Tutorial Nº 4
Unidad 4: Introducción a Tópicos Avanzados
Taller de Bases de Datos

Profesora: Eliana Providel Godoy - *eliana.providel@uv.cl*

Valparaíso, 2015

Resumen

El documento incluye lo conceptos teóricos abarcados en la unidad, como son la gestión de vistas, índices, bases de datos remotas y distribuídas. Para cada uno se entrega una explicación ejemplificada, se define la sintaxis junto con algunos ejemplos y se entregan datos adicionales sobre el tema.

1. Indexación

1.1. Definición

Un **índice** se define como una estructura de datos adicional, la cual es usada para optimizar la velocidad de recuperación de registros en respuestas a ciertas condiciones de búsqueda. Generalmente, estas proporcionan caminos de acceso secundario u alternativos para acceder a los registros, sin que se afecte la posición física de estos en el fichero. Esto permite un acceso eficaz a los registros, basándose en la indexación de los campos que se utilizan para construir el índice. [1]

La idea tras la estructura de acceso de un índice ordenado es parecida a la que hay tras el índice de un libro, que enumera al final de la obra los términos importantes ordenados alfabéticamente, junto con las páginas en las que aparecen. Podemos buscar en un índice en busca de los números de páginas y utilizar esas ubicaciones para localizar un término en el libro, buscando en las páginas especificadas. La alternativa, de no indicarse lo contrario, sería desplazarse lentamente por todo el libro, palabra por palabra, hasta encontrar el término en el que estuviéramos interesados; sería como hacer una búsqueda lineal en un fichero.

Se definen dos tipos de índices: *índices de un solo nivel o uninivel* e *índices multinivel*. Para cada uno de ellos, se definen diversos tipos de índices.

1. **Índices Uninivel**

   a) **Índice Primario**: es un fichero ordenado cuyos registros son de longitud fija y contienen dos campos. El primero contienen el mismo tipo de datos que el campo clave de ordenación (clave primaria) y el segundo es un puntero a un bloque de disco (dirección de bloque).

   Estos índices se pueden dividir a su vez en densos o dispersos. Un *índice total o denso* señala cada entrada a la dirección de un registro del fichero de datos, como se muestra en la figura 1. Un *índice disperso, escaso o no denso* apunta cada entrada a un grupo de registros del fichero de datos que debe estar ordenado, como se muestra en la figura 2.

   b) **Índice Agrupado**: es un fichero ordenado con dos campos. El primero es el mismo tipo que el campo de agrupación del fichero de datos. El segundo es un puntero a un bloque de disco.

   Si los registros de un fichero se encuentran ordenados físicamente según un campo no clave y que no tiene un valor distinto para cada registro, dicho campo se denomina *campo de agrupación*. Es posible crear un tipo diferente de índice llamado *índice de agrupación*, para acelerar la recuperación de registros con el mismo valor en el campo de agrupación, tal como se muestra en la figura 3.

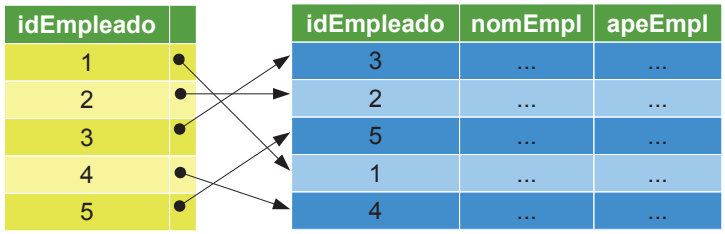

Figura 1: Índice Denso.

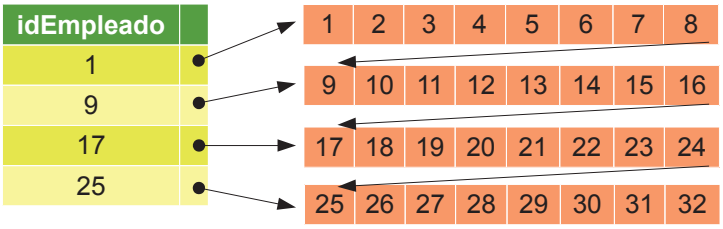

Figura 2: Índice Disperso.

