TEMA 3: FIREBIRD, CARACTERÍSTICAS BÁSICAS

.1.- INTRODUCCIÓN 1
2.- SQL. 4
3.- ESTRUCTURA DE UNA BASE DE DATOS 5
3.1.- METADATOS Y ELEMENTOS FÍSICOS EN UNA BASE DE DATOS 5
4.- SCRIPTS 7
5.- TIPOS DE DATOS. 8
5.1.- Variables de contexto 8
5.2.- Cadenas predefinidas. 9
5.3.- Conversión de tipos 10
5.4.- NULL 10
5.5.- Tipos de datos numéricos. 11
5.6.- Tipos fecha y hora. 12
5.7.- Tipos cadena de caracteres. 14
5.8.- Tipo booleano. 16
5.9.- Tipos especiales 16
6.- EXPRESIONES Y PREDICADOS 18
6.1- Operadores 19
6.1.1.- Operadores concatenación. 19
6.1.2.- Operadores aritméticos. 19
6.1.3.- Operadores de comparación. 20
6.1.4.- Operadores lógicos. 22
6.1.5.- Predicados existenciales. 22
6.2.- Funciones. 25
6.2.1.- Funciones de conversión. 25
6.2.2.- Funciones CASE 25
6.2.3.- Funciones de agregación. 27
6.2.4.- Funciones de ventana (funciones analíticas). 28
6.2.5.- Funciones generales. 29
6.2.6.- Funciones numéricas. 30
6.2.7.- Funciones sobre cadenas. 31
6.2.8.- Funciones de fecha y hora. 32
6.2.10.- Funciones estadísticas 33
6.3.- Expresiones regulares. 34
6.3.1.- Casos de uso. 35

1.- INTRODUCCIÓN

Firebird deriva del código fuente de Interbase 6.0 de Borland. Es open source y no hay licencias duales. Tanto para uso comercial como para aplicaciones open source, es totalmente libre. La tecnología de Firebird lleva 20 años funcionando, esto hace que sea un producto muy maduro y estable.

El desarrollo de Firebird lleva aparejado la aparición de versiones que incluyen nuevas características y posibilidades. Así se comenzó con la versión 1.0 (simplemente portar el código de internase 6.0 en c), la versión 1.5 (conversión de firebird a c++), la versión 2.0 (nuevas características como tablas derivadas, etc), la versión 2.1 (características de gestión de sesiones, etc), la versión 2.5 (alter view, etc), la versión 3.0 (rediseño de motor, booleanos, etc). Estos temas están escritos con las características hasta la versión actual disponible (3) por lo que puede que algunas características no estén disponibles en versiones anteriores.

Firebird tiene todas las características y la potencia de un RDBMS. Se pueden manejar bases de datos desde unos pocos KB hasta varios Gigabytes con buen rendimiento y casi sin mantenimiento.

Sus características principales son:

- Soporte completo de Procedimientos Almacenados y Triggers
- Las Transacciones son totalmente ACID compliant
- Integridad referencial
- Arquitectura Multi Generacional
- Muy bajo consumo de recursos
- Completo lenguaje para Procedimientos Almacenados y Triggers (PSQL)
- Soporte para funciones externas (UDFs)
- Poca o ninguna necesidad de DBAs especializados
- Prácticamente no necesita configuración - ¡sólo instalar y empezar a usarla!
- Una gran comunidad y muchas páginas donde conseguir buen soporte gratuito
- Opción a usar la versión embebida - de un solo fichero - ideal para crear CDROM con catálogos, versiones de evaluación o monousuario de aplicaciones
- Docenas de herramientas de terceros, incluyendo herramientas visuales de administración, replicación, etc.
- Escritura segura - recuperación rápida sin necesidad de logs de transacciones
- Muchas formas de acceder a tus bases de datos: nativo/API, driver dbExpress, ODBC, OLEDB, .Net provider, driver JDBC nativo de tipo 4, módulo para Python, PHP, Perl, etc.
- Soporte nativo para los principales sistemas operativos, incluyendo Windows, Linux, Solaris, MacOS.
- Backups incrementales
- Disponible para arquitecturas de 64bits
- Completa implementación de cursores en PSQL

El servidor Firebird viene en tres versiones: SuperServer**,** Classic/SuperClassic y Embedded. Actualmente, Classic es la versión recomendada para máquinas con SMP y algunas otras situaciones

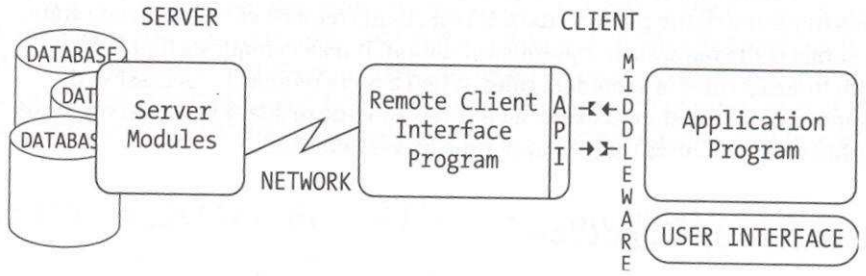
específicas. SuperServer comparte su caché para todas las conexiones y usa un hilo de ejecución para cada conexión. Ésta se suele usar en windows. Classic inicia un proceso de servidor independiente para cada conexión que se haga.

La versión embedded es una interesante variación del servidor. Es un servidor Firebird con todas sus características, empaquetado en unos pocos ficheros. El servidor no necesita instalación. Ideal para CDROM de catálogos, demos o aplicaciones de escritorio monousuario.

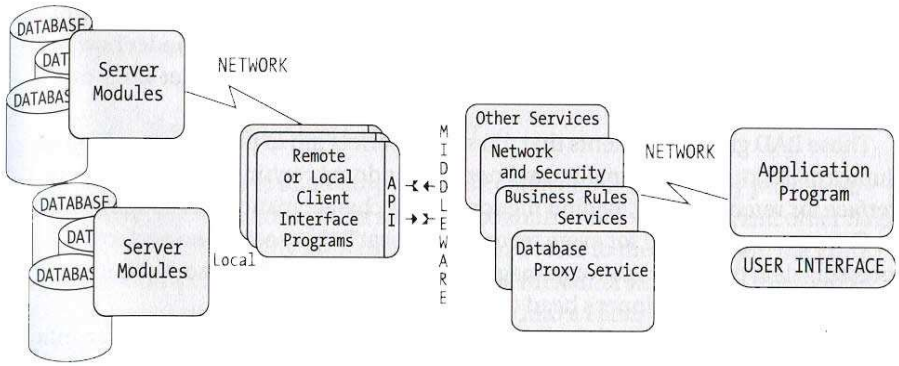
Firebird viene con un completo paquete de utilidades de línea de comando que te permiten crear bases de datos, generar estadísticas, ejecutar comandos y scripts SQL, hacer y recuperar copias de seguridad, etc. Si prefieres usar herramientas visuales, hay montones de opciones donde elegir, incluyendo gratuitas.

En Windows, se puede ejecutar Firebird como servicio o como aplicación. El instalador puede crear un icono en el panel de control que permite controlar el servidor (iniciarlo, pararlo, etc).

Firebird es un SGBD en plataforma cliente/servidor. El servidor acepta peticiones TCP/IP de los clientes, por defecto sobre el puerto 3050 (gds_db). Además puede comunicarse usando IPX. Para que los equipos clientes puedan conectarse al servidor es necesario instalar unas herramientas cliente, generalmente una librería, que en windows consiste en el fichero gds32.dll/fbclient.dll. Cuando instalamos firebird en un sistema podemos llegar a una configuración en dos niveles o en n-niveles.



Modelo 2- niveles



Modelo n-niveles

Además del protocolo TCP/IP se aceptan otros modos de conexión como local (XNET) sobre redes windows o usando NetBEUI (redes Windows), aunque se aconseja TCP/IP para acceder a todas las características que aporta FIREBIRD.

Firebird tiene establecido un sistema de seguridad basado en usuarios y contraseñas. En versiones previas a la 3.0, durante la instalación se creaba de forma automática el usuario SYSDBA con contraseña ‘masterkey’. En la versión 3.0 es posible indicar el nombre del usuario y contraseña que actuará como administrador del sistema.Esta información de usuarios se almacena en un fichero de base de datos Firebird: security2.fdb (para versiones de Firebird 2 o superiores), security3.fdb (para versiones de Firebird 3 o superiores).

Cuando un cliente quiere contactar con el servidor debe usar un protocolo de comunicación. Firebird aporta este protocolo implementado en forma de una DLL en el caso de WIN que se sitúa en una carpeta accesible del disco duro (normalmente c:\windows\system32) llamada **GDS32.DLL**, usada para compatibilidad con versiones anteriores o **fbclient.dll.**

El soporte principal de Firebird se puede encontrar en la página ‘www.firebirdsql.org’. Además se dispone de una página de soporte para la comunidad hispana en “www.firebird.com.mx”

Estos temas están creados a partir de la documentación de “las notas de versión de Firebird” que acompañan a las versiones de Firebird y del libro “The Firebird Book” de Helen Borrie, libro con el que me inicié en Firebird.

2.- SQL.

En Firebird, todas las operaciones se realizan a partir de SQL, un sublenguaje para acceso a sistemas de manejo de bases de datos relacionales.

Desde su introducción, el estándar SQL ha sufrido tres grandes revisiones: SQL-89, SQL-92, SQL-99, SQL-2003 y SQL-2008.

El lenguaje de consulta SQL es no procedimental, es decir, es orientado a los resultados de la operación en lugar de a indicar como obtener los mismos.

Los diferentes fabricantes se encargan de implementar en mayor o menor medida una versión de SQL, introduciendo, en la mayoría de los casos, características propias. En relación a Firebird, éste implementa un gran número de características de todos los estándar incluidos los últimos.

SQL describe una sintaxis. En teoríca, para cualquier operación que se quiera realizar sobre una Base de Datos, debe existir una sentencia que lo realice.

Las sentencias sintácticas se suelen agrupar según su propósito en:

- *Sentencias de definición de datos (DDL)*: Sentencias para la creación, modificación y borrado de los objetos en la base de datos. (CREATE, ALTER y DROP)
- *Sentencias de manipulación de datos (DML).* Se incluyen tanto sentencias para obtener datos de la Base de Datos (SELECT) como para introducir, modificar o borrar los propios datos almacenados (INSERT, UPDATE y DELETE).

En Firebird se tiene una implementación del lenguaje SQL que se puede dividir en varios subgrupos:

- *SQL Embebido (ESQL)*: Es la implementación SQL base, consistente en sintaxis DDL y DML que incorporan otros subconjuntos. Es la implementación original de SQL.
- *SQL Dinámica (DSQL):* Es el subconjunto más usado actualmente. Es usado como capa de interfaz para comunicar con el servidor a través de la interfaz de programación de aplicación (API). Algunas sentencias DDL implementadas en ESQL no estan disponibles aquí por lo que se reemplazan con funciones API.
- *SQL Interactiva (ISQL):* Es el lenguaje implementado para la utilidad en línea de comandos isql. Está basada en DSQL con extensiones para ver metadatos y algunas estadísticas del sistema.
- *SQL Procedimental (PSQL):* Es el lenguaje para escribir los procedimientos almacenados y los triggers. Se compone de todas las sentencias DML junto con otras extensiones especficas para procedimientos.

