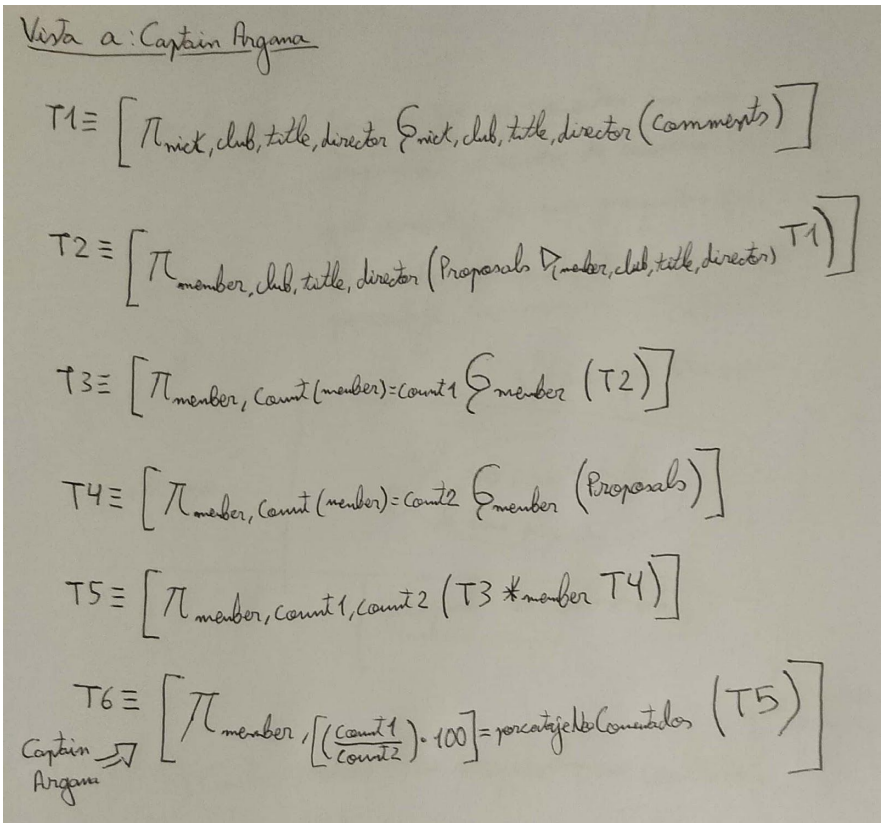
a. Diseño en álgebra relacional.

A continuación se explica la consulta diseñada dentro de la vista en álgebra relacional:

i. T1: Obtenemos todas las propuestas comentadas por cada usuario.
ii. T2: Obtenemos las propuestas no comentadas de cada usuario.
iii. T3: Contamos el número de propuestas no comentadas de cada usuario.
iv. T4: Contamos el número de propuestas totales realizadas por cada usuario.
v. T5: Sacamos una tabla con los miembros que no han comentado nunca junto a su número de propuestas no comentadas y el número total de propuestas creadas.
vi. T6: Por último obtenemos a partir de T5 el porcentaje calculandolo mediante la división del número de propuestas no contestadas y el total

de propuestas creadas por usuario, después multiplicado por 100 para obtener el porcentaje y así obtener el porcentaje de propuestas no comentadas de cada usuario junto su nombre. Hasta aquí seria la query de la vista.

A continuación en álgebra relacional:



b. Implementación en SQL.

Entre los aspectos a destacar en la implementación se encuentran:

i. Como decisión de diseño en la implementación hemos decidido dividir en subconsultas más pequeñas las cuales se crean en forma de vista para así lograr utilizar los resultados en la vista final de CaptainArgana.

```sql
-- 02 VISTA a CAPTAIN ARGANA

--SUBVISTAS / SUBQUERYS
CREATE OR REPLACE VIEW T1 AS( -- Obtenemos TODAS las propuestas comentadas POR CADA USUARIO.
SELECT NICK,CLUB,TITLE,DIRECTOR
FROM COMMENTS
GROUP BY NICK, CLUB, TITLE, DIRECTOR
);

CREATE OR REPLACE VIEW T2 AS( -- Obtenemos las propuestas NO comentadas DE CADA USUARIO.
SELECT MEMBER,CLUB, TITLE, DIRECTOR
FROM PROPOSALS
WHERE (MEMBER,CLUB,TITLE,DIRECTOR) NOT IN(SELECT NICK,CLUB,TITLE,DIRECTOR FROM T1)
);

CREATE OR REPLACE VIEW T3 AS(
SELECT T2.MEMBER, COUNT(*) AS MEM_COUNT1
-- Numero total de propuestas no comentadas por usuario de las que han sido creadas por el mismo.
FROM T2
GROUP BY MEMBER
);

CREATE OR REPLACE VIEW T4 AS(-- Propuestas creadas por cada usuario.
SELECT PROPOSALS.MEMBER, COUNT(*) AS MEM_COUNT2
-- Contamos el total de propuestas creadas por cada usuario.
FROM PROPOSALS
GROUP BY MEMBER
);

CREATE OR REPLACE VIEW T5 AS(
SELECT T3.MEMBER,T3.MEM_COUNT1,T4.MEM_COUNT2
-- Juntamos en una tabla las propuestas no comentadas por el creador y el total de propuestas cre
adas por el mismo.
FROM T3 INNER JOIN T4
    ON T3.MEMBER=T4.MEMBER
);

CREATE OR REPLACE VIEW T6 AS(
SELECT T5.MEMBER,((MEM_COUNT1/MEM_COUNT2)*100)PORCENTAJE_NO_COMENTADOS
--Realizamos la division y obtenemos el porcentaje del numero de propuestas creadas por el usuari
o que no han comentado los propios creadores.
FROM T5
) WITH READ ONLY;

-- VISTA FINAL
CREATE OR REPLACE VIEW CaptainArgana AS(
-- Vista final en la que obtenemos los resultados de T6 y los ponemos de forma ordenada.
SELECT subquery.MEMBER, subquery.PORCENTAJE_NO_COMENTADOS
FROM (SELECT T6.MEMBER, T6.PORCENTAJE_NO_COMENTADOS
FROM T6
ORDER BY PORCENTAJE_NO_COMENTADOS DESC) subquery
-- Creamos una subquery como solucion a que no permite usar el ORDER BY dentro de una vista.
)WITH READ ONLY;
```

c. Pruebas realizadas para demostrar su correcto funcionamiento.

i. Realizamos una prueba en la que añadimos una nueva propuesta a un usuario y añadimos un comentario del propio usuario también, y comprobamos que se modifica el porcentaje.

ii. Realizamos una prueba en la que añadimos una nueva propuesta a un usuario y no añadimos ningún comentario del propio usuario, y comprobamos que se modifica el porcentaje.

iii. Todas las pruebas se han comportado como se esperaba.

D.Tablas Históricas

En primer lugar, creamos las vistas necesarias, y más en concreto, primeramente, creamos la vista que inserta a los clubes actuales, es decir, aquellos que no tienen end_date. Y por otro lado creamos la vista que inserta los clubes existentes que tienen una fecha de finalización, es decir, que end_date no es null.

