TEMA 4: FIREBIRD, LENGUAJE DE MANIPULACIÓN DE DATOS

FIREBIRD: LENGUAJE DE MANIPULACIÓN DE DATOS

1.- CONCEPTOS BÁSICOS.

En Firebird la única forma de acceder a los datos es realizando una consulta. Una consulta es una sentencia SQL que es enviada al servidor. Una sentencia SQL es una sentencia consistente en palabras claves, frases y cláusulas con una forma determinada. Todas las operaciones de creación y modificación de metadatos, así como el acceso a los datos se realizan mediante consultas.

Una consulta de DML especifica un conjunto de datos y una o mas operaciones a ser realizadas sobre ellos. Además se pueden usar expresiones para transformar los datos cuando se devuelve, almacena o se busca sobre los conjuntos de datos.

Una consulta DML define una colección de elementos de datos situados en orden de izquierda a derecha en una o más columnas, conocidas como conjuntos. Un conjunto puede estar formado por una o varias filas. Una tabla es en sí un conjunto cuya especificación completa está en las tablas de sistema. El servidor de base de datos tiene sus propias consultas internas sobre las tablas de sistema para extraer los metadatos cuando son necesarios por una consulta de cliente.

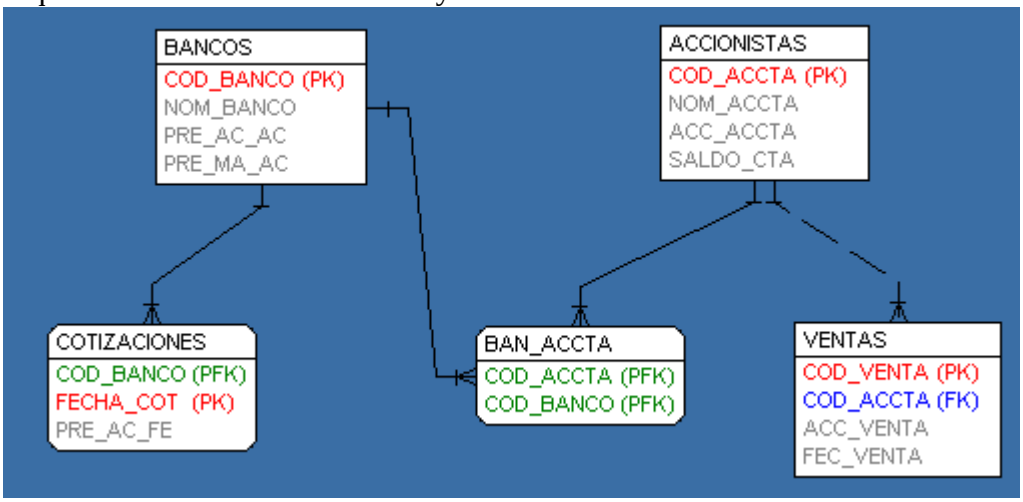
El uso más común de las consultas es el de obtener datos para mostrar a los usuarios. Esto se realiza mediante la sentencia *SELECT*. En esta sentencia se obtiene un conjunto resultado que se llama también como dataset o recordset.

En los conjuntos se debe tener en cuenta dos conceptos: cardinalidad y grado. La cardinalidad es el número de filas en el conjunto. Además se puede hablar de cardinalidad de una fila o cardinalidad de un valor clave, refiriéndose a la posición de una fila dentro de un conjunto de salida ordenado. El grado es el número de columnas en un conjunto.Podemos encontrar también el grado de una columna como la posición de la columna dentro del conjunto de columnas en el orden de izquierda a derecha.

En Firebird se utilizan los cursores como mecanismo para recorrer los conjuntos. Cuando realizamos una consulta se crean cursores en el lado del servidor, cursores transparentes al cliente. Se pueden definir también cursores en la parte del cliente para utilizarlos en PSQL y ESQL.

Por último es interesante destacar que es posible usar consultas SELECT dentro de otras, en una doble vertiente: subconsultas y tablas derivadas. Las subconsultas se usan en una columna de salida o en la cláusula WHERE. Las tablas derivadas se usan en la cláusula FROM.

En este tema desarrollaremos las sentencias de DML, para lo que usaremos el siguiente esquema de una base de datos muy elemental.



2.- SENTENCIA SELECT.

La sentencia SELECT es el método fundamental por el que los clientes obtienen conjuntos de datos de una base de datos. Su formato general es:

```
[WITH [RECURSIVE] <cte> [, <cte> …]]
SELECT
      [FIRST <m>] [SKIP <n>]
      [[ALL] | DISTINCT ]
      <lista columnas>
FROM <lista de conjuntos>
[WHERE <condicion busqueda>]
[GROUP BY <lista columnas agrupadas>]
[HAVING <condicion búsqueda>]
[UNION [DISTINCT | ALL] <expresión select>]
[PLAN <plan ejecución>]
[ORDER BY <lista columnas ordenacion>]
[ROWS <m> [TO <n>]]
[OFFSET]...[FETCH] .....
[FOR UPDATE [OF col1 [col2...]]
[WITH LOCK]]
[INTO  <variables>]
```

2.1.- Cláusula SELECT.

La cláusula select establece cual va a ser el conjunto resultado. Sobre este conjunto se pueden indicar los siguientes cuantificadores:

- ALL. Es el cuantificador por defecto. Indica que se incluirán todas las filas.
- DISTINCT: Este cuantificador elimina las filas duplicadas del conjunto de salida.
- FIRST m SKIP n: Este cuantificador permite devolver las primeras m filas del conjunto resultado ignorando las n iniciales. FIRST y SKIP se pueden usar juntas o cada una por independiente. Estos elementos son específicos de Firebird por lo que se aconseja usar en su lugar la cláusula ROWS … TO definidos en el estándar de SQL.

```
select all cod_banco from Ban_accta -- obtiene todos los códigos de bancos
select cod_banco from Ban_accta --equivalente a la anterior
select distinct cod_banco from Ban_accta -- devuelve los códigos de bancos sin repeticiones
select first 3 skip 2 distinct cod_banco from ban_accta -- devuelve tres códigos de bancos a partir de la
        segunda fila (se quitan las primeras 2) del conjunto resultado sin duplicados
```

- <lista de columnas> define las columnas que se devolverán en el conjunto resultado. Se sigue la siguiente sintaxis:

  <lista columnas> := <columna> [AS alias] [, <lista columnas]
  <columna> := {[alias_tabla.] columna_tabla |
  expresion | subconsulta-escalar | constante | variable_contexto |
  [alias_tabla.]*}

  En donde se tiene que:
  - AS alias. Con esto se le indica un identificador en tiempo de ejecución a una columna, es decir, se le da otro nombre a la columna. Se debe usar cuando la columna sea distinta de una columna_tabla para que tenga un identificador con el que referenciarse posteriormente. Tambien se utiliza con columna_tabla para renombrar un campo, por ejemplo, por aparecer en dos tablas de la cláusula FROM.

- alias_tabla: Es el nombre de un conjunto de datos origen o el nombre que se le asigna en la cláusula FROM.
- columna_tabla: Es un identificador de columna de una tabla o vista o un argumento de salida de un procedimiento almacenado.
- expresion: Cualquier expresión simple o compuesta. Podemos encontrar aquí expresiones formadas por funciones, constantes y subconsultas escalares (aquellas que sólo devuelven una fila con una única columna).
- constante: cualquier valor literal.
- variable_contexto: cualquier variable de sistema.
- Subconsulta-escalar: Es una sentencia select que devuelve una sola fila con una sola columna, es decir, un solo valor (un entero, una cadena, una fecha, …)
- *: Indica cualquier columna

select cod_banco from bancos – se selecciona una columna de la tabla bancos
select bancos.cod_banco from bancos – se selecciona una columna de la tabla bancos usando como alias el identificador de la tabla
select b.cod_banco from bancos b – se selecciona una columna de la tabla bancos usando el alia b indicado en la cláusula FROM
select cod_banco as codigo from bancos – se selecciona el código de banco y se cambia el identificador a codigo
select 'el banco es ' || nom_banco as nombre from bancos – una expresión como columna
select current_date as fecha, 'now' as ahora, 23, cast(cod_banco as char(10)) as codigo from bancos – se tiene como columnas una variable de sistema, un literal de fecha, un literal numérico y una expresión. Se tendrá una línea por cada fila en la tabla bancos.
*select * from bancos – todas las columnas de la tabla bancos.*
select bancos. from bancos – todas las columnas de la tabla bancos usando como alias el nombre de la tabla.*
select cod_banco, (select first 1 cod_accta from accionistas) from bancos – Se obtienen todos los cod_bancos en bancos junto con el codigo del primer accionista independiente del banco (una subconsulta).