Las sentencias SQL embebidas y las precompiladas en código son conocidas como SQL estático (ESQL). En contraste, las sentencias generadas por el cliente y enviadas al servidor para su ejecución son conocidas como SQL dinámico (DSQL). Por lo general siempre se trabaja con DSQL. Aquí podemos encontrarnos con las sentencias en la utilidad isql o las que se ejecutan a través de la API directa o indirectamente (a traves de controladores ODBC, JDBC, etc).

Trabajar con DSQL es siempre más lento que trabajar con ESQL, ya que para cada sentencia es necesario analizarla y prepararla para ejecución cada vez que se lance. En ESQL, se realiza la preparación una sóla vez.

3.- ESTRUCTURA DE UNA BASE DE DATOS

Firebird es un sistema gestor de bases de datos relacional. Como tal, esta diseñado para soportar la creación y mantenimiento de estructuras de datos abstractas, no sólo almacenar datos sino también mantener las relaciones y optimizar la velocidad y consistencia cuando los datos pedidos son enviados a los clientes.

En su conjunto, los objetos definidos en una base de datos son conocidos como **metadatos** o **esquema**. El proceso de creación y modificación de los metadatos es conocido como definición de datos.

Todos los objetos del esquema son creados usando un subconjunto del lenguaje SQL conocido como Lenguaje de Definición de Datos (DDL). Una sentencia **DDL** comienza con alguna de las palabras CREATE, ALTER, RECREATE o DROP, permitiendo crear, modificar, reconstruir o destruir respectivamente un objeto del esquema.

Firebird almacena los metadatos en un conjunto de tablas que se crean dentro de la Base de datos, las **tablas de sistema**. Todas las tablas del sistema tienen identificadores que comienzan con "RDB$". Por ejemplo RDB$DATABASE contiene información de definición de la propia base de datos.

3.1.- METADATOS Y ELEMENTOS FÍSICOS EN UNA BASE DE DATOS

Físicamente se almacenan los datos y las relaciones existentes entre ellos mediante una serie de elementos. Además se establecen mecanismos para garantizar las restricciones establecidas. Todos estos elementos son:

- *Tablas:* Una tabla de una base de datos es un bloque bidimensional compuesto de columnas y filas en donde se almacena la información del mundo real.
- *Ficheros y páginas*: Todos los datos de una base de datos se almacenan en un fichero o como mucho, en un conjunto de ficheros enlazados. Un fichero se divide físicamente en páginas. Se tienen diferentes tipos de páginas, acorde al tipo de dato que almacena (columnas de tablas, BLOBS, indices, etc). Todas las páginas tienen el mismo tamaño y son reservadas (localizadas) en el momento en el que son necesarias. El tamaño de página se establece en el momento de crear la Base de Datos y no puede ser modificada. Debido a esta forma de almacenamiento, es posible que los registros de una tabla no estén almacenados físicamente de forma continua (pueden estar en diferentes páginas en diferentes partes del fichero o incluso en diferentes ficheros).
- *Columnas y Campos*: Abstractamente una columna es un conjunto de valores para los que se les ha definido un tipo y que pueden ser almacenados en una celda de una fila de una tabla. En la definición de la tabla se establece un orden de izquierda a derecha en la que se evaluarán los campos y que puede ser modificado al devolverlo en una consulta. Un campo es el valor de una columna en una fila determinada. En muchas ocasiones se intercambian el significado de columna y campo.
- *Claves*: Las claves son valores que identifican a las filas de las tablas. Nos encrontramos
  - *Clave primaria*: Representa una o varias columnas que, para cada tabla, identifican de forma unívoca (no puede haber dos filas con idénticos valores en los campos de clave) cada fila. En un SGBD se conoce como restriccion PRIMARY KEY.
  - *Clave única*: Además de definir una clave primaria, se pueden definir otras claves que identifican de forma unívoca cada fila. En firebird se realiza mediante la restricción UNIQUE.

- *Clave foránea*: Son el mecanismo por el que se establecen las relaciones entre diferentes tablas. Se entienden como "la columna A de la tabla 1 tiene como valor uno de los que hay en la columna B de la tabla 2 a la que hace referencia"

Normalmente se usan como claves valores que no deban modificarse una vez establecidos y que puedan ser fácilmente gestionables. En general se usan columnas de tipo entero que no contienen información significativa para la fila. Por ejemplo en una identificación de un curso se podría usar como clave la columna con la descripción del mismo ("Curso 1 A ESO"), aunque cualquier error al rellenarlo podría suponer un problema en todas las tablas relacionadas. En su lugar se usa una columna de tipo entero que identifique la fila, (el 1 representa a "Curso 1 A ESO"). Si cambiamos la descripción, no es necesario para nada modificar la clave, el 1.

Una clave no es un índice. Una clave es una restricción a nivel de tabla. Normalmente, el SGBD responde a esta restricción creando uno o más metadatos para facilitar su gestión, como podrían ser índices únicos para claves primarias o únicas, indices no únicos para claves foráneas, etc.

- *Integridad Referencial:* Es el mecanismo que se establece físicamente en la base de datos para mantener restricciones que se hayan establecido. Así en una restricción de clave única garantiza que dos filas no tienen el mismo valor en la columna clave.
- *Indices y planes de consulta:* Un índice es el mecanismo que nos aporta la base de datos para localizar de forma rápida valores dentro de una tabla. Un buén índice acelera las búsquedas. Un mal índice o simplemente no tener índice puede hacer más lentas las búsquedas, uniones y ordenaciones.
  Cuando tenemos una consulta en la que se combinan dos o más tablas se hace necesario establecer físicamente como se realiza la unión de las mismas. Firebird utiliza algoritmos de optimización basados en el coste. Para una consulta dada, el optimizador evalua el coste relativo de usar o ignorar los índices disponibles y devuelve un plan de consulta. Este plan de consulta es posible establecerlo manualmente, aunque siempre se aconseja usar el obtenido por el optimizador.
- *Vistas:* Una vista es una consulta predefinida almacenada en la base de datos. Una vista es una clase de tabla que no almacena físicamente datos. Se suele usar cuando se quiere restringir el acceso a una columna a usuarios o para acceder a consultas que se utilizan mucho.
- *Procedimientos almacenados y triggers:* Son módulos de código ejecutable compilados en el servidor y que se ejecutarán directamente en él. El código fuente se genera a partir de un conjunto de extensiones de lenguaje SQL conocido como SQL Procedimental o **PSQL.**
  Un procedimiento almacenado puede ser ejecutable o seleccionable. Pueden recibir argumentos de entrada y devolver conjuntos de salida. Los procedimientos ejecutables devuelven una sola fila de valores constantes. Los procedimientos seleccionables devuelven una o más filas de un conjunto resultado, al estilo de una tabla o vista.
  Un trigger es un módulo que se declara para ejecutarse en una o más de los seis estados/ fases (antes o después de una inserción, actualización o borrado) durante una operación de manipulación de datos en una tabla. No aceptan argumentos de entrada ni devuelven conjuntos resultados.
  En PSQL se establecen mecanismos para manejar excepciones. Así se podrá definir excepciones como código para gestionar las mismas en un módulo.

4.- SCRIPTS

Con Firebird, como con otros SGBD SQL, se pueden crear todos los metadatos de una base de datos así como rellenarla a partir de sentencias SQL. Esta operación se puede automatizar mediante un script. Un script no es más que un conjunto de sentencias de DDL y/o DML almacenadas en un fichero de texto plano. Se suele usar para crear, modificar o borrar objetos de una base de datos, así como para rellenar, modificar o borrar datos de ella.

En un script se puede usar cualquier sentencia de DDL y de DML, junto con:

- *COMMIT, ROLLBACK:* sentencia que permite guardar o deshacer los cambios que se realizan en las sentencias anteriores.
- *SET AUTODDL {ON | OFF}*: Hace que automáticamente se guarden o no los cambios realizados en sentencias DDL.
- *Comentarios:* En un script se puede indicar dos tipos de comentarios: en bloque (/* comentario */) o en línea (-- comentario).
- *Terminadores:* todas las sentencias en un script deben finalizar con un símbolo. Por defecto se utiliza el ';'. Esto es siempre así salvo en los procedimientos almacenados, en los que al estar definiendo bloque de código se debe establecer otro terminador. Esto se realiza mediante la sentencia SET TERM. Por ejemplo podríamos tener el siguiente script:

```
…..
CREATE GENERATOR generador; -- linea con su terminador ;
SET TERM ^^;    -- fija a partir de ahora como terminador ^^
CREATE TRIGGER mitriger FOR tabla
ACTIVE BEFORE INSERT POSITION 0
AS
   IF (NEW.PK IS NOT NULL) THEN
         NEW.PK=GEN_ID(generador,1);
END ^^ -- cierro la sentencia CREATE TRIGGER con el nuevo terminador
SET TERM ;^^  /* establezco de nuevo como terminador el ; y acabo la sentencia con el terminador actual */
…..
```

5.- TIPOS DE DATOS.

Cuando se crea una nueva columna en una tabla se debe definir el tipo de la misma, el cual establece el conjunto de valores válidos. Además el tipo de dato establece también las operaciones que se pueden realizar sobre el mismo. Los tipos de datos definen también el espacio necesario para su almacenamiento en disco. Elegir el tamaño de dato acertado es importante a la hora de almacenar gran cantidad de información.

Firebird soporta los siguientes tipos de datos:

a) Tipos de datos numéricos:
   - BIGINTEGER, INTEGER y SMALLINT
   - FLOAT y DOUBLE PRECISION
   - NUMERIC y DECIMAL
b) Tipos fecha y hora
   - DATE, TIME y TIMESTAMP
c) Tipos caracter
   - CHARACTER, VARYING CHARACTER y NATIONAL CHARACTER
d) Tipos especiales
   - BLOB y ARRAY
e) Tipos booleanos
   - Boolean

Relacionado con los tipos están los dialectos SQL (propios de firebird). Versiones antiguas de interbase usaban el dialecto 1. A partir de Interbase 5 (interbase 6 y firebird) se usa el dialecto 3. El dialecto establece restricciones sobre los tipos de datos. En nuestro caso siempre usaremos el dialecto 3.

El tipo BLOB (binary large object) permite almacenar datos que no pueden ser almacenados fácilmente en los tipos SQL estandars. Suele usarse para objetos de un tamaño variable e indeterminado, como imágenes, videos, capítulos de un libro, etc.

El tipo ARRAY no los veremos al no ser un tipo que podemos encontrar en otros SGBD.