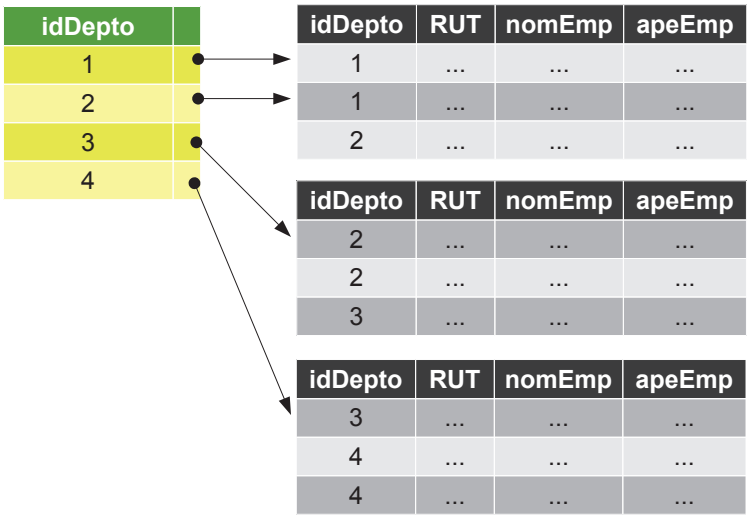

Figura 3: Índice Agrupado.

Una observación a realizar, es que un fichero puede tener ***como máximo un campo de ordenación física****, por lo que puede tener como máximo un índice primario o un índice de agrupación,* ***pero no ambos****.*

c) **Índice Secundario**: es también un fichero ordenado con dos campos. El primer campo es del mismo tipo de datos que el de cualquier campo que no sea el de ordenación del fichero de datos, y se denomina *campo de indexación*. El segundo campo es o bien un puntero a bloque o bien un puntero a registro. Puede haber varios índices secundarios (y por tanto, campos de indexación) para el mismo fichero.

Una estructura de acceso de índice sobre un campo de clave que tiene un valor distinto para cada registro es llamada *clave secundaria*. En este caso hay una entrada de índice para cada registro del fichero de datos, que contiene el valor de la clave secundaria para ese registro y un puntero, ya sea al bloque en el que se esta almacenando es registro o al registro mismo. La figura 4 ejemplifica lo anterior.