2.2.- Cláusula FROM

Establece el/los conjuntos origen para la consulta. Se sigue la siguiente sintaxis:

FROM <conjunto> <joins>

<conjunto> := {<tabla>
| <vista>
| <procedimiento almacenado> [(<val> [, <val> ...]]
| <tabla derivada>
| <CTE>
| <tabla enlazada>} [[AS] <alias>]

<joins> := <join> [<join> …]

<join> := [<tipo join>]JOIN <conjunto> <join condición>
| {CROSS JOIN | , } <conjunto>
| NATURAL [<tipo join>] JOIN <conjunto>

<tipo join> := [INNER] |
{LEFT | RIGHT | FULL } [OUTER]

<join condicion> := ON condicion | USING (lista-columnas)

<tabla derivada> := (sentencia-select)

Como <conjunto> podemos tener tanto una tabla, una vista o un procedimiento almacenado que nos devuelva un conjunto de salida. Además podemos tener una tabla derivada, que no es más que una consulta SELECT, y una Expresión de Tabla Común (CTE).

Para reconocer una <conjunto> se utilizan los *ALIAS*. Un alias es un identificador que se asigna. Es obligatorio en el caso de las tablas derivadas y cuando usemos el mismo <conjunto> varias veces dentro de la cláusula FROM.

```
select b.cod_banco from bancos b – se indica la tabla bancos con el alias b y se usa el alias al indicar
                el campo cod_banco
select b.cod_banco from (select cod_banco from bancos) as b – se define una tabla derivada en la
                cláusula from y se le da el alias b
```

En la mayoría de los casos una tabla derivada puede eliminarse y establecerla como parte de la sentencia base obteniendo el mismo resultado. Sin embargo, hay situaciones en las que el resultado es distinto. Por ejemplo, si queremos obtener totales de dos tablas y mostrar los totales en una sola consulta, si no usamos tablas derivadas, obtendremos totales erróneos (aparecerán multiplicados por dos, tres, …, es decir, el numero de veces en las que se pueden unir las filas de una y otra tabla sobre el campo de unión).

<join> es el mecanismo por el se establece la unión entre dos conjuntos de datos. En Firebird están implementados todos los tipos establecidos en el estándar SQL, a saber:

- *CROSS JOIN:* Es el producto cartesiano de dos conjuntos. Como resultado se obtiene un conjunto con todas las filas de un conjunto unidas con todas las filas del otro conjunto.

```
select b.cod_banco,a.cod_accta
        from bancos b
                CROSS JOIN accionistas a
```

Se podría usar la sintaxis ‘,’ para separar las distintas tabla aunque no se aconsejo actualmente.

- *NATURAL JOIN*: Es la yunción natural. En la yunción natural se une el conjunto con las definidos anteriormente usando todos los campos que se llaman igual. La unión se realizaría con la condición (a.campo=b.campo). Hay que tener en cuenta que se unen por TODOS los campos que se llaman iguan por lo que puede ocurrir que obtengamos resultados que no esperamos (se han unido campos que no pensábamos que se iban a unir).

```
select *
    from bancos
            NATURAL JOIN ban_accta -- une las dos tablas por el campo cod_banco
```

- <join condición>. Se puede indicar dos mecanismos para unir un conjunto a los conjunos anteriores. Usando ON para indicar una condición explícita o mediante USING que establece la unión por un campo (a.campo=b.campo).

```
select * -- unimos bancos y ban_accionista por cod_banco. Condicion implícita
    from bancos b
```

```
        JOIN ban_accta ba USING (cod_banco)

select * -- equivalente a la anterior indicando la condición de forma explícita
    from bancos b
        JOIN ban_accta ba ON (a.cod_banco=ba.cod_banco)

select * -- indicamos una condición distinta
    from bancos b
        JOIN ban_accta ba USING (a.cod_banco>=ba.cod_banco)
```

- *<tipo join>:* Cuando unimos dos conjuntos se puede indicar como se incluyen las filas de cada conjunto en el conjunto resultado. Existen dos mecanismos INNER y OUTER.
    - *[INNER]*. Se unen los conjuntos y solo aparecen aquellas filas que cumplen la condición de join.

```
select b.nom_banco,ba.cod_accta
        from bancos b
            INNER JOIN ban_accta ba ON (ba.cod_banco=b.cod_banco)
                -– sólo aparecerán aquellas filas para bancos para las que existe
                una fila en ban_accta para el cod_banco
```

    - *{LEFT | RIGHT | FULL } [OUTER]*: Es equivalente a un INNER JOIN en donde aparecerán en el conjunto resultado además, todas las filas del primer conjunto (LEFT), del segundo conjunto (RIGHT) o de ambos (FULL) que no cumplían con la condición de unión. Aquellos campos de la otra tabla que no se unen, aparecerán como NULL.

```
select b.nom_banco,ba.cod_accta
    from bancos b
        LEFT OUTER JOIN ban_accta ba ON (ba.cod_banco=b.cod_banco)
            – aparecen las filas de bancos para las que existe una fila en ban_accta para
                el cod_banco. Además aparecen las filas de bancos para las que
                no se cumple la condición. En este caso cod_accta aparece como
                NULL
```

2.3.- Cláusula WHERE.

En esta cláusula se indican las condiciones que se deben cumplir en el conjunto resultado. Las condiciones se establecen a nivel de fila sobre los conjuntos origen, es decir, los indicados en la cláusula FROM, y nos determinarán cuales son las filas candidatas para formar parte del conjunto resultado. Posteriormente, estas filas pueden ser procesadas con lo indicado en las cláusulas ORDER BY y/o GROUP BY.

Tenemos la siguiente sintaxis:

WHERE <condicion busqueda>

<condicion busqueda> := <val> <operador> { <val> | (<subconsulta escalar>)}
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] CONTAINING <val>
| <val> [NOT] SIMILAR TO <patrón similar> [ESCAPE <carácter escape>]
| <val> [NOT] STARTING [WITH] <val>

| <val> [NOT] IN (<val> [, <val> …] | <subconsulta de lista>)
| <val> IS [NOT] NULL
| <val> IS [NOT] DISTINCT FROM <val>
| <val> IS [NOT] {TRUE|FALSE|UNKNOWN}
| <val> <operador> {ALL | SOME | ANY} (<subconsulta de lista>)
| EXISTS (<subconsulta>)
| SINGULAR (<subconsulta>)
| (<condicion busqueda>)
| NOT <condicion busqueda>
| < condicion busqueda > OR < condicion busqueda >
| < condicion busqueda > AND < condicion busqueda >
<operador> := {= | < | > | <= | >= | !< | !> | <> | !=}

Esta misma sintaxis es aplicable en la cláusula HAVING.

Como se puede observar aparecen tres tipos de subconsultas:
- subconsulta escalar: Es aquella consulta en la que se obtiene un valor escalar, es decir, una sóla fila con una única columna.
- subconsulta de lista: Es aquella consulta en la que se obtiene una lista, es decir, 0 o más filas con una sóla columna.
- subconsulta: Aquella consulta que devuelve 0 o más filas.

Todo lo indicado como <condicion búsqueda> son expresiones que se pueden consultar en el tema 2.

2.4.- Cláusula GROUP BY.