5.1.- Variables de contexto

Firebird incluye una serie de variables mantenidas por el sistema que nos permiten acceder a datos interesantes durante una conexión de un cliente. Nos podemos encontrar:

VARIABLE	TIPO	DESCRIPCIÓN
CURRENT_CONNECTION	INTEGER	ID de la conexión actual

CURRENT_DATE	DATE	Fecha actual del servidor
CURRENT_ROLE	VARCHAR(31)	ROLE que usa el usuario en la conexión
CURRENT_TIME	TIME	Hora actual en el servidor.
CURRENT_TIMESTAMP	TIMESTAMP	Fecha/hora actual en el servidor
CURRENT_TRANSACTION	INTEGER	ID de la transacción actual
CURRENT_USER	VARCHAR(128)	Nombre del usuario actual
ROW_COUNT	INTEGER	Número de filas afectadas (cambiadas, borradas, añadidas) por la última sentencia de DML realizada. Solo en procedimientos almacenados
UPDATING	BOOLEAN	True si es una sentencia de actualización. Sólo en triggers.
INSERTING	BOOLEAN	True si es una sentencia de inserción. Sólo en triggers.
DELETING	BOOLEAN	True si es una sentencia de borrado. Sólo en triggers.
SQLSTATE	CHAR(5)	Devuelve el SQLSTATE dentro de un bloque WHEN. Solo en PSQL
SQLCODE	INTEGER	Devuelve el SQLCODE dentro de un bloque WHEN. Solo en PSQL
GDSCODE	INTEGER	Devuelve el GDSCODE dentro de un bloque WHEN. Solo en PSQL
USER	VARCHAR(128)	Nombre de usuario que es comunicado a través de la librería cliente.

Por ejemplo se podría usar una variable de contexto:

Select current_time as hora from RDB$DATABASE;

5.2.- Cadenas predefinidas.

En Firebird existen unas cadenas con un significado propio cuando se usan en determinadas sentencias.

CADENA	TIPO	SIGNIFICADO
'NOW'	TIMESTAMP	Dia/hora actual
'TODAY'	DATE	Dia actual
'YESTERDAY'	DATE	Dia actual –1
'TOMORROW'	DATE	Dia actual + 1

Por ejemplo podríamos tener:

Select cast('NOW' as date) as fecha from RDB$DATABASE

5.3.- Conversión de tipos

Normalmente se deben utilizar tipos compatibles para realizar operaciones aritméticas o para comparar datos en condiciones de búsqueda. Si se necesitan realizar operaciones sobre datos de tipos distintos se hace necesario conversión entre ellos. En firebird se realizan conversiones de tipo implícitas y explícitas. Las primeras las realiza de forma automática el sistema, generando un error en caso de no poder realizarla. Las segundas se realizan mediante la función **cast**.

Cast(valor|NULL AS tipo_dato)

Por ejemplo podríamos tener:

Cast('1' as smallint), cast('12/01/2008' as date), cast(null as integer)

5.4.- NULL

NULL es un punto en el que nos encontramos con resultados inesperados en operaciones realizadas con una base de datos. NULL indica sin valor o indeterminado, es decir, cualquier valor posible y por lo tanto no es igual a ninguno concreto. NULL es distinto de los valores cero: cadena vacía, 0 para los números, false para los lógicos, etc.

El valor NULL puede ser almacenado en cualquier columna que se haya definido como nullable (valor por defecto). Este valor se asigna automáticamente cuando se hace una sentencia de inserción y no se indica valor para una columna, y además, ésta no tiene definido un valor por defecto.

Como he dicho antes, NULL no es ningún valor concreto y por lo tanto no se pueden hacer sentencias del tipo *where (columna = NULL)*, en la que se generararía un error, y no se pueden hacer comparaciones (=, >, <, etc) con ningún valor En su lugar se utiliza el predicado IS NULL, por ejemplo, *where (columna IS NULL)*.

Hay que tener en cuenta que cuando se usan columnas para cálculos, y alguna de ellas tiene el valor NULL, el resultado es siempre NULL, Por ejemplo si tenemos

```
uptdate tabla
        set colA=colB +colC
```

Si colB o colC vale NULL, en colA se almacenará NULL

Esto no sucede cuando se usan funciones de agregación (SUM, AVG, COUNT, …). En éstas se eliminan las filas en las que aparecen NULL en la columna sobre la que se aplica la función.

En expresiones booleanas, se considera el valor NULL como false, es decir, es falso todo lo que no es verdadero (true). Hay una diferencia entre false y NULL. Si nosotros negamos una expresión con valor false se obtiene true (NOT false = true). Cuando negamos una expresión con valor NULL se obtiene NULL (NOT NULL = NULL). Por ejemplo podríamos tener

```
….
where
        NOT (colA=colB)
```

Si colA y colB tienen valores distinto de NULL y distintos, la igualdad se evalua a false y el NOT a true. Si colA o colB tienen valor NULL, la igualdad se evalua a NULL (false), y el NOT a NULL(false)

Por último, es posible asignar directamente el valor NULL a una columna en sentencias de inserción o de actualización.

5.5.- Tipos de datos numéricos.

Firebird soporta tipos numéricos decimales y de punto flotante. Entre los decimales nos encontramos con los que tienen parten fraccionaria 0 (SMALLINT, INTEGER y BIGINT) y los que tienen parte fraccionaria de tamaño fijo (NUMERIC y DECIMAL). Los tipos de punto flotante soportados son FLOAT y DOUBLE PRECISION. Firebird no soporta tipos de datos numéricos sin signo.

TIPO	TAMAÑO	RANGO/PRECISIÓN	DESCRIPCIÓN
SMALLINT	16 bits	–32,768 to 32,767	Entero corto con signo
INTEGER	32 bits	–2,147,483,648 a 2,147,483,647	Entero con signo
BIGINT	64 bits	-2^{63} a $2^{63}-1$	Entero largo con signo
NUMERIC (precision, escala)	Variable (16, 32 o 64 bits)	Precisión: 1 a 18, especifica el número exacto de dígitos de precisión. Escala: 0 a 18, especifica el número de lugares decimales.	Por ejemplo NUMERIC(10,4) sería un número con formato PPPPPP.EEEE (123456.7890)
DECIMAL (precision, escala)	Variable (16, 32 o 64 bits)	Precisión: 1 a 18, especifica el número mínimo de dígitos de precisión. Escala: 0 a 18, especifica el número de lugares decimales.	Por ejemplo DECIMAL(10,4) sería un número con formato PPPPPP.EEEE (123456.7890)
FLOAT	32 bits	1.175×10^{-38} a 3.402×10^{38}	IEEE simple precisión: 7 digitos
DOUBLE PRECISION	64 bits	2.225×10^{-308} a 1.797×10^{308}	IEEE doble precisión: 15 digitos

En firebird, con los tipos de datos numéricos, se pueden realizar las operaciones más comunes:

- Comparaciones: con los operadores relacionales (=, <, >, <=, >=, <> o ¡=)
- Aritméticas: (+, -, *, /)

Como se ha comentado anteriormente, se puede trabajar con datos decimales con la parte fraccionaria de tamaño fijo. Para ello se utilizan los tipos NUMERIC y DECIMAL. Estos tipos se utilizan cuando queremos trabajar con decimales con una precisión fija. Por ejemplo, si queremos manejar el precio de los articulos en un almacén, siempre usaremos un numero con 4 decimales. Si lo implementamos con un número real en punto flotante tendremos problemas a la hora del redondeo con operaciones como multiplicaciones o divisiones. Si usamos en su lugar, por ejemplo, un NUMERIC, se acabarán estos problemas.

En SQL estandar se definen estos dos tipos de datos: NUMERIC y DECIMAL. Conceptualmente son idénticos con la única diferencia que dada una precisión, en NUMERIC sólo se

admiten números hasta esa precisión, mientras que en DECIMAL se admiten números cuya precisión es de al menos la indica.

Para comprobar lo anterior queremos almacenar varios números para lo que definimos:

- NUMERIC(4,2): podemos almacenar el 7.2345 (se guarda como 7.23), el 23.45 pero no podría almacenarse el 753.22 (al necesitar una precisión de 5).
- DECIMAL(4,2): se podrían almacenar todos los números indicados para NUMERIC además del 753.22, ya que hemos indicado una precisión de al menos 4 cifras.

Internamente estos tipos de datos se almacenan como un entero del tamaño necesario para la precisión indicada.

En relación con las operaciones aritméticas con operandos de distinto tipo es interesante tener en cuenta que una división entre enteros da un resultado entero. 6/2 da 3, mientras que 1/3 da 0. Si queremos obtener un resultado de tipo real tendremos que hacer que uno de los operandos sea real. Por ejemplo 1/3 lo haríamos como (1*1.0)/3 o como (1+0.0)/3 dándonos el resultado de 0.333.

Firebird da soporte a la definición de literales hexadecimales y cadenas hexadecimales. En ambas se limita el número de caracteres hexadecimales a 16:

- literal hexadecimal: ***{ 0x | 0X } <dig hex> [<dig hex>...].*** Por ejemplo 0x12AF23.
- cadena hexadecimal: ***{ x | X }' <dig hex> [<dig hex>...]'.*** Por ejemplo x'12AF23'

Por ejemplo podríamos tener:

```
select cast(35 as SMALLINT),
     cast(33444 as integer),
     cast((11111111234.56789) as numeric(10,2)),
     cast((111111123441.56789) as decimal(10,2)),
     12/(5*1.0),
     0xA
   from RDB$DATABASE
```

5.6.- Tipos fecha y hora.

Nos encontramos aquí con tres tipos:

NOMBRE	TAMAÑO	RANGO/PRECISIÓN	DESCRIPCIÓN
DATE	32 bits con signo	1 enero 100 a.c. a 29 febrero 32768 dc	Fecha
TIME	32 bits sin signo	00:00 a 23:59:59.9999	Hora
TIMESTAMP	64 bits	–32,768 to 32,767	Fecha y hora almacenada como dos enteros.

Al estar todos los tipos fecha almacenados como números, es posible, realizar operaciones aritméticas. Así si a un tipo DATE le sumamos o restamos un entero, obtenemos una nueva fecha con el incremento o decremento en días correspondiente. Si restamos dos fechas, obtenemos el número de días entre ellas. Si restamos dos horas, obtenemos el número de segundos entre ellas.

```
select cast('12/01/2008' as date) +50, -- incremento la fecha en 50 dias
     cast('12:01:34' as time) +10*60*60 –incremento la hora en 10 horas
     cast('12/01/2008' as date) -
     cast('10/01/2008' as date)  -- diferencia entre fechas.
    from RDB$DATABASE
```

Cuando utilizamos fechas en expresiones con cadenas de caracteres, se suelen utilizar caracteres para identificar las diferentes partes:

CARACTERES	DESCRIPCIÓN
CC	Centenario, Primeros dos dígitos de un año.
YY	Año en el centenario. Dos últimas cifras de laño. Si no se indica la parte del centenario se obtiene una de forma automática.
MM	Mes. Entero que representa al mes en el rango de 1 a 12.
MMM	Mes. Iniciales en inglés del año. (JAN, FEB, MAR, etc)
DD	Dia del més. Entero en el rango de 1 a 31
HH	Hora. Entero en el rango 00 a 23.
NN	Minutos. Entero en el rango de 00 a 59
SS	Segundos. Entero en el rango de 00 a 59
nnnn	Diezmilésimas de segundo. En el rango de 0 a 9999.

Así nos podríamos encontrar fechas como 'CCYY-MM-DD', 'YY-MM-DD', 'MM/DD/CCYY', 'MM/DD/YY', horas como 'HH:NN:SS.nnnn' y timestamp como 'MM/DD/CCYY HH:NN:SS.nnnn'.