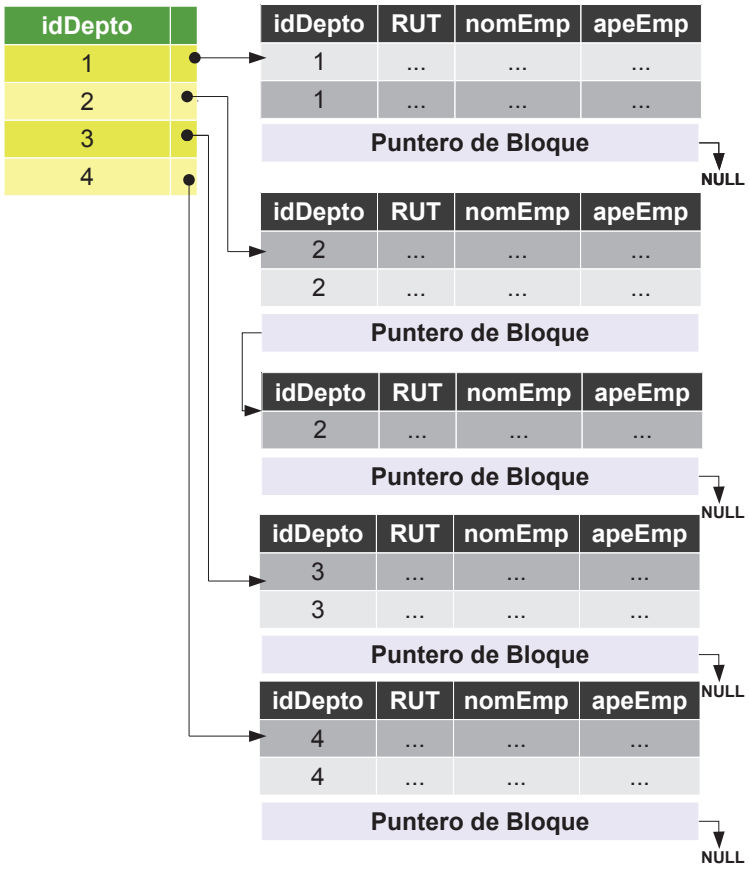

Figura 4: Índice Secundario.

2. Índices Multinivel

El **índice multinivel** considera el fichero del índice, denominado *primer nivel (o nivel base) del índice multinivel*, como un fichero ordenado con un valor distinto para cada registro. A partir de esto es posible crear un índice primario para este nivel, denominado *segundo nivel del índice multinivel*.

Es posible repetir este proceso para el segundo nivel. El *tercer nivel*, que es un índice primario del segundo nivel, tiene una entrada por cada bloque del segundo nivel. Cabe señalar que solo es necesario un segundo nivel si el primero requiere mas de un bloque de almacenamiento en disco, y, de manera similar, solo es necesario un tercer nivel si el segundo requiere mas de un bloque. De igual manera es posible repetir el proceso anterior hasta que todas la entradas de un nivel del índice quepan en un solo bloque, a este bloque se le denomina *índice de nivel superior*.

En resumen la idea de múltiples niveles es muy simple, y se basa en hacer un ciclo iterativo de indexación. De cierta manera los índices multiniveles forman una especie de árbol donde cada nivel es un *índice del nivel inferior*. Si bien existen diversos tipos de índices multinivel, se explicarán solo tres. Estos son los siguientes:

a) **Árbol de búsqueda binaria**: se define como un árbol binario, en la cual cumple con ciertas condiciones:

1) Todo árbol vacío es un árbol binario de búsqueda.
2) Un árbol binario no vacío, de raíz R, es un árbol binario de búsqueda si:
   - En caso de tener subárbol izquierdo, la raíz R debe ser mayor que el valor máximo almacenado en el subárbol izquierdo, y que el subárbol izquierdo sea un árbol binario de búsqueda.
   - En caso de tener subárbol derecho, la raíz R debe ser menor que el valor mínimo almacenado en el subárbol derecho, y que el subárbol derecho sea un árbol binario de búsqueda.

Por ejemplo, la figura 5 representa un árbol binario. En una búsqueda en orden nos entrega lo siguiente: $1 - 3 - 4 - 6 - 7 - 8 - 10 - 13 - 14$. Desafortunadamente este esquema sigue presentando algunos inconvenientes:

- Se siguen realizando muchísimos acceso al disco para poder avanzar por las ramas del árbol.
- La eliminación de algunos registros puede dejar algunos nodos del árbol casi vacíos.
- El principal obstáculo sería que el árbol no quede balanceado, aquí la solución sería emplear árboles AVL.

b) **Árbol B**: se define como una estructura de datos de árbol, en donde los nodos internos almacenan un número variable de nodos, dentro de un