A continuación pasamos a la creación de los disparadores necesarios para la creación de históricos. Para ello, necesitamos cuatro triggers. El primero se encarga de que al eliminar un club de históricos, este club se elimine también de la tabla de clubs, para ellos programamos que el disparador se active cuando se borre un club de históricos, y lo que hace es borrar dicho club de la tabla de *clubs*. El segundo trigger se encarga de que cuando se elimine un elemento de la tabla de los *clubes_vigentes*, se mande hacia la tabla de *clubes_históricos*, para ellos hacemos que el disparador se active cuando se elimina un club de *clubes_vigentes*, y lo que hace es declarar una variable *date* que será la fecha actual para añadirla al *end_date* del club, y pasamos los valores del club a *clubes_históricos*. En el tercer trigger nos encargamos de que se impida la función de modificar en la vista de los *clubes_vigentes*. Y por último en el cuarto disparador, impedimos la modificación de la vista de clubes_históricos.
A continuación se adjunta la implementación en sql:

```
-- 03 Historico
--------------------- Creacion de las vistas ----------------------------------------
--------------
CREATE OR REPLACE VIEW clubes_vigentes AS(
-- Insertamos los existentes de clubs en vigentes, es decir aquellos que no tienen fech
a de fin
SELECT name,founder,cre_date,slogan,open
FROM clubs
WHERE end_date IS NULL
);

CREATE OR REPLACE VIEW clubes_historicos AS(
-- Insertamos los clubes existentes que tienen fecha de finalizacion para que consten e
n la vista de historicos
SELECT name,founder,cre_date, end_date,slogan,open
FROM clubs
WHERE end_date IS NOT NULL
);
------ Creacion de los disparadores----------------------------------------
-- Trigger 1 que al eliminar de historicos pasa a eliminarse de la tabla de clubs
CREATE OR REPLACE TRIGGER eliminar_historicos
instead of DELETE ON CLUBES_HISTORICOS
FOR EACH ROW
BEGIN
    DELETE FROM CLUBS
    WHERE CLUBS.NAME = :OLD.NAME AND CLUBS.FOUNDER = :OLD.FOUNDER;
END;
-- Trigger 2 que cuando se elimina un elemento de vigentes pase a historicos
CREATE OR REPLACE TRIGGER eliminar_vigentes_to_historico
INSTEAD OF DELETE ON clubes_vigentes
FOR EACH ROW
DECLARE
  endd_date DATE;
BEGIN
    endd_date := CURRENT_DATE;
    INSERT INTO clubes_historicos (name, founder, cre_date, end_date, slogan, open)
    VALUES(:old.name, :old.founder, :old.cre_date, endd_date, :old.slogan, :old.open);
END;
-- Trigger 3 que impida la operacion de modificar en las vistas VIGENTES
CREATE OR REPLACE TRIGGER NO_UPDATE_VIGENTES
INSTEAD OF UPDATE ON clubes_vigentes
FOR EACH ROW
BEGIN
-- SOLO EVITA LA ACTUALIZACION DE DATOS EN LA VISTA
END;
EXCEPTION
    dbms_output.put_line('No se puede actualizar la vista.');
END;
--  Trigger 4 que impida la operacion de modificar en las vistas NO VIGENTES
CREATE OR REPLACE TRIGGER NO_UPDATE_HISTORICOS
INSTEAD OF UPDATE ON clubes_historicos
FOR EACH ROW
BEGIN
-- SOLO EVITA LA ACTUALIZACION DE DATOS EN LA VISTA
END;
EXCEPTION
    dbms_output.put_line('No se puede actualizar la vista.');
END;
```