Cuando se trabaja con fechas se obtiene el formato que se está usando según el separador. Así, y esto hay que tenerlo en cuenta, en una fecha que usemos '/' se sigue el formato 'MM/DD/CCYY' (mes/dia/año). Por lo tanto la fecha '14/01/2008', no es válida en Firebird ya que representa el 1 del mes 14 de 2008, més que no existe.

Como ya se indicó anteriormente se pueden utilizar una serie de literales y variables de contexto relacionados con las fechas: 'NOW' (fecha-hora actual), 'TODAY' (fecha actual), 'TOMORROW' (mañana), 'YESTERDAY' (ayer), CURRENT_TIMESTAMP (fecha-hora actual), CURRENT_DATE (fecha actual) y CURRENT_TIME (hora actual).

Una operación muy usual con las fechas es su conversión a cadena de caracteres. En Firebird se hace mediante CAST, convirtiendo el tipo a una cadena de tamaño adecuado:

```
Select cast((10+cast('today' as date)) as char(25)),
    cast('yesterday' as date) , --ayer
    current_date,   --hoy
    cast('tomorrow' as date)  --mañana
  from RDB$DATABASE.
```

Por último, Firebird incorpora una función para obtener partes de una fecha/hora, la función EXTRACT.

EXTRACT(elemento FROM fecha-hora)

Esta función devuelve el elemento indicado del campo fecha-hora. Los posibles valores para elemento son:

ELEMENTO	TIPO DEVUELTO	RANGO	DESCRIPCIÓN
YEAR	SMALLINT	0-5400	Año de la fecha o timestamp
MONTH	SMALLINT	1-12	Mes de la fecha o timestamp

DAY	SMALLINT	1-31	Día de la fecha o timestamp
HOUR	SMALLINT	0-23	Hora de la hora o timestamp
MINUTE	SAMLLINT	0-59	Minutos de la hora o timestamp
SECOND	DECIMAL(6,4)	0-59.9999	Segundos de la hora o timestamp
WEEKDAY	SMALLINT	0-6	Día de la semana (0: domingo,…, 6: sabado)
YEARDAY	SMALLINT	1-366	Día del año de la fecha o timestamp
WEEK	SMALLINT	1-52	Semana del año de la fecha o timestamp

Por ejemplo podríamos tener:

```
select extract(YEAR from current_date) as anio from RDB$DATABASE.
select cast(extract(SECOND from current_time) as char(2)) as segundos from RDB$DATABASE
```

5.7.- Tipos cadena de caracteres.

Firebird permite trabajar con cadenas de longitud fija y de longitud variable. En las cadenas de caracteres de tamaño fijo se almacenan exactamente todos los caracteres. En una cadena de longitud variable se almacenan los caracteres que realmente se usan mas un entero con la longitud de la cadena usada. Así, si tengo la cadena 'hola', y la quiero como una cadena de longitud 10, si uso cadenas de longitud fija se almacenaría físicamente como 'hola ' mientras que con cadenas de longitud variable sería 'hola' junto con longitud 4.

```
select '-' ||cast('hola' as char(10))||'-', -- cadena de longitud fija
    '-' ||cast('hola' as varchar(10))||'-' – cadena de longitud variable
  from RDB$DATABASE
```

Firebird define los siguientes tipos para cadenas de caracteres:

NOMBRE	TAMAÑO	DESCRIPCIÓN
CHAR(n), CHARACTER(n)	N caracteres	Cadena de n caracteres de longitud fija.
NCHAR(N), NATIONAL CHAR(n)	N caracteres	Cadena de n caracteres de longitud fija en la que se usa como conjunto de caracteres el ISO8859_1
VARCHAR(n), CHARACTER VARYING (n)	N caracteres	Cadena de n caracteres de longitud variable.
NCHAR VARYING(N), NATIONAL CHAR VARYING(n)	N caracteres	Cadena de n caracteres de longitud variable en la que se usa como conjunto de caracteres el ISO8859_1

Las cadenas no pueden ocupar más de 32767 bytes. Esta limitación hay que tenerla muy en cuenta ya que, según el conjunto de caracteres, se utilizan uno, dos o tres bytes para almacenar un carácter. Así podremos almacer como máximo 32767, 16383 o 8191 caracteres dependiendo del conjunto de caracteres utilizado.

En Firebird se utiliza la comilla simple para delimitar una cadena (') ('esta es la cadena'). Esto lleva al problema de cómo introducir una comilla dentro de una cadena. Para ello se utiliza un carácter escape, la comilla simple. Por ejemplo para poner la frase "IES 'Pedro espinosa'" indicaríamos la cadena 'IES ''Pedro Espinosa'''.

Una operación muy usual con cadenas de caracteres es la concatenación. En Firebird se realiza mediante '||'. Por ejemplo.

Select nombre || ' ' apellido1 || ' ' apellido2 as nombre_completo from RDB$DATABASE

Como he referido antes, es necesario tener en cuenta el conjunto de caracteres que se utiliza. En un conjunto de caracteres (character set) se definen los caracteres que reconoce y el número de bytes utilizados para almacenar cada carácter. En un conjunto de caracteres se establece el valor numérico que representa cada carácter. Es por ello, que el mismo carácter puede tener diferentes valores en diferentes conjuntos de caracteres. Además se establece la secuencia de ordenación (collate) que indica el orden alfabético de ordenación de los caracteres en el conjunto de caracteres. Puede ocurrir que varios grupos usen el mismo conjunto de caracteres (españoles y suecos) pero que cada grupo establezca diferentes criterios de ordenación de esos caracteres (collate).

Cuando trabajamos con una base de datos, se establece un conjunto de caracteres por defecto. Este conjunto por defecto es el que se aplicará a todas las definiciones de columnas salvo se indique una específica. Si no se indica ninguno se utiliza el conjunto de caracteres NONE.

Los principale conjuntos de caracteres definidos son:

CONJUNTO	TAMAÑO	DESCRIPCIÓN
NONE	1 byte	Cada byte es parte de una cadena, pero no se realiza ninguna relación entre el byte y conjunto de caracteres alguno.
OCTETS	1 byte	Los bytes no se pueden interpretar como caracteres. Se utiliza para almacenar datos binarios.
ASCII	1 byte	Los valores de 0..127 definidos por la tabla ASCII. Los números fuera del rango no son caracteres aunque están reservados
UNICODE_FSS	2 bytes	Implementación del UTF8. En ella también se puede almacenar UCS16.
ISO8859_1 (LATIN_1)	1 byte	Implementación de la pagina de código 8859-1 definida por ISO. Incorpora los caracteres utilizados en Europa. Semejante a la tabla ASCII, pero con los caracteres extendidos rellenos.
WIN1250, WIN1251, WIN1252, WIN1253, WIN1254	2 Bytes	Implementación de páginas de códigos de Windows.

En Firebird se puede establecer tanto la página de códigos como el orden. Por ejemplo podríamos tener

Alter table prueba add Cadena varchar(40) character set ISO8859_1 collate ES_ES

… group by cadena collate CA_CA – establece la agrupación en una sentencia según un criterio

5.8.- Tipo booleano.

En firebird se tiene el tipo boolean que cumple con el estándar SQL-2008 y que se almacena internamente como un tipo entero de 8 bits. En el tipo boolean encontramos tres valores: TRUE, FALSE y UNKNOWN, que se corresponden respectivamente con los valores verdadero, falso y desconocido (null). Este tipo se puede usar tanto para campos como en expresiones y predicados.

Para trabajar con datos booleanos tenemos el operador IS así como los operadores de equivalencia (=, <>, …).

El operador IS [NOT] permite comprobar si un dato es TRUE, FALSE o UNKNOWN. Por ejemplo podríamos tener: *var is true, var is false, var = true, var<>false*.

Antes de firebird 3.0 no existía el tipo booleano como tal. En ese caso se implementaba como un único carácter (con valores S o N, V o F) o mediante un SMALLINT.

5.9.- Tipos especiales

Aquí nos encontramos tantos los BLOBS como los ARRAYS. Los arrays quedarán pendientes.

Un BLOB (Binary Large Object) es una estructura compleja para almacenar datos de objetos de tamaño variable, desde unos pocos bytes a gigabytes. Firebird los almacena como punteros. En el registro aparece la referencia a otro lugar donde se almacena realmente el contenido.

En un BLOB se puede almacenar cualquier tipo de información: una imagen de mapa de bits, una imagen vectorial, un video, un capítulo de un libro. Esto hace que el cliente sea el encargado de procesar la información almacenada y que un campo BLOB no pueda ser indexado.

Aunque en un blob se puede almacenar cualquier tipo de información, Firebird incorpora ya dos tipos predefinidos (mediante el atributo subtype):

- BLOB SUB_TYPE 0: Puede almacenar cualquier tipo de dato.
- BLOB SUB_TYPE 1o BLOB SUB_TYPE TEXT: Para almacenar texto plano. Aportando algunas peculiaridades.

En firebird, es posible definir un subtipo propio. Para ello es necesario definir un filtro que nos permita transformar la información desde el servidor al cliente.

Para definir un campo blob haríamos

Create table (campo_memo BLOB SUB_TYPE 1);

En Firebird se tiende a hacer equivalentes las cadenas de caracteres y los BLOB TEXT. Así en todas las sentencias donde se puede usar una cadena de caracteres, se puede usar un campo BLOB.

6.- EXPRESIONES Y PREDICADOS

Las expresiones SQL proporcionan métodos para calcular, transformar y comparar valores. Es el mecanismo por el que transformamos la datos almacenados en la Base de Datos en información útil para el usuario.

Las expresiones se usan para:
- Transformar datos devueltos como columnas (cláusula SELECT de una sentencia SELECT)
- Indicar condiciones de búsqueda en la cláusula WHERE de una sentencia de DML.
- Establecer condiciones de validación en una restricción CHECK.
- Definir columnas calculadas (COMPUTED BY en CREATE TABLE o ALTER TABLE)
- Cuando se generan conjuntos de entrada en sentencias INSERT o UPDATE.
- Cuando se establecen criterios de ordenación o agrupación en conjuntos resultados.
- Como condiciones de control de flujo en módulos PSQL.
- En tiempo de ejecución para determinar condiciones en conjuntos.

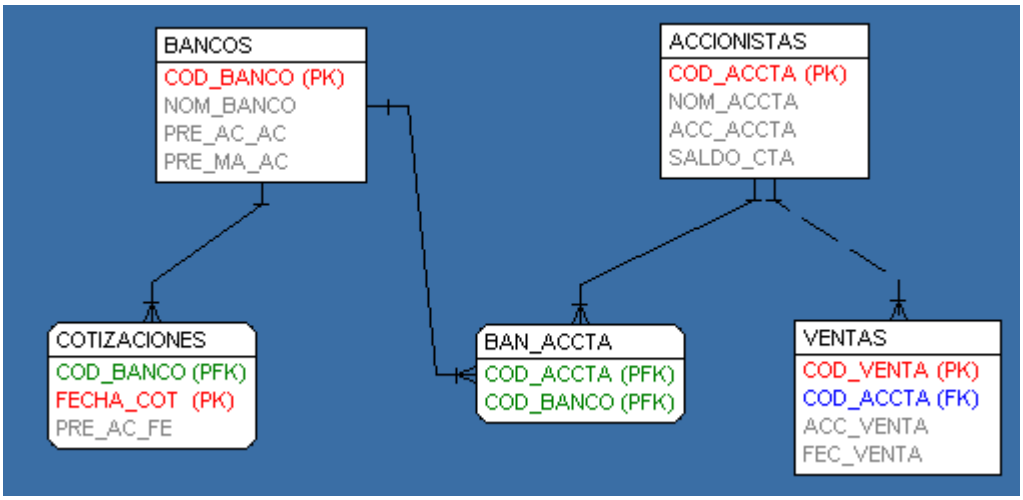
Un predicado es una expresión que establece la veracidad de una condición sobre un valor, es decir, indica si es verdad o no una determinada condición establecidad sobre un dato.