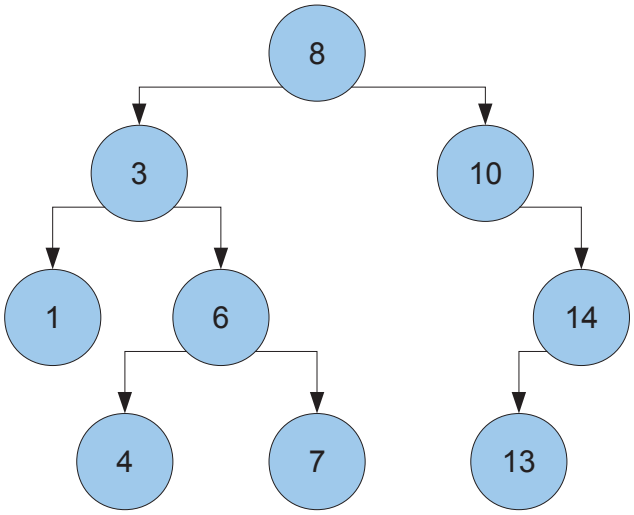

Figura 5: Árbol de Búsqueda Binaria.

rango predefinido. Cuando se inserta o se elimina un dato, la cantidad de nodos hijo varía dentro del nodo. Para que siga manteniéndose el número de nodos dentro del rango predefinido, los nodos internos se juntan o se parten.

Una ventaja del árbol B es que se realiza un número de rebalances bajo, dado que se permite un rango variable de nodos hijo. La desventaja de la estrucutra es que el uso de la memoria aumenta considerablemente, dado que los nodos no permanecen totalmente ocupados. La cantidad máxima y mínima de nodos hijo son definidos para cada implementación en particular; en un árbol B 2-3, por ejemplo, cada nodo sólo puede tener 2 ó 3 nodos hijo. La figura 6 muestra un árbol B.

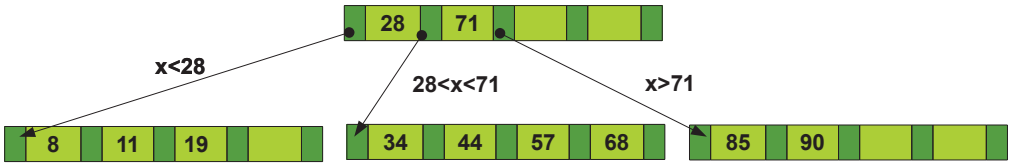

Figura 6: Árbol B.

Un derivado del árbol B es el **Árbol R**, el cual es ocupado para *bases de datos espaciales*.

c) **Hashing**: se define como una estructura de datos de árbol, en donde se almacena una clave y un puntero a un bloque, registro o estructura. Se asimila al concepto de las estructuras uninivel, pero con ciertas diferencias:

- La dirección generada por el árbol Hash suele ser aleatoria, donde no existe una relación aparente entre la clave y la localización del registro correspondiente.
- La función Hash permite que dos claves puedan producir la misma salida o misma dirección, efecto conocido como *colisión*.

Para el manejo de colisiones en Hash existen ciertas soluciones, entre las cuales se encuentran:

- Propagar los registros: Mediante funciones que distribuyan muy aleatoriamente los registros es posible evitar *agrupaciones de claves* que produzcan las mismas direcciones.
- Usar memoria extra: Proponer un espacio de direcciones posibles mayor que el número de registros a usar.
- Colocar más de un registro en una dirección: Este concepto se basa en *buckets* o *cubetas de datos* en cada dirección, donde se colocan casi todos los registros que colisionan de manera que al hacer una búsqueda, sea necesario recuperar la cubeta entera y luego buscar por el registro deseado.

1.2. Sintaxis

1.2.1. MySQL

En MySQL, existen dos operaciones para gestionar índices, las cuales permiten crear y borrar. Estas son **CREATE INDEX** y **DROP INDEX** [4]. Las sintaxis para cada una de las operaciones mencionadas son:

- **CREATE INDEX**:

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
    [index_type]
    ON tbl_name (index_col_name,...)
    [index_option] ...