El conjunto obtenido por la cláusula WHERE se puede particionar para formar grupos que sumaricen (agregen) grupos según criterios. Sobre estos grupos se aplican expresiones de agregación, es decir, expresiones que contienen funciones de trabajo sobre múltiples valores, tales como totales, medias, cuenta de filas y valores máximos/mínimos.

Se tiene la siguiente sintaxis:
GROUP BY <lista columnas agrupadas>
<lista columnas agrupadas> = <columna> | <expresion> | <grado>

Para obtener los grupos se definen una o varias columnas agrupables. Para sumarizar se definirán una o varias columnas sumarizables (usando expresiones que involucren funciones de agregación). Para que no nos de ningún error se tiene que cumplir que todas las columnas agrupables aparezcan en GROUP BY y que el resto de columnas de la cláusula SELECT sean sumarizables.

```
select cod_banco, count(cod_accta) as num_accionistas
        from ban_accta
        group by cod_banco -- se tienen dos columnas resultado, una agrupable cod_banco y otra
                        sumarizable

select cod_Banco, extract(year from fecha_cot)as anio, extract(month from fecha_cot) as mes,
                        avg(pre_ac_fe) as cot_media
        from cotizaciones
        group by cod_Banco, extract(year from fecha_cot) , extract(month from fecha_cot) --
                        obtenemos la cotizacion media por banco, año y mes
```

Para establecer la expresión en la cláusula having se puede usar el grado (posición de la columna en la cláusula SELECT).

```
select cod_Banco, extract(year from fecha_cot)as anio, extract(month from fecha_cot) as mes,
                    avg(pre_ac_fe) as cot_media
    from cotizaciones
    group by cod_Banco, 2 , 3 -- obtenemos la cotizacion media por banco, año y mes
```

Como funciones agregativas tenemos COUNT (cuenta el numero de apariciones, filas), SUM (suma de los valores de la columna), AVG (mefia), MIN (mínimo), MAX (máximo) y LIST (cadena a partir de la concatenacion de los valores de una columna).

2.5.- Cláusula HAVING.

Esta cláusula permite establecer condiciones a nivel de los grupos obtenidos por una cláusula GROUP BY, es decir, cuales de estos grupos se tendran en cuenta para obtener el conjunto resultado.

Sintaxis:

HAVING <condicion búsqueda>

Se pueden establecer condiciones de búsqueda, siguiendo la sintaxis de la cláusula WHERE en la que aparecen funciones agregativas.

```
select cod_banco, count(cod_accta) as num_accionistas
                    from ban_accta
                    group by cod_banco
                     having count(cod_accta) > 3 – Obtiene los bancos y el número de accionistas
                                de aquellos bancos con mas de tres accionistas.

select cod_accta
            from ban_accta
            where cod_accta>100
            group by cod_accta
             having count(cod_banco)>2 – accionistas con código de accionista mayor de 100 y
                                que aparecen en más de 2 bancos
```

2.6.- Cláusula UNION

Es la implementación de la Unión del algebra relacional. Permite unir dos consultas para generar un único conjunto resultado. Para ello se tiene que cumplir que ambas generen conjuntos con idénticas columnas (número y tipo de datos).

Sintaxis:

UNION [DISTINCT | ALL]<expresión select>

Por defecto se eliminan aquellas filas duplicadas (DISTINCT) que existan en el conjunto resultado. Si queremos que aparezcan, se puede usar ALL.

```
select distinct cod_banco,'banco' as tipo
            from bancos
union all
select distinct cod_accta, 'accionista'
             from accionistas – se obtienen todos los códigos de banco y accionistas junto con su
                        tipo
```

2.7.- Cláusula PLAN

Esta cláusula permite establecer el plan de consulta. En general, es mejor dejar al optimizador generar este plan. Por ello, no vamos a entrar en más profundidad.

2.8.- Cláusula ORDER BY.

En esta cláusula se establece el orden en el que se generará el conjunto resultado.

Se usa la siguiente sintaxis:

ORDER BY <lista columnas ordenacion>
<lista columnas ordenación> =<columna> | <expresion> | <grado>
ASC | des
[NULLS LAST | NULLS FIRST]

Se pueden establecer dos criterios de ordenación: ascendente (ASC) o descendente (DESC). Por defecto se usará el criterio ascendente.

```
select b.nom_banco, b.cod_banco, a.nom_accta, a.cod_accta
        from bancos b
                inner join ban_accta ba on (ba.cod_banco=b.cod_banco)
                inner join accionistas a on (a.cod_accta=ba.cod_accta)
        order by nom_banco ASC, nom_accta DESC
```

Es posible indicar donde aparecerán los NULL: al principio (NULLS FIRST) o al final (NULLS LAST). Por defecto se tiene NULLS FIRST.

Como se observa por la sintaxis es posible ordenar, no sólo por un campo, sino también por una expresión o por un número (grado), que representa la posición del campo en la cláusula SELECT.

```
select b.nom_banco, b.cod_banco, a.nom_accta, a.cod_accta
        from bancos b
                inner join ban_accta ba on (ba.cod_banco=b.cod_banco)
                inner join accionistas a on (a.cod_accta=ba.cod_accta)
        order by nom_banco || nom_accta ASC -- se ordena por la concatenación de los nombres.

select b.nom_banco, b.cod_banco, a.nom_accta, a.cod_accta
        from bancos b
                inner join ban_accta ba on (ba.cod_banco=b.cod_banco)
                inner join accionistas a on (a.cod_accta=ba.cod_accta)
        order by 1 ASC, 3 DESC -- se ordena por nom_banco ASC y nom_accta DESC
```

A diferencia de GROUP BY las columnas por las que se ordenan no tienen por qué aparecer en la cláusula SELECT.

```
select cod_banco
        from bancos
        order by nom_banco desc  -- muestra los códigos de banco ordenados por nombre en orden
                descendente.
```

2.9.- Cláusula selección de filas

En firebird se establece varios mecanismos para devolver un conjunto de filas concreto del conjunto resultado. Estos mecanismos son:

- FIRST … SKIP de la cláusula SELECT. Mecanismo propio de firebird ya visto anteriormente.
- Cláusula ROWS … TO …. Es un mecanismo propio de firebird.
  Su sintaxis sería ***[ROWS <expr1> [TO <expr2>]]***. Permite devolver las filas del conjunto resultado a partir de expr1 hasta expr2.

```
select cod_banco
    from bancos
    rows 2 to 5 – devuelve cuatro codigos de banco, los que ocupan la posición de 2 a 5
```

- Cláusula OFFSET … FETCH. Es un mecanismo establecido en SQL-2008.
  Su sintaxis sería:

  [OFFSET <valor_simple> { ROW | ROWS }]
  [FETCH { FIRST | NEXT } [<valor_simple>] { ROW | ROWS } ONLY]
  Mediante OFFSET se indica el numero de filas a saltar. Con FETCH se indica el numero de filas a devolver. Pueden devolverse las primeras/siguientes (FIRST/NEXT) filas.

```
select cod_banco
        from bancos
offset 2 rows fetch next 3 rows only
```

En ningún caso se puede se pueden usar dos mecanismos a la vez en la misma sentencia select.

2.10.- Cláusula FOR UPDATE.

No se utiliza normalmente, salvo en el contexto de sentencia usada para definir un cursor con nombre.

2.11.- Expresiones de tabla común

A partir de la versión 2.1 está disponible las CTE (common table expression). Un CTE es como una vista creada localmente en la consulta principal. El sistema la trata como una tabla derivada en la que no se almacena datos intermedios.

El uso de CTE permite la definicion de consultas recursivas. Hay que tener en cuenta que la única forma de hacer recursividad hasta el momento era a traves de procedimientos almacenados. En comparación con éstos, los CTE reducen el uso de memoria y CPU.

En caso de definir una CTE recursiva el sistema sigue los siguientes pasos.

- Se comienza la ejecución desde un miembro no recursivo.
- Para cada fila evaluada, se inicia la ejecución de cada miembro recursivo, usando los valores actuales de la fila externa como parámetros.
- Si no se devuelven filas al ejecutar un miembro recursivo, se vuelve al nivel anterior de ejecución volviendo a la siguiente fila del bucle exterior.