En SQL se suelen usar los predicados en:
- Cláusulas WHERE.
- Sentencias CASE (CASE, COALESCE y NULLIF).
- Condiciones en un JOIN
- Condiciones en un HAVING.
- En PSQL, condiciones para IF, WHILE y WHEN.

Un predicado puede ser Verdadero, Falso o no probado (NULL). En un predicado se consideran estos dos últimos valores como semejantes, es decir, “si no es verdadero, es falso”.

En una expresión nos podremos encontrar:
- Nombres de columnas de objetos de la base de datos.
- Nombres de columnas en conjuntos de salida.
- La palabra clave AS (usada en los alias de columnas o para campos calculados)
- Operadores aritméticos (+, -, /, *)
- Operadores lógicos (NOT, AND y OR)
- Operadores de comparación (<, >, =, <=, >= y <>).
- Otros operadores de comparación (LIKE, STARTING WITH, CONTAINING, BETWEEN, SIMILAR , IS [NOT] NULL, IS [NOT] (TRUE|FALSE|UNKNOWN), IS [NOT] DISTINCT FROM
- Operadores existenciales (IN, EXISTS, SINGULAR, ALL, ANY y SOME).
- Operador concatenación (|| para cadenas).
- Constantes (23, ‘hola’, etc)
- Literales de fecha (‘NOW’, ‘YESTERDAY’, ‘TOMORROW’)
- Variables internas de contexto.
- Subconsultas.
- Variables locales (en PSQL variables definidas)
- Funciones (incluye todo tipo de funciones como CASE, CAST, SUBSTRING, etc).
- (). Para agrupar expresiones.

En este tema desarrollaremos muchos de los ejemplos a partir del siguiente esquema de una base de datos muy elemental.



6.1- Operadores

SQL incorpora una serie de operadores para trabajar con columnas y constantes. Se pueden agrupar en cuatro grandes bloques:

- Operadores de concatenación.
- Operadores aritméticos.
- Operadores lógicos.
- Operadores de comparación.

Cuando nos encontramos en una expresión operadores de diferentes tipos, Firebird aplica unas reglas de precedencia para establecer el orden en el que se evalúan. Este orden de mayor a menor precedencia es: Operadores concatenación, aritméticos, comparación y lógicos.

6.1.1.- Operadores concatenación.

Aquí nos encontramos con el operador de concatenación de cadenas ||. Permite combinar dos cadenas para formar una sóla.

```
select b.nom_banco || '-' || a.nom_accta a banco_accionista
        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)
```

6.1.2.- Operadores aritméticos.

Las expresiones aritméticas son evaluadas de izquierda a derecha, excepto cuando hay ambigüedad. En este caso se aplica la siguiente tabla:

Operador	Significado	Precedencia
*	Multiplicación	1
/	División	2
+	Suma	3
-	Resta	4

Si en una expresión queremos cambiar el orden de precedencia podemos utilizar los paréntesis.

```
select ba.cod_accta-1000/10+ba.cod_banco
        from ban_accta
select (ba.cod_accta*1000)/(10+ba.cod_banco)
        from ban_accta   -- sería distinto de la sentencia anterior al introducir los paréntesis.
```

6.1.3.- Operadores de comparación.

Los operadores de comparación comprueban la relación específica entre dos valores o entre un valor y un rango de valores. Como resultado se obtiene un valor que puede ser verdadero o unknown.

Se aplica el siguiente orden de precedencia

Operador	Significado	Precedencia
=	Igual a	1
<>, !=, ~=, ^=	Distinto a	2
>	Mayor que	3
<	Menor que	4
>=	Mayor o igual a	5
<=	Menor o igual a	6
!>, ~>, ^>	No mayor a	7
!<, ~<, ^<	No menor a	8

Estos operadores trabajan sobre pares de valores que deben ser del mismo tipo. Cuando se aplican sobre valores aritméticos actuan como en otros lenguajes. Cuando actuan sobre valores de cadenas de caracteres trabajan según el código asignado a cada carácter. Como ya se vio anteriormente, este código depende del conjunto de caracteres y el criterio de orden de los valores involucrados.

Junto a los operadores básicos nos encontramos con los predicados de comparación. Todos ellos tienen a misma precedencia, siempre por debajo de los operadores de comparación. Cuando hay conflicto se evalúan de izquierda a derecha.

Predicado	Significado
BETWEEN … AND …	Valor en el rango
CONTAINING	Contiene una cadena. La comparación no distingue entre mayúsculas y minúsculas
IN	El valor está en un conjunto
LIKE	Igual a una cadena. Se pueden usar plantillas
STARTING WITH	El valor empieza con la cadena indicada
SIMILAR TO	El valor cumple con una expresión regular
IS NULL	Comprueba si un valor es null
IS	Comprueba el valor de verdad de una expresión
IS DISTINCT	Dos valores son distintos

BETWEEN (***valor BETWEEN arg1 AND arg2***) toma dos argumentos del mismo tipo compatible y comprueba si el valor está en el rango definido por los argumentos incluidos estos.

```
select cod_banco
        from bancos
        where cod_banco BETWEEN 10 AND 1000
```

CONTAINING (***valor CONTAINING arg1***) busca en una cadena la aparición de una secuencia de caracteres indicada como argumento. La búsqueda no distingue entre mayúsculas y minúsculas.

```
select *
        from bancos
        where nom_banco CONTAINING 'ban' – obtiene los bancos que contienen en el nombre ban
```

IN (***valor IN ({lista | subconsulta})***)toma como argumento una lista de valores y comprueba si el valor está o no en la lista. Esta lista se puede obtener a partir de una subconsulta de lista.

```
select *
        from bancos
        where cod_banco IN (1,2,10) – obtiene los datos de los bancos cuyo codigo es 1, 2 o 10

select cod_accta
        from ban_accta
        where cod_banco IN (select cod_banco from cotizaciones) – obtiene todos los cod_accta para
                los bancos que han tenido cotizaciones
```

LIKE (*valor LIKE patron [ESCAPE carecer]*)es un operador sensible al contexto (distingue mayúsculas y minúsculas). Se usa para realizar comparaciones con patrones de caracteres. Se pueden utilizar dos caracteres especiales:

- %: puede ser sustituido por cualquier carácter cualquier número de veces, incluida ninguna. Por ejemplo '%an%' puede ser sustituido por 'b**an**co', 'el b*an*esto', pero no por 'BANCO' ni 'BaNesto'.
- _ puede ser sustituido por cualquier carácter una sóla vez. Por ejemplo '_o_a' puede ser sustituido por '*c*osa', '*col*a', '*M*o*T*a' pero no por 'rebosa' ni 'come'.

Si se buscan cadenas que incluyen algún carácter especial se usa un carácter de escape.

```
select *
        from bancos
        where nom_banco LIKE '%an%'

select *
        from bancos
        where nom_banco LIKE 'B_nco %'

select *
        from bancos
        where nom_banco LIKE '%#_%' ESCAPE '#' – bancos que contienen en su nombre un
                subrayado
```

LIKE nunca hace las búsquedas sobre indices aunque estén disponibles.

STARTING WITH (***valor STARTING WITH arg***) es equivalente a LIKE 'arg%'. Busca la aparición de una cadena al inicio de la misma. Puede usar un índice si está disponible.

```
select *
        from bancos
        where nom_banco STARTING WITH 'Ban' – devuelve los bancos cuyo nombre empieza por
                Ban
```

SIMILAR TO (***valor SIMILAR TO 'expresión regular' [ESCAPE 'carácter']***). Busca si valor cumple con la expresión regular indicada como parámetro.

```
select *
        from bancos
        where nom_banco SIMILAR TO '[BC]%a' – devuelve los bancos cuyo nombre empieza por Bo
                C y acaban con a
```

IS NULL (***valor IS [NOT] NULL)*** permite comprobar si un valor es nulo o no. Hay que recordar que el valor NULL es un valor especial que indica 'nada' o 'desconocido' y que tiene sus propias implicaciones en la evaluación de las expresiones.

```
select *
        from bancos
        where pre_ac_ac IS NOT NULL – selecciona los bancos para los que se ha establecido el
                campo pre_ac_ac (se les ha asignado un valor)
```

IS (***valor IS [NOT] [TRUE| FALSE| UNKNOWN])*** permite comprobar si un valor es verdadero, falso o desconocido.

```
select *
        from bancos
        where (pre_ac_ac<12) IS TRUE– selecciona los bancos para los que el precio de acción es
                menos de 12
```

IS DISTINCT (***valor IS [NOT] DISTINCT FROM valor)*** permite realizar una comprobacion al estilo <> con una diferencia, con este predicado podemos comprobar si NULL es distinto de cualquier otro resultado

```
select *
        from bancos
        where correcto IS DISTINCT FROM null – selecciona los bancos para los que se ha
                establecido el campo correcto es distinto de null
```

6.1.4.- Operadores lógicos.

Firebird soporta tres operadores lógicos: NOT, OR y AND.

- NOT niega la condición sobre la que opera.
- AND une dos condiciones. Sólo vale true cuando todas las condiciones son ciertas.
- OR une dos condiciones. Vale true si alguna de las condiciones es cierta. Firebird utiliza una ejecución corta que hace que se devuelva true tan pronto encuentre una condición true, sin evaluar las condiciones siguientes.

El orden de precedencia en los operadores es NOT, AND y OR, en orden de mayor a menor precedencia. Este orden puede ser cambiado mediante los ().

```
select *
        from bancos
        where cod_banco > 1000 AND
                (pre_ac_ac between 0 and 100 OR pre_ac_ac>1000) AND
                NOT nom_banco like '%ban%'
```

6.1.5.- Predicados existenciales.