index_col_name:
    col_name [(length)] [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option:
    KEY_BLOCK_SIZE [=] value -- MySQL >= 5.1
  | index_type -- MySQL >= 5.1
  | WITH PARSER parser_name -- MySQL >= 5.1
  | COMMENT 'string' -- MySQL >= 5.5
```

La operación `CREATE INDEX` permite crear un nuevo índice. Se define un nombre para el índice en el campo `index_name`, y este debe ser único.

El campo `index_type` indica la estructura que será ocupada en la indexación. Por omisión, la estructura que se ocupa para esto es el árbol B; sin embargo, se pueden definir otros, según el tipo de almacenamiento que se encuentre ocupando. La tabla 1 indica las estructuras que se pueden utilizar para cada uno.

Tipo de Almacenamiento	**Índice**
MyISAM	BTREE
InnoDB	BTREE
MEMORY/HEAP	HASH, BTREE
NDB	HASH, BTREE

Tabla 1: Índices a ocupar para cada tipo de almacenamiento.

Los campos `UNIQUE`, `FULLTEXT` o `SPATIAL` indican el tipo de índice que se va a crear.

- Con `UNIQUE` se crea una restricción, en donde todos los valores en el índice tiene que ser distintos (la clave `PRIMARY KEY` es un tipo de índice único). En caso de que se trate de insertar un valor que ya se encuentra, arroja un error. Para ocupar esta restricción, las columnas que se vayan a indexar deben tener la restricción `UNIQUE`.
- `FULLTEXT` puede ser ocupado solo con el tipo de almacenamiento **MyISAM** y con los tipos de datos `CHAR`, `VARCHAR`, y `TEXT`.
- `SPATIAL` puede ser ocupado con los tipo de almacenamiento **MyISAM**,**InnoDB**,**NDB** y **ARCHIVE**, aunque su uso para cada uno de estos puede variar. Se ocupa para los tipos de datos espaciales `POINT` y `GEOMETRY`.

El campo `ON tbl_name` define la tabla que será indexada. Cada (index_col_name) definido indicará las columnas que se ocuparán para generar el índice.

Por ejemplo, la sentencia:

```
CREATE INDEX part_of_name ON customer (name(10));
```

Define un nuevo índice llamado `part_of_name`, el cual ocupará los 10 primeros caracteres de la columna `name` en la tabla `customer`. A partir de allí generará un árbol B con los 10 primeros caracteres de cada fila en la columna invocada.

El campo `KEY_BLOCK_SIZE [=]` indica el tamaño en bytes a ocupara para la generación de los bloques de índice. Esta opción provee una pista al motor de

almacenamiento sobre el tamaño en bytes a usar para los bloques de las claves del índice. Por omisión, o con un valor 0, se toma el valor predefinido para el tipo de almacenamiento.

El campo `WITH PARSER` solo puede ocupado con los índices FULLTEXT. Asocia un tipo de parseo para la generación de índices y las operaciones de consulta. El campo `COMMENT`, ocupado desde la versión 5.5 de MySQL, provee de este campo para realizar comentarios al índice.

Además de *CREATE INDEX*, se puede ocupar la operación `ALTER TABLE` para crear índices. La sintaxis para esto es:

```
ALTER TABLE tbl_name ADD INDEX index_name (columna indexada);
```

- **DROP INDEX**:

```
DROP INDEX index_name ON tbl_name
```

Para eliminar un índice, se ingresa el nombre de este y la tabla en donde es ocupada. También se puede ocupar la operación `ALTER TABLE` para eliminar un índice. La sintaxis para esto es:

```
ALTER TABLE tbl_name DROP INDEX index_name
```

1.2.2. PostgreSQL

En PostgreSQL, existen dos operaciones para gestionar índices, las cuales permiten crear y borrar. Estas son **CREATE INDEX** y **DROP INDEX** [5]. Las sintaxis para cada una de las operaciones mencionadas son:

- **CREATE INDEX**:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ]
    ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ]
    [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

La operación `CREATE INDEX` permite crear un nuevo índice. Se define un nombre para el índice en el campo `name`.

Con el campo `UNIQUE` se crea una restricción, en donde todos los valores en el índice tiene que ser distintos (la clave `PRIMARY KEY` es un tipo de índice único). En caso de que se trate de insertar un valor que ya se encuentra, arroja un error.

Para ocupar esta restricción, las columnas que se vayan a indexar deben tener la restricción `UNIQUE`.

El campo `CONCURRENTLY`, PostgreSQL construirá el índice sin considerar restricciones impuestas sobre operaciones concurrentes de `INSERT`, `UPDATE` o `DELETE`.

El campo `ON table` define la tabla en donde se realizará la indexación. Ocupando el campo `USING method`, definimos la estructura que será ocupara para la creación. PostgreSQL ocupa cuatro tipos de estructuras: *btree*(Árbol B), *hash*, *gist* (*Generalized Search Tree*, para BD espaciales), y *gin* (*Generalized Inverted Index*). Por omisión, se considera *btree* como la estructura de dato a ocupar. Por ejemplo, la expresión:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Genera un índice único con árbol B en la tabla *films* y la columna *title*.

En el campo `column` se indica la lo las columnas que serán indexadas. Se permite ocupar una expresión como columna. Por ejemplo, la expresión:

```
CREATE INDEX ON films ((lower(title)));
```

Generará un índice en donde permitirá realizar búsquedas eficientes, ya que estas serán siempre en minusculas. En este caso no se ha colocado un nombre en el índice, por lo que PostgreSQL generará uno a partir de la instrucción que realiza. En este caso, el nombre a generar será `films_lower_idx`.

El campo `opclass` permite definir ciertas operaciones a realizar con la columna. Los campos `ASC` y `DESC` determinan si el índice a generar con la columna tiene un orden ascendente o descendente. Con `NULLS FIRST` o `NULLS LAST` definen si los elementos nulos serán colocados al principio o al final del resto de los elementos no nulos.

El campo `WITH ( storage_parameter = value [, ... ]` define algunos parámetros del índice. Existe dos tipos de parámetros que se pueden ocupar:

- Con `FILLFACTOR`, ocupados en los índices *B-tree*, *hash* y *GiST*, indica el porcentaje de compresión de las estructuras y toma valores entre 1 y 100. A menor porcentaje, optimiza la eficiencia de la estructura estructura, pero ocupa un mayor espacio en disco, y viceversa. Para tablas estáticas en el tiempo y sin muchas modificaciones se recomienda un valor cercano a 100, mientras que para tablas en constante actualización y uso se recomienda valores bajos. Para *B-tree*, el valor por omisión es 90.
- Con `FASTUPDATE`, ocupado en el índices *Gin*, habilita o deshabilita la técnica de actualización rápida. Es un parámetro boolean: ON habilita la opción, OFF la deshabilita.

El campo `TABLESPACE tablespace` define en donde se almacenará el índice. Por omisión, se ocupa el espacio de `default_tablespace` o `temp_tablespaces`, según si la tabla es temporal o no.

El campo `WHERE predicate` define restricciones para índices parciales.

- **DROP INDEX**:

```
DROP INDEX [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

La operación `DROP INDEX` permite eliminar un índice existente. Con la operación `IF EXISTS` permite omitir la generación de errores en caso de que no exista el índice. El campo `name` indica el nombre del índice a eliminar.

Los operadores `CASCADE` y `RESTRICT` permiten condicionar el borrado. Con el primer operador se elimina los objetos que dependan de la vista. Con el segundo, en caso de que la vista sea ocupado en otros elementos, cancela el borrado. Por omisión se encuentra en `RESTRICT`.

1.3. Consejos sobre indexación

El proceso de creación de índices no es sencillo. Requiere conocer el uso que se le dará a cada una de las tablas y columnas durante el transcurso del tiempo. Si se genera de manera incorrecta un índice puede conllevar a mermar el rendimiento de las consultas. Es por ello, que es recomendable conocer algunos datos importantes y conocer algunas desventajas que pueden surgir con el uso de la indexación.