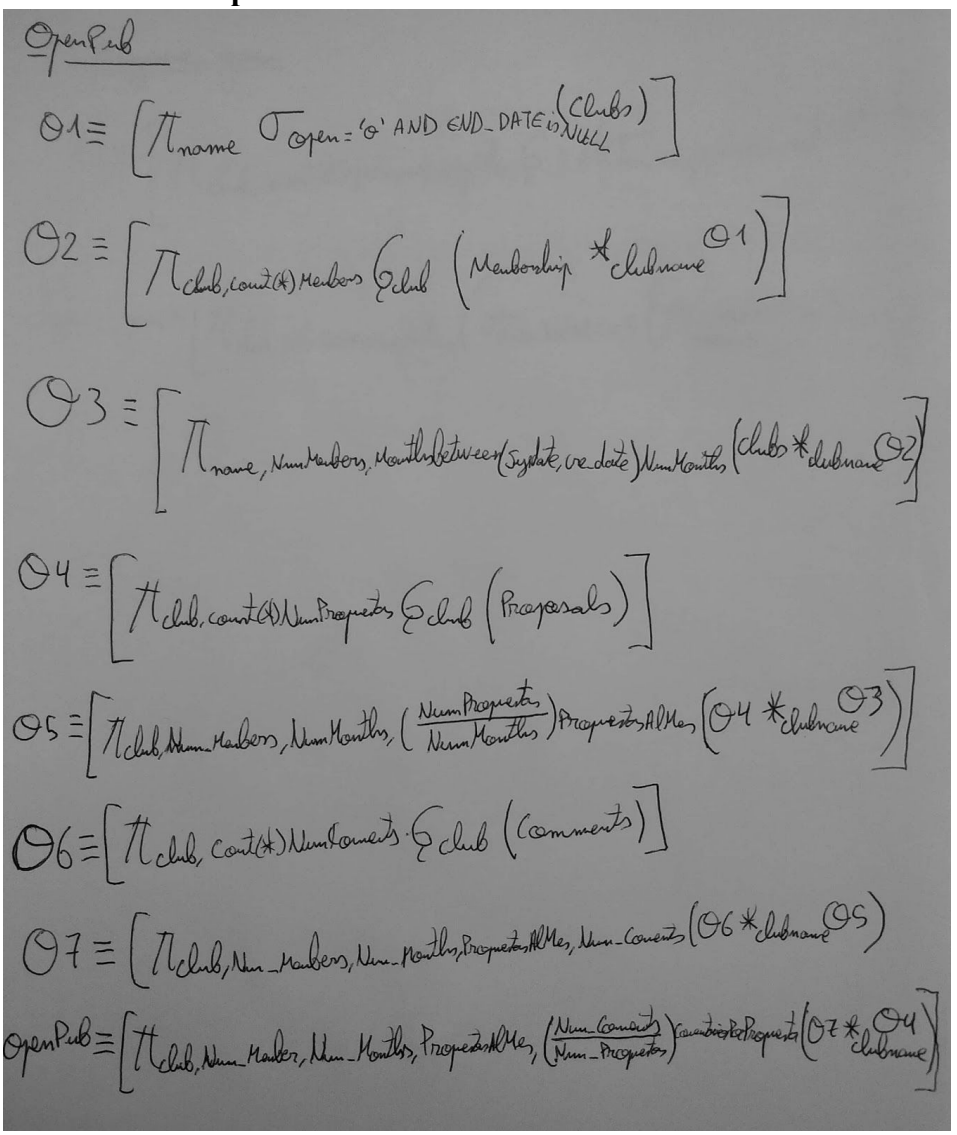
E. Diseño Externo

a. Descripción

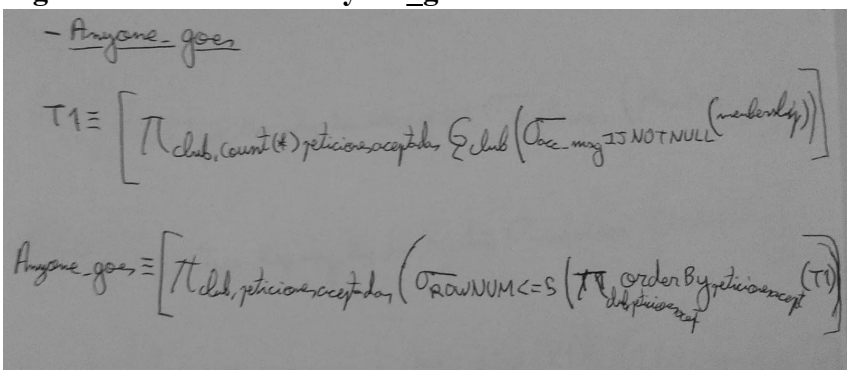
A continuación se ha creado un rol en la base de datos llamado *usuarioregistrado* a para el cual se le han otorgado privilegios de SELECT, es decir solamente para que pueda realizar consultas en las tablas de películas como en otras que tienen que ver con el rol propio de un usuario registrado en la aplicación. Para este rol se han creado tres vistas especiales para que pueda visualizar parámetros interesantes para el usuario. Una de ellas OpenPub visualiza los clubes abiertos que hay con cierta información sobre ellos como la cantidad de miembros, meses que lleva abierto, las propuestas que realizan al mes con el fin de ver lo activos que son y los comentarios por propuesta. Esto ha sido diseñado creando subvistas con el fin de modularizar la consulta, estas subvistas no son visibles para el rol, solo la de OpenPub que es para la que se le conceden los permisos de consulta. Las subvistas de OpenPub tienen el índice de Oi siendo i un número. En la segunda vista Anyone_goes se trata de ver los clubes que más peticiones aceptan, colocando en un top 5 aquellos que más aceptan. Para esta vista también se han realizado subvistas como en la anterior, con el índice de Ai. Por supuesto solo se le conceden permisos para la vista final. La tercera y última vista para este rol es la de Report, en la cual se ofrece información sobre el propio usuario que lo está viendo, para implementar esto se ha tenido que hacer uso de *user* que ofrece el nombre de usuario del sistema, dado que no había ningún usuario con nick FSDB138 no se ha podido llevar a cabo su prueba, pero funcionaria sin problema. Esta vista obtiene información de todos los clubes en los que ha tenido relación el usuario que está visualizandola como también en los que ha sido candidato para su acceso. A continuación se tocaran los temas de diseño en álgebra relacional de las consultas que se encuentran en las vistas así como su implementación en SQL.