Son aquellos predicados en los que se pueden establecer condiciones sobre valores devueltos por una subconsulta. Los predicados disponibles son:

Predicado	Significado
ALL	Comprueba si la comparación es igual para todos los valores devueltos por la subconsulta
[NOT] EXISTS	Existe (o no) alguna fila en la la subconsulta
[NOT] IN	Existe (o no) en al menos un valor devuelto por la subconsulta

[NOT] SINGULAR	Comprueba si exactamente un valor es devuelto por la subconsulta. Si la subconsulta devuelve NULL o más de una fila, SINGULAR devuelve false.
SOME	Comprueba si la comparación es true para al menos un valor devuelto por la subconsulta
ANY	Equivalente a SOME

EXISTS (***[NOT] EXISTS(subconsulta)***), permite comprobar si existen o no filas en la subconsulta indicada. Es el mecanismo más rápido para comprobar la existencia de filas en un conjunto de datos (no usar count(*) por su alto coste de cálculo). No genera un conjunto de salida. Este predicado recorre el conjunto de datos hasta que encuentra una fila que cumple la condición, en ese momento finaliza la búsqueda. Si no encuentra ninguna fila devuelve false.

```
select *
        from bancos
        where EXISTS(select * from accionistas) -- devuelve todos los bancos si hay accionistas en
                la base de datos
```

Normalmente se usa en combinación con la consulta principal para comprobar la existencia de filas en la subconsulta en las que se relaciona una o varias columnas con la consulta principal.

```
select b.*
        from bancos b
        where EXISTS(select * from ban_accta ba
                        where ba.cod_banco=b.cod_banco) -- devuelve todos los bancos que
                tienen accionistas
```

Se utiliza para implementar la diferencia del álgebra relacional. Para ello se utiliza el NOT EXISTS () en el que se unen la subconsulta con la consulta principal. Se tiene que trabajar con el NOT ya que se evalua la no existencia de filas con la condición indicada en la subconsulta. Por ejemplo, para saber los bancos que no tienen accionistas, podríamos tener:

```
select b.*
        from bancos b
        where NOT EXISTS(select * from ban_accta ba
                        where ba.cod_banco=b.cod_banco) -- devuelve todos los bancos que
                no tienen accionistas, es decir, a todos los bancos les quitamos aquellos bancos
                que tienen algún accionistas.
```

IN (***valor [NOT] IN (subconsulta)***). Es semejante al operador EXISTS con la diferencia que ya se realiza una unión entre un valor y el campo devuelto en el conjunto de datos obtenido en la subconsulta. Al igual que con EXISTS se evaluan las filas hasta que se encuentra valor en la subconsulta. En este caso se devuelve true. Si se evalúa todo el conjunto resultado de la subconsulta sin que se encuentre el valor, se devuelve false.

```
select *
        from bancos
        where cod_banco IN(select cod_banco from ban_accta) -- devuelve todos los bancos que
                tienen accionistas.
```

Al igual que con el NOT EXISTS, el NOT IN se utiliza para implementar el operador diferencia del álgebra relacional. En el caso anterior, para obtener los bancos que no tienen accionistas podríamos tener:

```
select *
        from bancos
```

```
        where cod_banco NOT IN(select cod_banco from ban_accta) -- devuelve todos los bancos
                que tienen accionistas.
```

ALL (***valor {***<|>|=|<>|***…}ALL (subconsulta)***). Comprueba que el valor sea menor, mayor, igual, etc que **todos** los valores de columna devueltos en conjunto de datos de la subconsulta.

```
select *
        from bancos
        where pre_ac_ac >= ALL
                (select pre_ac_fe from cotizaciones) -- devuelve los bancos cuyo pre_ac_ac es mayor
                        o igual a todas las cotizaciones existentes, es decir, se haya los bancos cuya
                        su precio actual es igual al máximo de las cotizaciones.
```

Igual que con los otros operadores existenciales pueden usarse enlazando la subconsulta con la consulta principal.

```
select *
        from bancos b
        where b.pre_ac_ac >= ALL
                (select pre_ac_fe
                        from cotizaciones c
                        where b.cod_banco=c.cod_banco) -- devuelve los bancos cuyo pre_ac_ac
                        es mayor o igual a todas las cotizaciones existentes para ese banco, es
                        decir, se haya los bancos cuya su precio actual es igual al máximo de las
                        cotizaciones suyas.
```

SINGULAR (***SINGULAR (subconsulta)***). Este predicado es equivalente a un exists en el la subconsulta como resultado una y sólo una fila.

```
select *
        from bancos b
        where SINGULAR
                (select *
                        from cotizaciones c
                where b.cod_banco=c.cod_banco
                        and b.pre_ac_ac>=c.pre_ac_fe) -- devuelve los bancos cuyo pre_ac_ac es
                        mayor o igual a la única cotizacion existente para ese banco. Si la subconsulta
                        devuelve mas filas falla
```

ANY (***valor {***<|>|=|<>|***…} ANY(subconsulta)***) y **SOME** (***valor {***<|>|=|<>|***…} SOME (subconsulta)***) son dos predicados idénticos. Son true cuando se cumple la condición en la que valor es menor, mayor, etc que algún valor devuelto por la subconsulta.

```
select *
        from bancos b
        where b.pre_ac_ac > SOME
                (select pre_ac_fe
                        from cotizaciones c
                        where b.cod_banco=c.cod_banco) -- devuelve los bancos cuyo pre_ac_ac
                        es mayor que el de alguna cotización existente para el banco.
```

6.2.- Funciones.

En las expresiones es posible usar funciones para realizar diferentes operaciones. En Firebird se trabaja con dos tipos de funciones: funciones internas y funciones definidas por el usuario (UDF). Las funciones internas son aquellas que implementa Firebird en su núcleo y que están siempre disponibles. Las funciones UDF son funciones creadas en otros lenguajes como C y que se incorporan a una Base de Datos. Normalmente se incorpora a la base de datos un fichero con la interfaz de las funciones definidas dentro de archivos DLL. Estos ficheros deberán estar accesibles en el momento en el que se quieran usar las funciones.

En este tema trataremos las funciones internas.

Las funciones internas se pueden agrupar en diferentes bloques:
- De conversión. Usadas en la conversión de datos como CAST.
- Case. Familia de funciones derivadas de CASE, como CASE, COALESCE y NULLIF.
- De agregación. Funciones que se usan para obtener resultados sumarizados de grupos de datos. Tenemos funciones como SUM, COUNT, AVG, etc.
- Generales. Funciones de gestión general.
- Numéricas. Funciones para tratamientos numéricos.
- Cadenas. Funciones para tratamiento de cadenas.
- De fecha y hora. Funciones para tratamiento de fechas y horas.

6.2.1.- Funciones de conversión.

Son funciones que premiten transformar tipos de datos. Muchas funciones de otros bloques se pueden considerar también como de conversión, en tanto convierten entre distintos tipos de datos. Por ejemplo podríamos tener EXTRACT (para fechas/horas), ASCII_VAL (para cadenas), etc.

Nos encontramos en este bloque con
CAST (***Cast(valor|NULL AS tipo_dato)***). Convierte el valor al tipo de dato indicado. Para que esto se pueda hacer es necesario que valor sea compatible con tipo_dato. Por ejemplo si convierto valor (que es cadena) a integer, valor debe ser una cadena con valores numéricos.

```
Select cod_banco, CAST(cod_banco as CHAR(5)) || ' ' || nom_banco as identificación
    from bancos
```

6.2.2.- Funciones CASE.

Nos encontramos aquí aquellas funciones relacionadas con la funcion CASE definida en el estandar SQL-99. Se obtiene un resultado a partir de una condición que se evalúa en tiempo de ejecución.

CASE. En esta función se indican una o más condiciones obteniéndose un resultado por condición. Se empiezan a evaluar las condiciones y se devuelve el resultado correspondiente a la primera que se evalúa como trae. Si no se cumple ninguna y se tiene cláusula ELSE, se devolverá el resultado indicado en esta cláusula. Si no se cumple ninguna condición ni se tienen cláusula ELSE, la función devolverá NULL.

Tiene definidas dos sintaxis:

```
CASE elemento
    WHEN {NULL | valor1} THEN {resultado1 | NULL}
    ……..
    WHEN {NULL | valorN} THEN {resultadoN | NULL}
    [ELSE {resultadoN+1 | NULL}
```

END

CASE
WHEN condicion1 THEN {resultado1 | NULL}
……..
WHEN condicionN THEN {resultadoN | NULL}
[ELSE {resultadoN+1 | NULL}
END

La primera sintaxis se usa cuando las condiciones se establecen sobre valores de un elemento (una columna, una variable). La segunda se utiliza cuando queremos establecer condiciones especiales para cada caso.

```
Select cod_banco,
    case cod_banco
        when 0 then 'Banco nuevo'
        when 1 then 'Banco especial'
        else nom_banco
    end as tipo
    from bancos

Select cod_banco,
    case
        when cod_banco<1000 then 'Banco antiguo'
        when (cod_banco between 1000 and 5000) then 'Banco normal'
        else 'Banco moderno'
    end as tipo
    from bancos
```

COALESCE (***COALESCE(valor1, {valor2, [,… valor n]})***). Permite que una columna se rellene con el primer valor que no sea nulo. Para que funcione correctamente, se debe indicar un valor no nulo como el último en la lista (normalmente un literal: número, cadena, …..).

Seria equivalente a

```
CASE
    WHEN valor1 IS NOT NULL THEN valor1
    ……..
    WHEN valorn-1 IS NOT NULL THEN valorn-1
    ELSE valor n
END
```

```
Select cod_banco, COALESCE(nom_banco,'Sin indicar') as nom_banco
    from bancos
```

NULLIF (***NULLIF(valor1, valor2)***). Si valor 1 es igual a valor2 se devuelve NULL, en otro caso se devuelve valor1.

Sería equivalente a

```
CASE
    WHEN valor1=valor2 THEN NULL
    ELSE valor1
END
```

```
Update bancos set pre_ac_ac=NULLIF(pre_ac_ac,0) – se pone el campo pre_ac_ac a NULL si vale 0
```

DECODE (**DECODE(expresion, valor1, resultado1,[…., valorN, resultado N] [resultado_default])**) Es equivalente a

```
CASE expresión
        WHEN valor1 THEN resultado1
        …
        WHEN valorN THEN resultadoN
        ELSE resultado_default
END
```

IIF (***IIF(condicion, valor1, valor2)***). Es equivalente a

```
CASE
        WHEN condicion THEN valor1
        ELSE valor2
END
```

6.2.3.- Funciones de agregación.

Son funciones que obtienen resultados a partir de bloques de valores. Suelen usarse en conjunción con sentencias con cláusula GROUP BY en las que se pretende obtener resultados sobre los conjuntos obtenidos.

COUNT (***COUNT ([DISTINCT] columna | *)***) Permite contar ocurrencias de filas (*) o columnas. Permite además contar valores distintos de columna (con DISTINCT).

SUM (***SUM(columna)***). Suma los valores de una columna determinada.

AVG (***AVG(columna)***). Obtiene la media de los valores de una columna.

MAX (***MAX(columna)***). Obtiene el máximo de los valores de una columna.

MIN (***MIN(columna)***). Obtiene el mínimo de los valores de una columna.

```
Select cod_banco,
        COUNT(*) as num_bancos,
        SUM(pre_ac_ac) as suma_precio_actual,
        AVG(pre_ac_ac) as precio_medio,
        MIN(pre_ac_ac) as precio_minimo,
        MAX(pre_ac_ac) as precio_maximo
    From bancos
    Group by cod_banco
```

LIST ***(LIST([{ALL | DISTINCT}] expresion [, delimitador])***). Devuelve una cadena que es la concatenación de todos los valores de un grupo no nulos usando el delimitador indicado. ALL es el valor por defecto. Si se omite delimitador se usa ,.

```
select cod_banco, list(cod_accta, '-') as lista_accionistas
    from ban_accta
group by cod_banco
```

6.2.4.- Funciones de ventana (funciones analíticas).

Son funciones de agregación que no "filtran" el conjunto resultado de una consulta. Los resultados de aplicar las funciones de agregación son mezclados en las mismas filas de los datos de la consulta, es decir, no se generan filas agrupación sino que la operación de agregación aparece en la propia fila de donde se obtienen los datos.