Recomendaciones

- Crear los índices sobre aquellas columnas que usan una cláusula `WHERE`, y no sobre aquellas columnas que vayan a ser objeto de un simple `SELECT`.
- Pueden resultar mejores candidatas a indexar aquellas columnas que presentan muchos valores distintos, mientras que no son buenas candidatas las que tienen muchos valores idénticos, porque cada consulta implicará siempre recorrer prácticamente la mitad del índice.
- La regla de la izquierda: Si se necesita un `SELECT` del tipo `SELECT ... WHERE columna 1 = X AND columna 2 = Y` y ya se tiene un `INDEX` con la columna 1, es posible crear un segundo índice con la columna 2, o crear un único índice combinado con las columnas 1 y 2. Estos son los índices multicolumna, o compuestos.
- Cuando un índice contiene más de una columna, cada columna es leída por el orden que ocupa de izquierda a derecha, y a efectos prácticos, cada columna (por ese orden) es como si constituyera su propio índice.

- Si se tiene índices multicolumna y se utilizan en las clausulas `WHERE`, se debe incluir siempre de izquierda a derecha las columnas indexadas; de lo contrario el índice NO sera usado.
- No es recomendable usar como índices columnas en las que serán frecuentes operaciones de escritura (`INSERT`, `UPDATE`, `DELETE`).
- No es recomendable crear índices sobre columnas cuando cualquier `SELECT` sobre ellos va a devolver una gran cantidad de resultados; por ejemplo una columna booleana que admita los valores Y/N.
- No es necesario usar índices en tablas demasiado pequeñas, ya que en estos casos no hay ganancia de rapidez frente a una consulta normal.
- Se debe considerar que los índices ocupan espacio, incluso en casos mas espacio que las mismas tablas de datos.

2. Transacciones

2.1. Definición

Una **transacción** es un *programa en ejecución que constituye una unidad lógica del procesamiento de una base de datos*. Una *transacción* incluye *una o más consultas a la base de datos* (consulta de lectura, inserción, eliminación o modificación). Una forma de definir los límites de una transacción es especificando explícitamente las sentencias **begin transaction** y **end transaction**; en este caso, todas las operaciones de acceso a la base de datos que se encuentran entre estas sentencias son consideradas como una transacción. [6].

Un criterio que sirve para detectar la necesidad del uso de transacciones, es el número de usuarios que utilizan el DBMS al mismo tiempo. Un DBMS es **monousuario** si sólo *lo puede utilizar un usuario a la vez*, y es **multiusuario** si *varios usuarios pueden utilizar el sistema* (y, por tanto, acceder a la base de datos) *simultáneamente*. Los **DBMSs monousuario** están *principalmente restringidos a los sistemas de computación personal*; la mayoría *de los demás* **DBMSs son multiusuario**. Por ejemplo, un sistema de reservas en aerolíneas lo utilizan simultáneamente cientos de agentes de viajes. Los sistemas de bancos, aseguradoras, supermercados, etcétera, también operan con muchos usuarios que envían transacciones simultáneamente al sistema [6].

Una transacción mantiene todas las operaciones realizadas en el buffer del DBMS hasta que se indique que esta finaliza. Por ejemplo, se pueden apreciar dos transacciones, T1 y T2, en la figura 7. Existen dos operaciones básicas de acceso a la base de datos que una transacción puede incluir:

1. `read_item(X)`: lee un elemento de base de datos denominado X y lo almacena en una variable de programa. Para simplificar la notación, se asume que la variable de programa también se llama X.

2. `write_item(X)`: escribe el valor de la variable de programa X en un elemento de base de datos denominado X.

Estas funciones indican la lectura de la variable en disco para almacenarlo en la memoria virtual del DBMS y la escritura de este al disco, respectivamente. Una transacción incluye operaciones read_item y write_item para acceder y actualizar la base de datos. El conjunto de lectura (*read-set*) de una transacción es el conjunto de todos los elementos que la transacción lee, y el conjunto de escritura (*write-set*) es el conjunto de todos los elementos que la transacción escribe. Por ejemplo, el conjunto de lectura y escritura de T1 en la figura 7 es X, Y.