La definición de la sentencia Select, incluyendo las CTE, sería

[Definicion_CTE] select .....

En donde se tiene que:

Definicion_CTE:

WITH [RECURSIVE] CTE [, CTE]

CTE:

<Nombre_alias> [(campo[,campo])]
AS '(' expresión_select
| expresion_select UNION[ALL] expresion_select
')'

Como ejemplo podríamos tener la siguiente consulta:

```
with ban as
    (select cod_banco
          from bancos
          where nom_banco similar to '[b]%'     -- expresión de tabla
    )
 select * from ban     --sentencia principal
```

En una sentencia select se podrán usar CTE con las siguientes especificaciones (en caso de ser CTE no recursivas):

- Se pueden definir múltiples CTE en la misma sentencia select.
- Cualquier cláusula legal en una sentencia select se podrá usar en una expresión de tabla.
- Las expresiones de tabla pueden hacer referencia a otras expresiones de tabla.
- No pueden existir referencias cruzadas (bucles) entre expresiones que usan otras.
- Se pueden usar expresiones de tabla en cualquier lugar parte de la consulta principal u otra expresión de tabla.
- La misma expresión de tabla puede ser usada más de una vez en la consulta principal.
- Las expresiones de tabla (como subconsultas) pueden ser usadas en sentencias INSERT, UPDATE y DELETE.
- Las expresiones de tabla se pueden usar en PSQL.

Por otro lado, si queremos usar expresiones de tabla recursivas se tendrán que aplicar las siguientes reglas:

- Una CTE recursiva será aquella que hace una referencia a sí misma.
- Una CTE recursiva está formada por la unión de miembros recursivos y no recursivos con:
  - Debe existir al menos un miembro no recursivo.
  - Los miembros no recursivos se colocan los primeros en la UNION.
  - Los diferentes miembros se separarán mediante la sintaxis:

    Miembro no recursivo (base)
    UNION [ALL | DISTINCT]
    Miembro no recursivo (base)
    UNION [ALL | DISTINCT]
    Miembro no recursivo (base)
    UNION ALL
    Miembro recursivo
    UNION ALL
    Miembro recursivo
- No puede haber bucles entre referencias de CTE

- En los miembros recursivos no se permiten cláusulas de agregación (DISTINCT, GROUP BY y HAVING) ni funciones de agregación (SUM, COUNT, etc)
- Un miembro recursivo puede hacer referencia a sí mismo una y sólo una vez en la cláusula from.
- Un miembro recursivo no puede aparecer en un OUTER JOIN.

Para ver las expresiones de tabla recursivas usaré la siguiente tabla

```
CREATE TABLE UBICACIONES (
   COD_UBICACION DCODARTICULO NOT NULL,
   DESCRIPCION DCAD30,
   DENTRO_DE DCODARTICULO);


ALTER TABLE UBICACIONES ADD PRIMARY KEY (COD_UBICACION);

ALTER TABLE UBICACIONES
                ADD CONSTRAINT FK_UBICACIONES
                      FOREIGN KEY (DENTRO_DE)
                      REFERENCES UBICACIONES(COD_UBICACION)
                      ON UPDATE CASCADE;
```

Esta tabla UBICACIONES tiene la columna DENTRO_DE que representa una referencia al código de ubicación dentro de la que se encuentra, que es a su vez otra ubicación. Aquellas ubicaciones que no están dentro de otras contendrán en DENTRO_DE el valor NULL.

Si quisieramos listar cada ubicación con la lista de ubicaciones dentro de la que está podríamos usar la siguiente expresión:

```
WITH RECURSIVE UBI AS
  (SELECT COD_UBICACION, DESCRIPCION, DENTRO_DE,
             cast('' as VARCHAR(300)) AS DESC_DENTRO_DE,
             cast(cod_ubicacion as varchar(100)) as cod_dentro_de
          FROM UBICACIONES
          WHERE DENTRO_DE IS NULL
   UNION ALL
   SELECT U.COD_UBICACION, U.DESCRIPCION,U.DENTRO_DE,
            ub.descripcion ||' - ' || Ub.desc_dentro_de,
            cast(u.cod_ubicacion as varchar(10)) || ' ' || ub.cod_dentro_de
         FROM UBICACIONES U
             INNER JOIN UBI UB ON (U.dentro_de=UB.cod_ubicacion)
         WHERE U.DENTRO_DE IS NOT NULL
  )
 SELECT * FROM UBI
```

Es importante que el campo que rellenamos de forma recursiva tenga una longitud apropiada. En este campo DESC_DENTRO_DE lo he definido varchar(300) para que en el quepan las distintas descripciones de las ubicaciones en las que se encuentra.

Si partimos de los siguientes datos:

COD_UBICACIOI	DESCRIPCION	DENTRO_DE
1	ALMACEN	Null
2	PLANTA BAJA	1
3	PRIMERA PLANTA	1
4	PB-A1	2
5	PB-A2	2
6	PB-A1-EST1	4
7	PP-S1	3

Obtendríamos el siguiente conjunto resultado al ejecutar la cte recursiva:

COD_UBICACION	DESCRIPCION	DENTRO_DE	DESC_DENTRO_DE	COD_DENTRO_DE
1	ALMACEN	Null		1
2	PLANTA BAJA	1	ALMACEN -	2 1
4	PB-A1	2	PLANTA BAJA - ALMACEN -	4 2 1
6	PB-A1-EST1	4	PB-A1 - PLANTA BAJA - ALMACEN -	6 4 2 1
5	PB-A2	2	PLANTA BAJA - ALMACEN -	5 2 1
3	PRIMERA PLANTA	1	ALMACEN -	3 1
7	PP-S1	3	PRIMERA PLANTA - ALMACEN -	7 3 1

3.- SENTENCIA INSERT

La sentencia INSERT es usada para insertar nuevas filas en una única tabla. Esta sentencia puede operar también sobre vistas, siempre que cumplan ciertas condiciones (se hayan definido trigger que añadan esos valores en las tablas sobre las que están definida la vista).

Para indicar los valores a insertar se pueden usar dos mecanismos: indicar valores constantes o indicar una subconsulta que nos los rellene.

INSERT INTO tabla | vista (<lista columnas>)
VALUES (<lista de valores>)
[RETURNING <lista_columnas> [INTO <lista variables>]]

INSERT INTO tabla | vista (<lista columnas>)
SELECT <lista valores> FROM …..
[RETURNING <lista_columnas> [INTO <lista variables>]]

```
INSERT INTO bancos (cod_banco,nom_banco)
    VALUES (1000, 'BANCO NUEVO') -- inserta un nuevo banco indicando valores para
                cod_banco y nom_banco. Inserta una sola fila

INSERT INTO cotizaciones (cod_banco, fecha_cot, pre_ac_fe)
    select cod_banco, current_date, 10 from bancos -- inserta una nueva cotización por cada
                banco existente, con fecha el dia de hoy y valor 10. Inserta múltiples filas
```

En ambas sintaxis se tiene una cláusula RETURNING usada tando en DSQL como en PSQL para devolver valores de columnas en la fila insertada (en este caso no se permite que se inserten más de una fila). Por ejemplo puede usarse para devolver el valor de una clave autonumérica generada dentro de un trigger.

```
INSERT INTO bancos (cod_banco,nom_banco)
    select first 1 cod_banco, current_date, 10 from bancos
    RETURNING cod_banco INTO :codigo-- inserta una sola fila de cotizacion y guardo ese valor
                en una variable. Esta sentencia debería estar dentro de un procedimiento
                almacenado o trigger.
```

El comportamiento de una sentencia INSERT puede verse afectado por los siguientes casos:

- Se incluye en la sentencia INSERT un campo calculado. En este caso la sentencia INSERT genera una excepción.
- Columnas por defecto. Si tenemos una sentencia INSERT para una tabla en la que no se indican todos los campos de la misma, se rellenarán a NULL salvo que se haya indicado un valor DEFAULT en la definición de la columna, valor que se asignará en este caso. Si a alguna columna se le asigna el valor NULL y la columna está definida como no NULL se generará una excepción.

  ```
  INSERT INTO bancos (cod_banco,nom_banco)
      VALUES (1000, 'BANCO NUEVO') -- el campo pre_ac_ac al no tener valor por defecto
                se rellena con NULL
  ```
- En los triggers BEFORE INSERT. Si tenemos definido un trigger para antes de insertar la fila, por ejemplo, para dar un nuevo valor a una columna autonumérico, la columna que se rellene en el trigger no debe aparecer en la sentencia INSERT.