Las funciones ventana son usadas con la cláusula **OVER**. Esta cláusula solo puede aparecer en la parte Select o en la parte ORDER BY de una consulta.

En una función de ventana se puede indicar un particionado y un criterio de ordenación.

La sintaxis de las funciones de ventana sería:

<función_ventana> := <nombre_funcion_ventana>([<exp> [,<exp>…]])
OVER(
[PARTITION BY <exp> [,<exp> …]]
[ORDER BY <exp> [<dirección>] [<lugar_nulos>]
[, <exp> [<dirección>] [<lugar_nulos>]…]
)
<dirección> := {ASC | DESC}
<lugar_nulos := NULLS {FIRST | LAST}

Cuando aplicamos OVER () sin nada más obtenemos el resultado de la función de agregación sobre el total de registros del conjunto resultado.

Si indicamos PARTITION BY, establecemos que expresión debe usarse para agrupar/particionar el conjunto. A cada partición se aplicará la función de agregación. Por tanto, como resultado se obtendrán uno o mas valores, enlazando cada valor a la/s fila/s que cumple con los mismos criterios de agrupación/particionado.

Si indicamos ORDER BY, se irá aplicando la función de agregación según se vayan obteniendo las filas (por la expresión de ORDER BY). Si encontramos que hay varias filas que coinciden en los valores de la expresión de ORDER BY, se agrupan obteniendo un solo valor resultado de aplicar la función de agrupación sobre los elementos que la forman.

Como funciones ventana pueden aplicarse las funciones de agregación vistas en el apartado anterior (excepto list).

Además existen unas funciones ventana específicas que podemos agrupar en:

- Funciones de rango: indican la posición (rango) de una fila en la partición.
  - **DENSE_RANK()**: Devuelve la posición según el orden comenzando desde 1. Si hay varias filas con igual valor para la expresión, tendrán el mismo rango. Por ejemplo si tengo tres filas con el mismo valor que ocupan el puesto 3 tendrán como rango 3. La siguiente fila será la 4
  - **RANK()**: Devuelve la posición según el orden comenzando desde 1. Si hay varias filas con iguan valor para la expresión, tendrán el mismo rango. Se diferencia de la anterior en el número que asignaría a la siguiente fila. Por ejemplo si tengo tres filas con el mismo valor que ocupan el puesto 3 tendrán como rango 3. La siguiente fila será la 6 (tengo en cuenta que en el puesto 3 hay tres filas).
  - **ROW_NUMBER()**: Devuelve la posición según el orden comenzando desde 1. Este valor es semejante a un autonumérico. Por lo tanto cada fila tendrán un valor distinto.
- Funciones de navegación: añaden el valor de otra fila en la fila actual.
  - **FIRST_VALUE(<exp>)**. El primer valor dada la expresión <exp>.
  - **LAST_VALUE(<exp>)**. El ultimo valor dada la expresión <exp>.

- **NTH_VALUE(<exp>, <desp>) [FROM FIRST | FROM LAST]**. Devuelve el valor que ocupa la posición <desp> según la expresión <exp>. Esto se hará bien desde el principio o desde el final.
- **LAG(<exp>, [, <desp> [, <defecto>]])**. Mira el valor correspondiente a <desp> filas anteriores de la fila actual dada la <exp>. Si se sale fuera del rango de filas existentes se devuelve null (o <defecto> si se establece).
- **LEAD(<exp>, [, <desp> [, <defecto>]])**. Mira el valor correspondiente a <desp> filas posteriores dada la <exp>. Si se sale fuera del rango de filas existentes se devuelve null (o <defecto> si se establece).

6.2.5.- Funciones generales.

Nos encontramos aquí funciones difícilmente ubicables en los otros bloques.

GEN_ID (***GEN_ID(generador, valor_paso)***). Se usa con los generadores. Calcula un nuevo valor para el generador y devuelve el valor obtenido. El nuevo valor se obtiene añadiendo al actual el valor_paso. Normalmente se usa como valor_paso el 1. Si usamos el 0, obtenemos el último valor generado.

```
Select GEN_ID(generador,1) as numero FROM RDB$DATABASE
```

GEN_UUID (***GEN_UUID()***). Devuelve un identificador universal único. Un UUID se define en el formato de char(16) de OCTECTS

CHAR_TO_UUID (***CHAR_TO_UUID(cadena)***), **UUID_TO_CHAR** (***UUID_TO_CHAR(cadena)).*** Permite realizar conversiones de UUID entre su formato de char(16) OCTECT y char(32) ASCII ((XXXXXXXXXXXXX-XXXX-XXXX-XXXXXXXXXXXX).

```
select char_to_uuid('93519227-8D50-4E47-81AA-8F6678C096A1')
        from rdb$database;
select uuid_to_char(gen_uuid())
        from rdb$database;
```

RDB$GET_CONTEXT (***RDB$GET_CONTEXT(espacio, variable)***)
RDB$SET_CONTEXT (***RDB$GET_CONTEXT(espacio, variable, valor)***). Estas funciones de sistema se usan para acceder a las variables de contexto, así como para asignar y devolver datos de usuario y datos asociados a una transacción o conexión.

Se pueden usar los siguientes espacios:

- *USER_SESSION*: Permite trabajar con variables definidas por el usuario en la sesión actual (conexión con el servidor).
- *USER_TRANSACTION*: Permite trabajar con variables definidas por el usuario en la transacción actual.
- *SYSTEM*: Permite acceder en modo de solo lectura a las siguiente variables.
  - **NETWORK_PROTOCOL** :Protocolo de red usada por el cliente. Los posibles balores son Currently used values: “TCPv4”, “WNET”, “XNET” y NULL.
  - **CLIENT_ADDRESS** La dirección remota del cliente como cadena. The wire protocol address of the remote client, represented as a string. La dirección IP si el protocolo es TCPv4, en la forma "xxx.xxx.xxx.xxx". El ID de proceso local si se usa XNET y NULL para cualquier otro protocolo.
  - **DB_NAME** Nombre de la base de datos actual. Será el alias o el nombre completo incluyendo la ruta.

- **ISOLATION_LEVEL** Nivel de bloqueo para la transacción actual. El valor puede ser "READ COMMITTED", "SNAPSHOT", "CONSISTENCY".
- **TRANSACTION_ID** Identificador de la transacción actual. Es equivalente a la variable CURRENT_TRANSACTION.
- **SESSION_ID** Identificador de sesión actual. Es equivalente a la variable CURRENT_CONNECTION.
- **CURRENT_USER** Usuario actual. Es equivalente a CURRENT_USER y USER.
- **CURRENT_ROLE** Role usado en la conexión. Es equivalente a la variable CURRENT_ROLE.

```
select RDB$SET_CONTEXT('USER_SESSION','edad',20)
    from rdb$database – rellena la variable edad definida en el espacio de usuario de sesion a 20.
select RDB$SET_CONTEXT('USER_SESSION','edad',NULL)
    from rdb$database – se elimina la variable edad.
select RDB$GET_CONTEXT('SYSTEM','NETWORK_PROTOCOL')
    from rdb$database
```

6.2.6.- Funciones numéricas.

Se incluyen aquí funciones para tratamiento de valores numéricos.

ABS (***ABS(numero)***). Valor absoluto de un número.

ACOS, ASIN, ATAN (***funcion(numero)***). Arcocoseno, arcoseno, arcotangente de un número entre -1 y 1. El resultado está en el rango -Π/2 y Π/2.

ACOSH, SINH, ATANH (*funcion(numero*)). Arcoseno, arcocoseno y arcotangente hiperbólico de un número expresado en radiane,

ATAN2 ***(ATAN2(valor1,valor2)***). Devuelve el arcotangente de valor1/valor2.

BIN_AND, BIN_OR, BIN_XOR (***funcion(numero[, numero...])***) Realiza la and, or y xor a nivel binario sobre los números que se pasan como parámetros.
BIN_NOT(numero). Realiza la negación a nivel de bits del número.

BIN_SHL, BIN_SHR (***funcion(numero1,numero2)***). Realiza un desplazamiento a nivel de bits del primer número hacia la izquierda (numero1 << numero2) o derecha (numero1 >> numero2) tantas veces como nos indica el segundo.

CEIL, CEILING, FLOOR (***funcion(numero)***). Devuelve el entero más pequeño que es mayor o igual a número (redondeo por arriba) (CEIL, CEILING) o el entero mayor que es menor o igual a número (redondeo por abajo) (FLOOR).

COS, COSH, COT, SIN, SINH, TAN, TANH (***funcion(numero)***). Coseno, coseno hiperbólico, cotangente (1/tan(numero), seno, seno hiperbólico, tangente y tangente hiperbólica de un número.

EXP, LN (***funcion(numero)***). Devuelve respectivamente la potencia número en base e y el logaritmo neperiano de número.

LOG (***LOG(numero1,numero2)***). logaritmo en base numero1 de numero2.

LOG10 (***LOG10(numero)***). Logaritmo en base 10 de numero.

MAXVALUE, MINVALUE (***funcion(valor1,…., valorn)***). Devuelven respectivamente el máximo y el mínimo de la lista de valores.

MOD (***MOD(numero1,numero2)***). Resto entero de la divisione de numero1 entre numero2.

PI (***PI()***). Devuelve el valor de PI.

POWER (***POWER(numero1,numero2)***). Devuelve numero1 elevado a numero2.

RAND (***RAND()***). Devuelve un número aleatorio en el rango 0 a 1.

ROUND (***ROUND(numero[,escala])***). Redondea un número a la escala indicada. Si la escala es negativa o se omite, se redonde la parte entera.

ROUND(1234.567,0) – devuelve 1235.000
ROUND(1234.567,1) – devuelve 1234.600
ROUND(1234.567,-1) – devuelve 1230.000.

SIGN (***SIGN(numero)***). Devuelve 1, 0 o .-1 dependiendo de si número es positivo, cero o negativo.

SQRT (***SQRT(numero)***). Obtiene la raiz cuadrada de numero.

TRUNC (***TRUNC(numero[,escala])***). Devuelve la parte entera hasta la escala indicada.

TRUNC(1234.567) – devuelve 1234
TRUNC(1234.567,1) – devuelve 1234.500
TRUNC(1234.567,-1) – devuelve 1230.000.

6.2.7.- Funciones sobre cadenas.

Se incluyen aquí funciones que trabajan sobre cadenas o que nos permiten obtener cadenas.

ASCII_CHAR (***ASCII_CHAR(numero)***). Devuelve el carácter cuyo código ASCII es número. Número debe estar entre 0 y 255.

ASCII_VAL (***ASCII_VAL(cadena)***).Devuelve el código ASCII del primer carácter de la cadena. Si la cadena está vacía devuelve 0. Si el primer carácter es multibyte genera un error.

BIT_LENGTH, CHAR_LENGTH, OCTET_LENGTH (***funcion(cadena)***). Devuelven respectivamente el número de bits, el número de caracteres y el número de bytes que ocupa la cadena. Las dos últimas darán resultados s la cadena es de un tipo de caracteres multibyte.

HASH (***HASH(cadena)***). Devuelve el hash de la cadena.