b. Diseño de las vistas

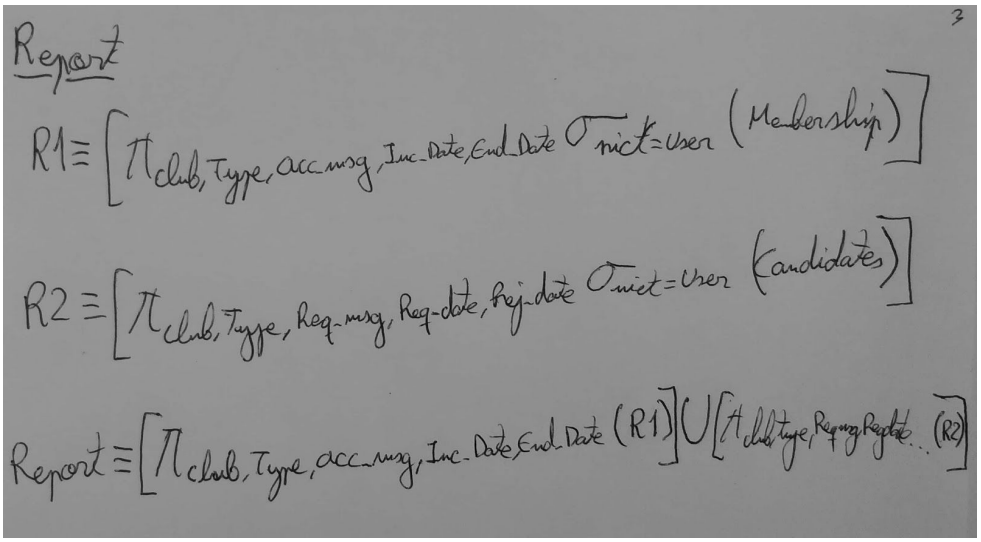
i. Algebra relacional de OpenPub



ii. Algebra relacional de Anyone_goes



iii. Algebra relacional de Report



c. Implementación en SQL

Recomendamos hacer zoom en la imagen dada la longitud del código, en caso de dificultades para su lectura adjuntamos el fichero sql 04-a-usuarioregistrado.sql

```
-- 04 DISEÑO EXTERNO A: USUARIO REGISTRADO
-- Creamos el rol de usuario registrado
CREATE ROLE usuarioregistrado;
--Le damos los privilegios necesarios al rol creado de las tablas relativas a preliculas y especificas para su rol
GRANT select on movies to usuarioregistrado;
GRANT SELECT on stars to usuarioregistrado;
GRANT SELECT on casts to usuarioregistrado;
GRANT SELECT on keywords to usuarioregistrado;
GRANT SELECT on genres_movies to usuarioregistrado;
GRANT SELECT on clubs to usuarioregistrado;
GRANT SELECT on proposals to usuarioregistrado;
GRANT SELECT on profiles to usuarioregistrado;
------------------------------------------ CREACION DE VISTAS-------------------------
-- SUBVISTAS PARA OPENHUB
CREATE OR REPLACE VIEW O1 AS(
-- Obtenemos todos los clubs que estan abiertos OPEN      CASILLA DEFINITIVA 1
    SELECT name  FROM CLUBS WHERE OPEN='O' AND END_DATE IS NULL
)WITH READ ONLY;

CREATE OR REPLACE VIEW O2 AS(
-- Obtenemos por CLUB la cantidad de miembros    CASILLA DEFINITIVA 2
    SELECT CLUB, COUNT(*) AS NUM_MEMBERS
    FROM MEMBERSHIP INNER JOIN O1
    ON CLUB=O1.NAME
    GROUP BY CLUB
)WITH READ ONLY;

CREATE OR REPLACE VIEW O3 AS(
-- Obtenemos los meses de actividad de cada club  CASILLA DEFINITIVA 3
    SELECT NAME,NUM_MEMBERS, MONTHS_BETWEEN(SYSDATE, cre_date) as NUM_MONTHS
    FROM CLUBS INNER JOIN O2
    ON NAME=O2.CLUB
)WITH READ ONLY;

CREATE OR REPLACE VIEW O4 AS( -- Obtenemos el total de propuestas por club
    SELECT CLUB, COUNT(*) AS NUM_PROPUESTAS
    FROM PROPOSALS
    GROUP BY CLUB
)WITH READ ONLY;

CREATE OR REPLACE VIEW O5 AS( -- Obtenemos PROPUESTAS/MES  CASILLA DEFINITIVA 4
    SELECT O4.CLUB, NUM_MEMBERS,NUM_MONTHS,(O4.NUM_PROPUESTAS/O3.NUM_MONTHS) AS
 PROPUESTASALMES
    FROM O4 INNER JOIN O3
    ON O4.CLUB = O3.NAME
)WITH READ ONLY;

CREATE OR REPLACE VIEW O6 AS( -- Obtenemos el total de comentarios por club
    SELECT CLUB, COUNT(*) AS NUM_COMMENTS
    FROM COMMENTS
    GROUP BY CLUB
)WITH READ ONLY;

CREATE OR REPLACE VIEW O7 AS(
-- Obtenemos el total de comentarios/propuesta   CASILLA DEFINITIVA 5