4.- SENTENCIA UPDATE

La sentencia UPDATE es usada para cambiar los valores en columnas existentes de las tablas. La sentencia UPDATE se aplica sobre una única tabla o vista (siempre que sea actualizable).

UPDATE tabla | vista [[AS] alias]
SET columna= <expresion> [,columna=<expresión> ……]
[WHERE {<condicion búsqueda> | CURRENT OF cursor}]
[ORDER BY <element_ordenacion>]
[ROWS <m> [TO <n>]]
[RETURNING <lista_columnas> [INTO <lista variables>]]

La sentencia UPDATE permite actualizar una o varias columas indicadas en la cláusula SET, con valores obtenidos a partir expresiones:

```
UPDATE bancos SET
        pre_ac_ac=0,
        pre_ma_ac=0 -- establece el precio actual y el precio máximo a 0 a todos los bancos.

UPDATE bancos SET
        pre_ac_ac=0,
        pre_ma_ac=(select max(pre_ac_fe)
                                from cotizacion c
                                where c.cod_banco=bancos.cod_banco) -- establece el precio actual
                                0 y el precio máximo a la máxima cotización del banco que estemos
                                actualizando. La actualización se realiza para todos los bancos.
```

Se pueden indicar criterios de filtrado para las filas a actualizar indicando condiciones de forma semejante a como se indican en la cláusula WHERE de una sentencia SELECT.

```
UPDATE bancos SET
        pre_ac_ac=0,
        pre_ma_ac=0
      WHERE cod_banco< 100 -- establece el precio actual y el precio máximo a 0 para bancos con
                        código menor de 100.

UPDATE bancos SET
        pre_ac_ac=0,
        pre_ma_ac=(select max(pre_ac_fe)
                                from cotizacion c
                                where c.cod_banco=bancos.cod_banco)
        where cod_banco between 10 a 100 -- establece el precio actual 0 y el precio máximo a la
                                máxima cotización del banco que estemos actualizando, para
                                bancos con código de banco entre 10 y 100.
```

También es posible actualizar la fila actual de un cursor (actualización posicionada) usando la sintaxis WHERE CURRENT OF.

Se puede limitar el número de filas a actualizar mediante la cláusula ROWS. Si solo se indica <m> se actualizaran las m primeras filas. Si aparece <m> y <n> se actualizarán las filas entre la m y la n.

También se puede fijar el orden de las filas que se actualizarán mediante ORDER BY.

Por último, y al igual que con la sentencia INSERT, se puede indicar una cláusula RETURNING para devolver algún valor de la única fila modificada.

5.- SENTENCIA DELETE

La sentencia DELETE permite borrar una ovarias filas de una única tabla.

DELETE FROM tabla [[AS] alias]
[WHERE {<condicion búsqueda> | CURRENT OF cursor}]
[ORDER BY <lista_ordenacion>]
[ROWS <m> [TO <n>]]
[RETURNING <lista_columnas> [INTO <lista variables>]]

Si no se indica ninguna condición se borrarán todas las filas.

```
delete from bancos -- borra todos los bancos
```

Se le puede indicar una condición de filtrado de filas para restringir los registros a borrar:

```
delete from bancos
    where cod_banco>100
```

También se puede borrar la fila actual apuntada por el cursor usando la sintaxis WHERE CURRENT.

Se puede limitar el número de filas a borrar mediante la cláusula ROWS. Si solo se indica <m> se borran las m primeras filas. Si aparece <m> y <n> se actualizarán las filas entre la m y la n.

También se puede fijar el orden de las filas que se borrarán mediante ORDER BY.

Por último es posible devolver valores de columnas de la fila borrada usando RETURNING.

6.- SENTENCIA UPDATE OR INSERT.

Esta sentencia permite actualizar una fila o insertarla en caso de no existir. Esta disponible desde Firebird 2.1.

UPDATE OR INSERT INTO tabla | vista [(<lista columnas>)]
VALUES (<lista valores>)
[MATCHING (<lista columnas>)]
[RETURNING <lista columnas> [INTO <lista variables>]]

Actua de forma semejante a una sentencia INSERT, en la que se comprueba los campos indicados en la cláusula MATCHING. Si los valores a insertar ya se encuentran en la tabla, se aplica el UPDATE sobre el resto de campos. Si no se encuentran se hace un INSERT.

Si no se indica cláusula MATCHING la tabla debe tener una clave principal, de esta forma se hace la comprobación sobre las columnas de definición de la clave principal.

```
UPDATE OR INSERT INTO bancos (cod_banco, nom_banco)
    VALUES (1000, 'banco modificado')
    MATCHING cod_banco -- se actualiza el banco 1000 cambiando el nombre del banco. Si no
            existe se inserta el banco.
```

Igual que con el resto de sentencias DML se puede devolver valores de columnas de la fila modificada/insertada mediante la cláusula RETURNING.

7.- SENTENCIA MERGE.

Esta sentencia permite insertar/actualizar de forma masiva en una tabla. Esta disponible desde Firebird 2.1.

Su sintaxis es:

```
MERGE
    INTO <tabla o vista> [ [AS] <alias> ]
    USING <tabla, vista o tabla derivada> [ [AS] <alias> ]
    ON <condicion>
    WHEN MATCHED THEN
        UPDATE SET <lista_asignaciones>
    WHEN NOT MATCHED THEN
        INSERT [ (lista_columnas)]
        VALUES (lista_valores)
```

Cuando se indica una sentencia MERGE se usa <condición> para unir las dos tablas usando un right join. Si la condición se cumple, se actualiza la fila usando la cláusula update. Si la condición no se cumple (no existe la fila) se inserta usando la cláusula insert.

Por ejemplo

```
MERGE INTO bancos b
    using accionistas a
    on (b.cod_banco=a.cod_banco)
    when matched then
        update set b.nom_banco='banco ' || a.nom_accta
    when not matched then
        insert (cod_banco,nom_banco)
            values (a.cod_accta,'banco ' || a.nom_accta)
```

8.- TRANSACCIONES

En un servidor de bases de datos cliente/servidor como Firebird, las aplicaciones cliente nunca tocan directamente los datos físicamente almacenados en las páginas de la base de datos. En su lugar, se establecen relaciones de petición-respuesta entre el cliente y el servidor dentro de una transacción.

En un sistema con múltiples clientes atacando a una misma base de datos de forma simulánea se pueden dar una serie de problemas:

- Perdida de actualizaciones: Ocurre cuando dos usuarios tienen la misma vista de un conjunto de datos. Actualiza el primero los valores. Después actualiza el segundo perdiéndose los cambios hechos por el primero.
- Lecturas sucias: Un usuario ve cambios en datos realizados por otro usuario sin que éste los haya confirmado.
- Lecturas no reproducibles: Un usuario está continuamente leyendo los mismos datos mientras que otro esta modificandolos. El primero, si está generando totales, obtendrá resultados no válidos en relación al momento en el que inició la consulta.
- Filas fantasmas: Un usuario ve filas pero no todas las que ha insertado otro usuario, sin que éste último las haya confirmado.
- Transacciones entrelazadas: Un usuario realiza cambios en una fila que afectan a otras filas en la misma u otras tablas siendo accedidas por otros usuarios. Da problemas cuando no hay mecanismos para secuenciar las operaciones realizadas por lo que el resultado es impredecible.