LEFT, RIGHT (***funcion(cadena,numero)***). Devuelve los numero caracteres de la parte izquierda o derecha de una cadena. La cadena empieza en la posición 1.

LOWER, UPPER (***funcion(cadena)***). Develve la cadena convertida a minúscula o mayúscula.

LPAD, RPAD (***funcion(cadena1, longitud[, cadena2])***). Añade cadena2 al inicio o al final de cadena1 hasta que el tamaño de la cadena resultante sea longitud. Si se omite cadena2, se añaden espacios en blanco.

OVERLAY (***OVERLAY(cadena1 PLACING cadena2 FROM inicio [FOR longitud])***). Permite sustituir en cadena1 los longitud caracteres desde inicio por cadena2. Si se omite longitud, se sustituyen la longitud de cadena 2.

Es equivalente a SUBSTRING(cadena1, 1 FOR inicio - 1) || cadena2 || SUBSTRING(cadena1, inicio + longitud)

POSITION (***POSITION(cadena2 IN cadena1), POSITION(cadena2,cadena1 [, desplazamiento])***). Obtiene la primera posición de cadena2 en cadena1. En la segunda sintaxis se puede indicar una posición desde la que comenzar a buscar.

REPLACE (***REPLACE(cadena, cad_a_buscar, cad_reemplazo)***). Reemplaza en cadena todas las apariciones de cad_a_buscar con cad_reemplazo.

REVERSE (***REVERSE(cadena)***). La cadena invertida.

SUBSTRING (***SUBSTRING(valor FROM pos_ini [FOR longitud])***). Permite obtener una subcadena de una dada (valor) empezando en la pos_ini y con un tamaño máximo longitud. Pos_ini y longitud se deben evaluar a un valor >=1.

Substring permite también la búsqueda usando expresiones regulares con la siguiente sintaxis:

SUBSTRING(valor [NOT] SIMILAR TO <patrón similar> ESCAPE <carácter escape>)

TRIM (***TRIM ([[LEADING | TRAILING | BOTH] [cadena_borrar] FROM] cadena)***. Permite borrar la cadena_borrar al inicio (LEADING), final (TRAILING) o en ambas partes de cadena. Por defecto se tendrá BOTH y como cadena_borrar ‘ ‘.

6.2.8.- Funciones de fecha y hora.

Son funciones para manejo de fechas y horas.

DATEADD (***DATEADD(numero <parte> TO fecha_hora), DATEADD(<parte>,numero, fecha_hora)***). Devuelve una fecha/hora a partir de fecha_hora con <parte> incrementada (decrementada) en número. <parte> puede ser {YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | MILLISECOND}

DATEADD(-1 DAY TO CURRENT_DATE) –seria ayer

DATEDIFF (***DATEDIFF(<parte> FROM fecha_hora_ini TO fecha_hora_fin), DATEDIFF(<parte>, fecha_hora_ini, fecha_hora_fin)***). Devuelve un valor numérico que representa la diferencia entre las fechas dadas para la parte indicada. Este valor puede ser positivo (fecha_hora_ini < fecha_hora_fin), 0 (fecha_hora_ini = fecha_hora_fin) o negativo (fecha_hora_ini > fecha_hora_fin) Parte puede ser {YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | MILLISECOND}.

EXTRACT (***EXTRACT(parte FROM campo)***). Extrae la parte indicada de un campo de tipo fecha segun la tabla siguiente:

PARTE	TIPO DEVUELTO	RANGO	DESCRIPCIÓN
YEAR	SMALLINT	0-5400	Año de la fecha o timestamp
MONTH	SMALLINT	1-12	Mes de la fecha o timestamp
DAY	SMALLINT	1-31	Día de la fecha o timestamp
HOUR	SMALLINT	0-23	Hora de la hora o timestamp
MINUTE	SAMLLINT	0-59	Minutos de la hora o timestamp
SECOND	DECIMAL(6,4)	0-59.9999	Segundos de la hora o timestamp
WEEKDAY	SMALLINT	0-6	Día de la semana (0: domingo,…, 6: sabado)
YEARDAY	SMALLINT	1-366	Día del año de la fecha o timestamp
WEEK	SMALLINT	1-52	Semana del año de la fecha o timestamp
MILLISECOND	INTEGER		Milisefundos de la hora o timestamp

```
Select EXTRACT(YEAR FROM pre_ac_fe) as anio
    From bancos
```

6.2.10.- Funciones estadísticas.

Son funciones que nos permite la realización de diversas operaciones estadísticas. Se pueden dividir en dos grupos:

a) Funciones de agregación.
VAR_SAMP (<expr>): Devuelve la Varianza Simple de <expr>
VAR_POP(<expr>): Devuelve la Varianza de Población de <expr>.
STDDEV_SAMP(<expr>): Devuelve la Desviación Estandar Simple de <expr>.
STDDEV_POP(<expr>): Devuelve la Desviación Estándar de Población de <expr>
COVAR_SAMP(<expr1>, <expr2>): Devuelve la Población Simple de <expr1> y <expr2>.
COVAR_POP(<expr1>, <expr2>): Devuelve la Covarianza de la Población de <expr1> y <expr2>.
CORR(<expr1>, <expr2>): Devuelve el Coeficiente de Correlaci de <expr1> y <expr2>.

b) Funciones de regresión linear.
REGR_AVGX(X, Y): Devuelve la media de X (independiente de Y).
REGR_AVGY(X, Y): Devuelve la media de Y (independiente de X).
REGR_AVGY(X, Y): Devuelve la media de Y (independiente de X).
REGR_COUNT(X, Y): Devuelve el número de pares (X, Y). Solo se tiene en cuenta los pares en los que X e Y son distintos de NULL.
REGR_INTERCEPT(X, Y): Devuelve el punto de corte en y de la línea de regresión determinada por el conjunto de pares (Y, X).
REGR_SLOPE(X, Y): Devuelve la inclinación de la línea de regresión dada por los pares los pares (Y, X).
REGR_SXX(X, Y): Devuelve la suma de los cuadrados de la expresión Y en los pares (Y, X).
REGR_SYY(X, Y): Devuelve la suma de los cuadrados de la expresión dependiente en los pares (Y, X).

6.3.- Expresiones regulares.

Una expresión regular permite definir un patrón de caracteres con el que podremos establecer si una cadena cumple o no con el mismo. Se utilizan en el predicado SIMILAR TO y permiten extender las búsquedas que se pueden realizar mediante el predicado LIKE (mucho mas limitado).

El predicado SIMILAR TO tiene como sintaxis
valor [NOT] SIMILAR TO <patrón similar> [ESCAPE <carácter escape>]

cadena SIMILAR TO 'A1' – cadena debe cumplir con el patrón 'A1'
'se1' SIMILAR TO '%1' – true ya que se1 cumple con el patron %1.

Una expresión regular está formada por una o más caracteres o símbolos acompañados de otros con un significado especial. Son expresiones regulares 'AA', 'A01' o 'abcd' que se emparejarían con cadenas con exactamente esos caracteres, o expresiones como 'A*B', '[1-5]{2,}' en las que aparecen caracteres con significados especiales.

Los caracteres que se utilizan son:

Caracter	Busca	Ejemplo
|	Indica uno de dos posibles valores	ab | cd | ef Encuentra "ab" o "ef" pero no "abc".
_	Indica cualquier carácter no vacío	_ Encuentra "a", "1" pero no "" ni "a1"
%	Indica una cadena de cualquier longitud incluyendo la cadena vacía	% Encuentra "", "asssad". A%A Encuentra "AA", "AbbdgA", pero no "A" ni "bA"
()	Agrupa varios caracteres para que se tomen como uno sólo	(a2)* Encuentra "a2", "a2a2a2" pero no "aa2a22"
[…]	Cualquier carácter que se incluya dentro	[em]sto Encuentra "esto" y "msto" pero no "asto"
[^…]	Ninguno de los caracteres incluidos	[^em]sto Encuentra "asto" pero no "esto"

Además se pueden usar unos determinados caracteres para indicar el número de repeticiones del carácter que le precede.

Caracter	Con el que lo precede	Ejemplo
*	0 o mas veces	co*che Encuentra "cche", "coche" y "coooooche" pero no "cache".
?	0 o 1 vez	co?che Encuentra "coche" pero no "cooche"
+	1 o mas veces	co+che Encuentra "coche" pero no "cche"
{n}	Exactamente n veces	co{3}che Encuentra "coooche.
{n,}	N veces o mas	co{3,}che Encuentra "coooooche" pero no "cooche"
{n,m}	Entre n y m veces	co{1,2}che Encuentra "coche" y "cooche" pero no "coooche"

Ya que es muy normal referirnos a un carácter alfabético o a una cifra decimal, se definen los identificadores de clases de caracteres:

Identificador	**Descripción**	**Notas**
ALPHA	Todos las letras simples latinas (a-z, A-Z)	Se incluyen letras con acentos cuando el criterio de ordenación es

		insensibe a los acentos
UPPER	Todas las letras mayúsculas latinas (A-Z)	Se incluyen las letras en minúsculas si el criterio de ordenación no distingue el caso
LOWER	Todas las letras minúsculas latinas (a-z)	Se incluyen las letras en mayúsclas si el criterio de ordenación no distingue el caso
DIGIT	Todos los dígitos numéricos (0-9)	
SPACE	Carácter de espacio (ASCII 32)	
WHITESPACE	Todos los caracteres de espacio (tabulador vertical (9), nueva linea (10), tabulador horizontal (11), retorno de carro (13), inicio de linea (12), espacio (32))	
ALNUM	Todos los caracteres que son letras latinas (ALPHA) o dígitos numéricos (DIGIT)	

Por ejemplo podríamos tener

'4' SIMILAR TO '[[:DIGIT:]]' -- Verdadero
'a' SIMILAR TO '[[:DIGIT:]]' -- falso
'4' SIMILAR TO '[^[:DIGIT:]]' -- falso
'a' SIMILAR TO '[^[:DIGIT:]]' -- verdadero

En determinadas ocasiones es necesario incluir un carácter especial pero sin que tenga un significado especial. En este caso se utiliza un carácter de escape:

telefono similar to '\([0-9]{3}\) [0-9]{6}' escape '\' – define un número de teléfono como '(111) 111111'

6.3.1.- Casos de uso.

Como ya se ha indicado anteriormente y como se verá en temás posteriores, las expresiones se pueden utilizar en numerosas sentencias. Aquí veremos unos ejemplos con los que ubicar el uso de las mismas.

En sentencias de DDL.

```
CREATE TABLE prueba (
        numero integer CHECK (numero in (0,1)), -- expresión para indicar los valores posibles
        numero_texto COMPUTED BY (case numero when 0 then 'BASE' when 1 then 'GRANDE' end),  – expresión para definir un campo calculado
        fecha DATE DEFAULT CURRENT_DATE) – fecha tiene como valor por defecto una variable de contexto.
```

En sentencias de DML.

```
select (fecha+1) as dia_despues, cast(numero as char(5)) as numero1 – expresiones en los campos devueltos
        from prueba
        where fecha between '01/01/2008' and '07/01/2008' – en la cláusula where

update prueba
        set fecha=current_date – expresión en una sentencia update
```

En módulos de PSQL

```
if (numero not in (select ……..)) then
        ….
numero = numero + 25;
```