    SELECT O6.CLUB,NUM_MEMBERS,NUM_MONTHS,PROPUESTASALMES,NUM_COMMENTS
    FROM O6 INNER JOIN O5
    ON O6.CLUB = O5.CLUB
)WITH READ ONLY;

-- NOTA: Se van haciendo inner joins consecutivamente para arrastrar las columnas y que siempre tenga coherencia por el club
-- NOTA: Creamos la vista OpenPub, que estara compuesta de otras vistas para dividir el proceso, el usuario final solo tendra privilegios de consulta sobre la de OpenPub
CREATE OR REPLACE VIEW OpenPub AS( -- FUNCIONANDO PERFECTAMENTE
SELECT O7.CLUB, O7.NUM_MEMBERS, O7.NUM_MONTHS, O7.PROPUESTASALMES,(O7.NUM_COMMENTS
/O4.NUM_PROPUESTAS) AS COMENTARIOSPORPROPUESTA
FROM O7 INNER JOIN O4
ON O7.CLUB=O4.CLUB
)WITH READ ONLY;

-- Le otorgamos al rol los permisos para OpenPub
GRANT SELECT on OpenPub to usuarioregistrado;

---------- VISTA DE ANYONE_GOES-----------------------------------------------
CREATE OR REPLACE VIEW A1 AS (
-- VISTA PARA OBTENER LOS RESULTADOS DE TODOS LOS CLUBS QUE ACEPTAN PETICIONES JUNTO AL NUMERO DE PETICIONES QUE ACEPTAN
SELECT CLUB, COUNT(*) AS PETICIONESACEPTADAS
FROM MEMBERSHIP
WHERE ACC_MSG IS NOT NULL
GROUP BY CLUB
)WITH READ ONLY;

CREATE OR REPLACE VIEW Anyone_goes AS ( -- AQUI OBTENEMOS EL ORDEN
SELECT  CLUB, PETICIONESACEPTADAS
FROM (SELECT * FROM A1 ORDER BY  PETICIONESACEPTADAS DESC)
WHERE ROWNUM<=5
)WITH READ ONLY;

-- le damos al rol los permisos de consulta de anyone_goes
GRANT SELECT ON Anyone_goes TO usuarioregistrado;

--- REPORT DEL USUARIO ACTUAL ------------------------------------
-- SUBVISTAS PARA HACER EL REPORT
CREATE OR REPLACE VIEW R1 AS(
-- COGEMOS INFO DE MEMBERSHIP DE LOS CLUBES A LOS QUE PERTENECE O HA PERTENECIDO EN ALGUN MOMENTO
SELECT CLUB, TYPE, ACC_MSG, INC_DATE, END_DATE
FROM MEMBERSHIP
WHERE NICK=(SELECT USER FROM DUAL)
)WITH READ ONLY;

CREATE OR REPLACE VIEW R2 AS(
-- COGEMOS INFO DE CANDIDATES PARA OBTENER LOS CLUBS A LOS CUALES HA PRESENTADO CANDIDATURA
SELECT CLUB, TYPE, REQ_MSG, REQ_DATE, REJ_DATE
FROM CANDIDATES
WHERE NICK=(SELECT USER FROM DUAL)
)WITH READ ONLY;

-- VISTA (MEMBERSHIP) REPORT
CREATE OR REPLACE VIEW REPORT AS(
-- Unimos la informacion para darle al usuario toda su información de clubes a los que pertenece o ha pertenecido así como sus candidaturas
SELECT CLUB, TYPE, ACC_MSG, INC_DATE, END_DATE FROM R1
UNION
SELECT CLUB, TYPE, REQ_MSG, REQ_DATE, REJ_DATE FROM R2
)WITH READ ONLY;

--Le concedemos los permisos de consulta sobre la vista
GRANT SELECT ON REPORT TO usuarioregistrado;
```