Para resolver estos problemas, Firebird aplica un modelo de gestión que aisla cada tarea dentro de un único contexto que previene de hacer definitivos los cambios en los datos si entran en conflicto con tareas hechas por otros usuarios.

Los problemas indicados anteriormente dieron lugar a estudios de los que salió el concepto ACID, es decir, los requerimientos basados en cuatro preceptos: Atomicitad (atomicity), consistencia (consistency), aislamiento (isolation) y durabilidad (durabilidad). Estos preceptos son los usados en el diseño de transacciones en los sistemas de bases de datos actuales.

- *Atomicidad*: Una transacción o unidad de trabajo se considera consistente en una colección de operaciones de transformación de datos. Para ser atómica, la transacción debe ser implementada para que se cumpla que se realizan todas las operaciones o ninguna, es decir, se toman todas las operaciones como un bloque y se aceptan o se deshacen como un todo.
- *Consistencia:* Se asume que las transacciones realizan transformaciones correctas del estado del sistema, esto es, una base de datos queda en un estado consistente tras una transacción, se haya confirmado o deshecho. El concepto de transacción asume que los programadores tienen mecanismos para establecer puntos de consistencia y validación.
- *Aislamiento:* Cuando una transacción está actualizando datos compartidos, el sistema debe hacerle creer que se está ejecutando de forma aislada, es decir, que el resto de transacciones se ejecutan antes de comenzar o después de que confirma. Mientras los datos están en transición entre el estado inicial consistente y el estado final consistente, podrían hacer inconsistente el estado de la base de datos. Los diferentes niveles de aislamiento establecen como actuar ante posibles problemas que puedan ocurrir.
- *Durabilidad:* Una vez que una transacción confirma, sus actualizaciones deben ser duraderas, es decir, el nuevo estado de todos los objetos expuestos a las otras transacciones tras la confirmación deberá mantenerse y ser irreversible, salvo problemas de tipo hardware o rotura de software en ejecución.

En base a lo anterior se llega al concepto de **transacción**, que no es más que toda la "conversación" establecida entre el servidor y el cliente. Cada transacción tiene un contexto único que está aislado del resto de transacciones en base a unas normas establecidas. Las reglas para el contexto de transacción las establece la aplicación cliente en el momento en el que la inicia indicando una serie de parámetros. Una transacción empieza cuando el cliente manda la orden de comenzar la transacción (begin tran, set transaction) y recibe un manejador de transación por parte del servidor, y permanece activa hasta que el cliente la confirma (commit) o la deshace (rollback).

Toda operación solicitada por el cliente ocurre dentro de la transacción activa. Una transacción podrá englobar una o varias peticiones del cliente junto con las respuestas del servidor. En una transacción podrán aparecer peticiones a más de una base de datos. De igual forma todas las sentencias de DML se deben encontrar dentro de una transacción. De esta forma llegamos a un punto en el que cliente y servidor actúan como:

- *cliente*: Inicia todas las transacciones. Una vez que se incia una transacción el cliente es el responsable de enviar las peticiones de lectura y escritura así como de completar cada transacción que inicia. Una conexión de un cliente puede tener varias transacciones activas en un momento dado.
- *servidor*: El servidor se encarga de todo el control de asignación de transacciones y de mantener el estado de la base de datos correcto, resolviendo los posibles conflictos que pudieran aparecer para mantener la base de datos en un estado consistente.

8.1.- Modelo de arquitectura multigeneracional (MGA)

Para implementar el mecanismo de transacciones, Firebird utiliza una arquitectura multi-generacional (MGA). En este modelo, cada fila almacenada en la base de datos mantiene el ID de transacción único de la transacción que lo escribió. Si otra transacción guarda cambios a la fila, el servidor escribe en disco una nueva versión de la fila, con el nuevo ID de transacción, convirtiendo una imagen de la versión vieja en una referencia (delta) de esta nueva versión. Así el servidor mantiene dos versiones (generaciones) de la misma fila.

Las filas que se van creando en una transacción no serán visibles a las transacciones que se inicien posteriormente mientras no se finalice mediante un commit. Si al hacer esta operación se produce algún conflicto, el servidor devuelve una excepción al cliente y la confirmación falla. Normalmente la solución es hacer un rollback por parte del cliente, por lo que se deshace toda la transacción.

Un rollback nunca falla, por lo que cualquier cambio que se halla hecho durante una transacción se podrá deshacer. En determinadas situaciones una operación de rollback no hace que se borren físicamente las versiones de las filas del disco. Esto puede suceder cuando la transacción realiza muchas actualizaciones de datos o cuando se rompe el servidor durante una transacción.

Las versiones de registros de las transacciones deshechas son eliminadas de la base de datos cuando el sistema las encuentra en el curso de operaciones de procesado de datos. Por lo general, si una fila es accedida por una sentencia, cualquier versión antigua del registro que puede ser elegible para ser borrada, se marcará para que el proceso de recoleccion de basura (Garbage collection) la elimine. Puede suceder que haya filas a las que no se accede normalmente, en este caso la recolección de basura se realiza mediante operaciones como el backup.

En MGA, la existencia de una nueva versión de una fila pendiente de confirmar puede bloquear la fila. En muchas situaciones, la existencia de una versión más nueva confirmada bloquea una petición para actualizaro borrar la fila, en este caso se produce un conflicto de bloqueo. Esto nunca

sucede con las inserciones ya que no hay versiones delta ni bloqueos para la fila insertada. Sólo fallaran éstas cuando se produzca una violación de alguna restricción.

Cuando se recibe una petición de actualización o borrado el servidor inspecciona el estado de cualquier transacción que se propietaria de una versión más nueva de la fila. Si estas transacciones están activas o se han confirmado, el servidor responde acorde al contexto (parámetros de nivel de aislamiento y resolución de bloqueos) a la transacción solicitante.

Si la transacción con versión más nueva de la fila está activa, la transacción solicitante esperará (valor por defecto) hasta que se complete la otra transacción (commit o rollback). y luego el servidor permitirá que continue. Sin embargo, si se indica como parámetro NOWAIT, se elevará una excepción por conflicto a la transacción solicitante.

Si la transacción con versión más nueva de la fila es confirmada y la transacción solicitante está en nivel de aislamiento SNAPSHOT (concurrente), el servidor rechaza la petición e informa de un conflicto de bloqueo. Si la transacción está en nivel READ COMMITED, con la configuración por defecto RECORD_VERSION, el servidor permite la pertición y escribe una nueva versión del registro.

Firebird no usa el metodo convencional de bloqueo en dos fases. En nuestro caso todo el bloqueo es a nivel de fila y de tipo optimista, es decir, cada fila esta disponible para todas las transacciones de lectura- escritura hasta que alguna escriba una versión más moderna de ella.

Tras una operación de confirmación correcta, con las versiones viejas puede ocurrir:

- Si la operación es una actualización, la nueva imagén se convierte en la última versión confirmada y la imagen original del registro es marcada para recolección de basura.
- Si la operación fue un borrado, una marca reemplaza al registro obsoleto. Una operación de sweep o backup limpia esta marca y libera el espacio fisico en disco ocupado por la fila borrada.

En definitiva, bajo condiciones normales:

- Cualquier transacción puede leer cualquier fila que fue confirmada antes de que se iniciara.
- Cualquier transacción de lectura-escritura puede solicitarse para actualizar o borrar una fila.
- Se podrá confirmar una petición si no hay otra transacción de lectura-escritura que haya confirmado un cambio a una versión más moderna de la fila. A las transacciones que confirman lecturas normalmente se les permite indicar cambios que sobrescriben versiones confirmadas por nuevas transacciones.
- Si se permite un cambio en una fila, la transacción bloquea la fila. Otros lectores pueden leer la última versión confirmada de la fila, pero ninguna podrá indicar cambios mediante operaciones de update o delete para la fila.