F. Disparadores

Para cada disparador resuelto se debe incluir una subsección que contenga:

- Descripción del diseño: Tabla a la que está asociado, Evento o eventos en los que se dispara, Temporalidad (antes, después o en vez de), Granularidad (por fila o sentencia), Condición (si la tiene) y Acción (descripción en lenguaje natural).
    - Aplicattion: para realizar este disparador hemos creado una vista para juntar los candidatos que tiene cada club, con dichos clubs. A continuación, ponemos que cuando se inserte un nuevo candidato en la tabla candidates, se inserte también en la vista anteriormente creada. Ahora creamos el disparador principal que comprueba si el nuevo candidato insertado, se inserta en un club con privacidad 'C' (close), en ese caso, le añadirá al candidato la fecha actual de rechazo y en el mensaje de rechazo añadirá "Club con privacidad cerrada".
    - Overwrite: creamos el disparador en el que cuando se inserte un nuevo comentario, compruebe si ya hay un antiguo comentario escrito por el mismo nick, y en ese caso sobreescribimos el antiguo.
- Código (en SQL)

```
-- 05 Disparadores A
--Vista para juntar club y candidates
CREATE OR REPLACE VIEW T7 AS(
SELECT candidates.nick, candidates.club, candidates.member,
candidates.type, candidates.req_date, candidates.req_msg,
candidates.rej_date, candidates.rej_msg, clubs.open
FROM candidates INNER JOIN clubs
    ON candidates.club=clubs.name
);

CREATE OR REPLACE TRIGGER llenar_T7
INSTEAD OF INSERT ON T7
FOR EACH ROW
BEGIN
INSERT INTO T7(nick, club, member, type, req_date, req_msg, rej_date, rej_msg, open)
VALUES (candidates.new.nick, candidates.new.club, candidates.new.member,
 candidates.new.type, candidates.new.req_date, candidates.new.req_msg,
 candidates.new.rej_date, candidates.new.rej_msg, club.open);
END;

-- Disparador Application
CREATE OR REPLACE TRIGGER application
INSTEAD OF INSERT ON T7-- SE INSERTA LA INFORMACION DESPUES DE TENERLA EN CLUBS
--INSTEAD OF INSERT ON T7
FOR EACH ROW
BEGIN
IF :T7.open = 'C'
THEN
    INSERT INTO candidates(nick, club, member, type, req_date, req_msg, rej_date, rej_msg)
    VALUES
 (:new.nick, :new.club, :new.member, :new.type, :new.req_date, :new.req_msg, current_date, "
Club con privacidad cerrada");
END IF;
END;
```

```
-- 05 Disparador Overwrite B
CREATE OR REPLACE TRIGGER overwrite
BEFORE INSERT ON comments
-- SE INSERTA LA INFORMACION DESPUES DE TENERLA EN CLUBS
FOR EACH ROW
BEGIN
IF :new.nick IS old.nick
THEN
    INSERT INTO comments(old.club, old.nick, old.msg_date,
    old.title, old.director, old.subject, old.msg, old,valoration)
    VALUES (new.club, new.nick, new.msg_date, new.title,
    new.director, new.subject, new.msg, new,valoration);
END IF;
END;
```