Se puede aplicar también bloqueo a nivel de tabla. Hay dos mecanismos para ello: indicando el nivel de aislamiento para la transacción como SNAPSHOT TABLE STABILITY o al reservar la tabla (select * from tabla for update with lock). Este bloqueo no se aconseja salvo en ocasiones puntuales

8.2.- Las transacciones y su indentificador.

Para cada transacción se define un identificador. Un identificador de transacción TID es un entero de 32 bits único que se asigna a cada transacción solicitada por un cliente y aceptada. Este número se restaura a 1 cuando la base de datos se crea o se restaura. Este identificador nos sirve para establecer la “edad” de una transacción: los números más bajos, transacciones más antiguas.

En el sistema se mantienen varios TID, almacenados en la cabecera de la base de datos, la transacción más vieja de interés (OIT), la transacción activa más antigua (OAT) y el número a asignar a la próxima transacción.

El TID actual puede accederse mediante CURRENT_TRANSACTION

En la relación entre cliente y servidor se usan los TID para establecer estados para las transacciones. Así podemos tener transacciones activas, en limbo, deshechas, rotas y finalizadas. La existencia de muchas transacciones en algunos de los estados anteriores puede hacer que el sistema pierda eficiencia, se relentice, etc. De estos estados destacamos:

- transacciones deshechas. A las transacciones deshechas (rollback) no se les aplica la recolección de basura. Permanecen hasta que una operación de sweep las marca para después ser limpiadas. Es por ello que se hace necesario un proceso periódico manual de sweep.
- Transacciones rotas. Son aquellas que aparecen como activas pero no tienen conexión asociada. Normalmente ocurre con las aplicaciones clientes que no cierran las transacciones cuando se desconecta el cliente o cuando la aplicación se rompe o se pierde la conexión de red. El servidor no tiene mecanismo para diferenciar entre una transacción activa y una rota. La única forma es usar el time-out de la conexión para que el proceso de recolección las pueda limpiar sin que afecte al sistema. En este caso se consideran como deshechas (rollback) y se actua en consecuencia. Si las roturas de conexiones se producen con frecuenta y de forma abundante, puede afectar al rendimiento del sistema.
- Transacciones en limbo. Ocurren cuando fallan operaciones de COMMIT en dos fases a través de múltiples bases de datos. El sistema las reconoce como un caso especial por lo que no puede decidir no decide de forma automática ni confirmarlas ni deshacerlas ya que podrían producir inconsistencias en las bases de datos. Estas transacciones sólo pueden ser resueltas mediante la herramienta gfix.

8.3.- Configurar transacciones.

Una pieza clave en la gestión de las bases de datos es la transacción. Una transacción es configurable. Esta configuración determina como actuar cuando se encuentra el servidor con conflictos de datos. En Firebird los parámetros configurables son:

- Nivel de aislamiento.
- Modo de resolución de bloqueos.
- Modo de acceso
- Reserva de tabla.

Además, con el nivel de aislamiento READ COMMITED, se tiene que tener en cuenta el estado actual de las versiones de registros.

8.3.1.- Nivel de aislamiento.

En SQL estandar se define el aislamiento en transacciones como un mecanismo de bloqueo en dos fases. Este estandar se define no tanto como en terminos ideales como en términos de permitir o denegar unos determinados elementos. Se consideran como elementos a tener en cuenta:

- Lectura sucia: Ocurre si la transacción puede leer cambios de otras sesiones no confirmadas.
- Lectura no repetible: Ocurre si las lecturas posteriores del mismo conjunto de filas durante el curso de una transacción puede ser diferente respecto de lo que leyó al principio de la transacción.
- Filas fantasmas: Ocurre si un subconjunto de filas leidas durante la transacción difiere del conjunto leido inicialmente. La filas fantasmas ocurren cuando el subconjunto incluye

nuevas filas insertadas y/o excluye filas borradas que fueron confirmadas desde la lectura inicial.

En SQL estandar se definen varios niveles de aislamiento que cumplen:

Nivel aislamiento	Lecturas sucias	Lecturas no repetibles	Filas fantasmas
READ UNCOMMITTED (lecturas sin confirmar)	Permitidas	Permitidas	Permitidas
READ COMMITTED (lecturas confirmadas)	No permitidas	Permitidas	Permitidas
REPEATEABLE READ (lecturas repetibles)	No permitidas	No permitidas	Permitidas
SERIALIZABLE (consecutivas)	No permitidas	No permitidas	No permitidas

En Firebird no se permite el nivel READ UNCOMMITTED, READ COMMITTED se considera el estándar y los otros nivles no son posibles (por el sistema implementado basado en versiones (MGA)).

Así en Firebird se implementan los siguientes niveles de aislamiento:

READ COMMITED

Es el nivel mas bajo de aislamiento. Es el único nivel en el que están disponibles las filas confirmadas (modificadas e insertadas) por otras transacciones. En este nivel se permiten las lecturas no repetibles y las filas fantasmas. Es el nivel más util para trabajar con grandes volúmenes y con entrada de datos en tiempo real, aunque no es apropiado para tareas que necesiten vistas reproducibles (que no involucren cambios durante la transacción).

El nivel READ COMMITED, puede ser configurado para responder de forma mas conservadora ante confirmaciones externas y otras transacciones pendientes:

- Con RECORD_VERSION (valor por defecto), permite que la transacción lea la última versión confirmada. Si la transaccióne está en modo READ WRITE, se le permite sobrescribir la última versión confirmada si su TID es más nuevo que el del registro confirmado.
- Con NO_RECORD_VERSION, Boquea la transacción al leer una fila si está pendiente de actualizar. Esto se completa según el modo de resolución de bloqueo:
    - Con WAIT, la transacción espera hasta que la otra transacción confirma on deshace los cambios. Sus cambios se permitirán si la otra transacción hace rollback o si su TID es más nuevo que el de la otra transacción. Finaliza en una excepción por conficto de bloqueo si la otra transacción tiene un TID mas nuevo.
    - Con NOWAIT, la transacción recive inmediatamente una notificación de conflicto por bloqueo.

SNAPSHOT (concurrencia)

Este es el nivel medio, conocido tambien como de lecturas repetibles o concurrente. Aisla la vista de la transacción de cambios a nivel de fila. Este nivel aisla la transacción de las nuevas filas confirmadas por otras transacciones por lo que no se puede ver filas fantasmas. Sin embargo no es idéntico a SERIALIZABLE ya que otras transacciones pueden actualizar o borrar filas que están en la transacción, aunque ésta los haya modificado primero.

Con esta nivel se garantiza que se tiene una vista de la base de datos que no se verá afectada por cualquier confirmación hecha por otra transacción. Es la apropiada cuando se quieren realizar tareas de histórico como informes o obtención de resúmenes.

SNAPSHOT TABLE STABILITY (consistencia).

Es el nivel mas restrictivo de aislamiento, llamado también como consistente. En este nivel las transacciones de lectura-escritura no pueden leer tablas bloqueadas por la transacción.

8.3.2.- Modo de acceso.

La transacción se puede configurar como de solo lectura (READ ONLY) o de lectura-escritura (READ WRITE). Con el primer método se indica que la transacción se va a usar sólo para leer datos. Esto permite que el servidor tenga que reservar menos recursos para la transacción. Por defecto el modo de acceso es de lectura-escritura.

8.3.3.- Modo de resolución de bloqueos.

El modo de resolución de bloqueo determina como se actua cuando una transacción entra en conflicto con otras cuando actualiza datos ya cambiados por las otras. Se tienen dos posibilidatdes WAIT o NOWAIT.

WAIT.

En este modo la transacción espera hasta que las filas bloqueads por una transacción pendiente son liberadas, antes de determinar si puede actualizarla el. Así, si otra transacción ha guardado un cambio con una versión más alta, la transacción que espera recibirá una notificación conflicto por bloqueo. Este funcionamento puede hacer que se ralentice el trabajo cuando se tiene volúmenes altos de cambios o cuando se tiene aplicaciones interactivas.

Este tipo de resolución se desaconseja si trabajamos con transacciones de larga duración ya que pueden bloquear las transacciones por un tiempo considerable a la espera de que finalicen las otras transacciones.