G. Conclusiones

En primer lugar, debéis defender el resultado que hayáis alcanzado, haciendo hincapié en la cobertura semántica y potencia de vuestra implementación.

Después, comentad vuestro desempeño en estas dos primeras prácticas: esfuerzo requerido, organización de vuestro equipo de trabajo, progreso en vuestros conocimientos, etc. También podéis proponer mejoras para otros años (tamaño del problema, elementos que se piden, valoración, plazos, material de apoyo, etc).

Como primer punto, comentar que estamos orgulloso del trabajo realizado ya que hemos conseguido que funcione una gran parte de los apartados necesarios, y aquellos que no hemos conseguido son por cuestiones puntuales en las que pese a poner todo el esfuerzo y tiempo necesario para poder conseguirlo no ha sido posible lamentablemente. A continuación comentar que gracias a esta práctica hemos adquirido una gran cantidad de conocimientos relacionados con el SQL, como la correcta realización de las consultas, apoyándose en el diseño de hacer varias subconsultas (WITH T1 AS…), las diferencias entre las vistas y las consultas a la hora de aplicarlas al SQL… En resumen de este punto comentar que creemos que hemos realizado un buen trabajo y que se ha quedado una muy buena implementación.
Para continuar, nos gustaría que hemos puesto un gran esfuerzo y mucho tiempo en la realización de esta práctica, además teniendo en cuenta el inconveniente principal de las dificultades que proporciona la situación de cuarentena que estamos viviendo ya que nos ha provocado problemas de comunicación, y de rendimiento en el trabajo debido a que el aula virtual nos ha dado bastantes fallos.
Como mejoras, nos gustaría proponer que hubiera una menos cantidad de trabajo, aunque no sabemos muy bien si la percepción que hemos tenido de que fuera muy largo ha venido dada por las dificultades que hemos tenido y por la actual situación que estamos viviendo. Por otro lado también nos gustaría comentar que en el grafo proporcionado como solución de la primera práctica tenía fallos respecto al modelo sql de tablas proporcionado también, ya que, por ejemplo, había tablas en las que el orden de los atributos eran diferentes, esto supone una dificultad extra a la hora de insertar datos ya que como no aprecies a que es debido el fallo, puedes dedicar una gran cantidad de tiempo a buscar el error sin saber que es por eso, ya que en el sql no te especifica bien el error.
Para terminar, nos gustaría concluir con que hemos disfrutado haciendo la práctica, pero se nos ha hecho un poco larga debido, seguramente a la situación actual.
**Los archivos sql con los scripts de todo lo implementado se adjuntan en el zip.