NO WAIT.

En este modo, el servidor notifica inmediatamente al cliente un conflicto si detecta que la transacción intenta modificar una fila para la que detecta una versión nueva sin confirmar. En un entorno empresarial en el que hay mucho volumen de trabajo es la opción mas apropiada. El cliente posteriormente resuelve el bloqueo mediante una operación deshacer, reintento posterior u otra técnica apropiada.

8.3.4.- Reserva de tabla

Firebird soporta bloqueo de tabla para forzar el bloqueo completo sobre una o varias tablas durante la transacción. La cláusula RESERVING <lista tablas> bloquea inmediatamente todas las filas confirmadas en las tablas listadas, permientedo que una transacción acceda de forma exclusiva a expensas de cualquier otra transacción concurrente.

A diferencia de la técnica normal de bloqueo, ésta es pesimista, se hace la reserva todas las filas al inicio de la transacción en lugar de en el punto en el que se requiere el bloqueo de la fila.

Se usa para:

- Para asegurar que las tablas se bloquean al inicio de la transacción en lugar que cuando se accerden por primera vez por una sentencia. En caso de bloqueo con otra transacción se aplica el proceso estandar. Si la transacción tiene definidoo WAIT, esperará y con NOWAIT se generará una excepción. El bloqueo de tabla reduce mucho la posibilidad de interbloqueos.
- Para evitar el bloqueo por dependecia, es decir, el bloqueo que se produce a partir de actualizaciones en triggers o por restricciones de integridad.
- Para garantizar el orden de actualización de las tablas. Al hacer la reserva, garantizamos que durante la transacción no habrá ninguna actualización.

Se suele usar la reserva de tabla con los niveles SNAPSHOT y READ COMMITED. Este mecanismo es el menos agresivo y más flexible cuando se desea bloquear una tabla, aunque se tiene que tener en cuenta que no debe ser el funcionamiento cotidiano, sólo para momentos especiales.

Para establecer como se actua cuando otras transacciones desean acceder a los registros blqoeuados se tiene la sintaxis:

[PROTECTED | SHARED] {READ | WRITE}

Si se indica PROTECTED, la trasacción tiene bloqueo de lectura en exclusividad sobre la tabla y permite que cualquier otra transacción en modo SNAP_SHOT o READ COMMITED pueda leer filas. La escritura se restringe según:

- PROTECTED WRITE, permite que la transacción escriba en la tabla y bloquea todas las otras escrituras.
- PROTECTED READ, no permite escribir en la tabla a ninguna transacción, incluyendo la propia transacción.

Si se indica SHARED, permite a las transacciones SNAPSHOT o READ COMMITED leer la tabla mientras que para las actualizaciones

- SHARED WRITE permite a las transacciones SNAPSHOT de lectura-escritura o READ COMMITED de lectura-escritura, actualizar filas en el conjunto si no han pedido escritura en esclusividad.
- SHARED READ es la condición más liberal. Permite a cualquier otra transacción de lectura-escritura actualizar la tabla.

8.4.- Sentencias de gestión de transacciones.

En Firebird, no todas las características para gestión de transacciones están accesibles mediante sentencias SQL. En este caso las acciones se tienen que realizar mediante funciones definidas en la API de Firebird. De esta forma nos encontramos con dos posibilidades para gestionar las transacciones:

- ESQL. Incluye sentencias SQL para configurar determinados aspectos de las transacciones, como SET TRANSACTION para iniciar una transacción. Se pueden usar en determinados casos dentro de DSQL y en la utilidad ISQL.
- API. Aporta una interfaz de funciones completas para programadores de C y C++ para crear aplicaciones cliente. Un grupo de ellas realizan funciones semejantes a sentencias SQL para transacciones. Por ejemplo isc_start_transaction es equivalente a SET TRANSACTION.

INICIAR TRANSACCIONES (SET TRANSACTION)

Permite iniciar una transacción. Su sintaxis es:

SET TRANSACTION [NAME <nombre transacción>]
[READ WRITE | READ ONLY] -- modo de acceso
[WAIT | NO WAIT] -- modo de resolución de bloqueo
[ISOLATION LEVEL] -- nivel de aislamiento
{SNAPSHOT [TABLE STABILITY] | READ COMMITED [[NO] RECORD VERSION]}
[RESERVING <clausula reserva> | USING <manejador>]; -- reserva de tabla

<clausula reserva> = <tabla> [, <tabla>…]
[FOR [SHARED | PROTECTED] {READ | WRITE }]
[, <clausula reserva> ....]

En cuanto a la API, existe una sentencia para la definición de las transacciones en la que se usa una serie de constantes cuya equivalencia se puede observar en la siguiente tabla:

Tipo atributo	Atributo SQL	Constante API
Modo de acceso	READ ONLY	isc_tpb_read
	READ WRITE	isc_tpb_write
Nivel de aislamiento	READ COMMITED	isc_tpb_read_commited
	SNAPSHOT	isc_tpb_concurrency
	SNAPSHOT TABLE STABILITY	isc_tpb_consistency
Modo de resolucion de bloqueo	WAIT	isc_tpb_wait
	NO WAIT	isc_tpb_nowait
Versión de registros	RECORD_VERSION	isc_rec_version
	NO RECORD_VERSION	isc_no_rec_version
Reserva de tabla	SHARED	isc_tpb_shared
	PROTECTED	isc_tpb_protected
	READ	isc_tpb_lock_read
	WRITE	isc_tpb_lock_write

PUNTOS DE RETORNO (SAVEPOINTS)

Son usados para crear grupos de sentencias dentro de una transacción y confirmarlas como un todo. Se se produce una excepción, la transacción puede ser deshecha hasta el último punto de retorno. Las sentencias que se lanzaron desde el último punto de retorno y la excepción son deshechas y la aplicación puede continuar para confirmar, deshacer, etc toda la transacción.

Para crear un punto de retorno se usa
SAVEPOINT <identificador>

Posteriormente se puede volver al punto de retorno mediante
ROLLBACK [WORK] TO [SAVEPOINT] <identificador>

Ya que el sistema de los puntos de retorno es un mecanismo que puede tener un impacto grande en recursos (se gestiona en memoria), especialmente si la misma fila se actualiza repetidas veces, Firebird aporta una sentencia para liberar los puntos de retorno.

RELEASE SAVEPOINT <identificador> [ONLY];

Si no se indica ONLY se libera el punto de retorno <identificador> y todos los que se definieron posteriormente.

FINALIZAR TRANSACCIONES (COMMIT y ROLLBACK)

Una transacción se puede finalizar confirmando los cambios (COMMIT) o deshaciéndolos (ROLLBACK).

COMMIT [WORK] [RETAIN [SNAPSHOT]]

Con una sentencia COMMIT se confirma todos los cambios realizados durante la transacción y se libera todos los recursos asociados a la misma. Si se produjera una excepción, el cliente debe realizar un rollback de la misma.

El commit se puede hacer con la opción RETAIN (solo en transacciones SNAPSHOT). De esta forma el servidor mantiene una copia del estado en el que se confirmó la transacción y se inicia una

nueva transacción como un clone de la confirmada. En este caso no se liberan los recursos de la transacción.

Este mecanismo se añadió para dar solución a los programadores que modificaban datos en grids. En este caso, tras cualquier cambio, se podia hacer un autocommit o (commit retain). Esto reduce el esfuerzo del programador y reduce los recursos necesarios, aunque tiene varios problemas:

- La transacción mantiene su visión inicial, es decir, no ve los resultados de otras transacciones que confirman y estaban pendiente cuando se inició la transacción primera.
- Puede hacer que se incremente el uso de memoria y por lo tanto ralentice el servidor.
- No se marcan como obsoletas las versiones de las filas, por lo que no las elimina el recolector de basura.

ROLLBACK

Libera todos los recursos en el servidor y finaliza físicamente la transacción. La base de datos vuelve al estado en el que estaba antes de iniciarse la transacción.