Una de las razones para ocupar transacciones, es que las consultas relizadas por varios usuarios pueden ejecutarse concurrentemente y pueden acceder y actualizar los mismos elementos de la base de datos. Si esta ejecución concurrente no está controlada, pueden surgir problemas, los cuales son:

T1
read_item(X);
X = X – N;
write_item(X);
read_item(Y);
Y = Y + N;
write_item(Y);

T2
read_item(X);
X = X + M;
write_item(X);

Figura 7: Ejemplo Transacciones.

1. **Problema por pérdida de actualización**: este problema surge cuando *dos transacciones que acceden a los mismos elementos de la base de datos tienen sus operaciones interpoladas* de un modo que hace que *el valor de algunos elementos de la base de datos sean incorrectos*. Suponga que las transacciones TI y T2 se envían aproximadamente al mismo tiempo, y suponga que sus operaciones están interpoladas como se aprecia en la figura 8; el valor final del elemento X es incorrecto porque T2 lee el valor de X antes de que T1 lo cambie en la base de datos; por tanto, se pierde el valor actualizado resultante de T1.

Tiempo

T1	T2
read_item(X);	
X = X – N;	
	read_item(X);
	X = X + M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y = Y + N;	
write_item(Y);	

Figura 8: Transacciones - Problema por pérdida de actualización.

2. **Problema de la actualización temporal (o lectura sucia)**: este problema ocurre cuando *una transacción actualiza un elemento de la base de datos y, después, falla.* El elemento actualizado es accedido por otra transacción antes de cambiar a su valor original. La figura 9 muestra un ejemplo donde T1 actualiza el elemento X y después falla antes de concluir, por lo que el sistema debe devolver

X a su valor original. Sin embargo, antes de poder hacerlo, la transacción T2 lee el valor temporal de X, que no se grabará permanentemente en la base de datos debido al fallo de T1. El valor del elemento X que es leído por T2 se denomina **dato sucio** porque lo ha creado una transacción que todavía no se ha completado y confirmado.

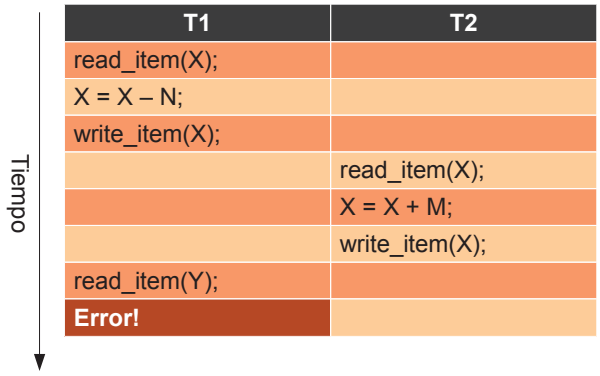

Figura 9: Transacciones - Problema de la actualización temporal.

3. **Problema de la suma incorrecta**: si una transacción está calculando una función de suma agregada sobre varios registros, mientras otras transacciones están actualizando algunos de esos registros, *la función agregada puede calcular algunos valores antes de que sean actualizados y otros después de ser actualizados*. Por ejemplo, suponga que una transacción T3 está calculando el número total de reservas en todos los vuelos; mientras tanto, se está ejecutando la transacción T1. Si se produce la interpolación de operaciones de la figura 10, el resultado de T3 estará desfasado una cantidad N porque T3 lee el valor de X después de que se hayan sustraído N plazas de ella, pero lee el valor de Y antes de que esas N plazas se hayan añadido a él.

4. **Problema de la lectura irrepetiblea**: este problema ocurre cuando una transacción T lee un elemento dos veces y el elemento es modificado por otra transacción T entre las dos lecturas. Por tanto, T recibe *valores diferentes* en sus dos lecturas del mismo elemento. Por ejemplo, si durante una transacción de reserva de un bus, un cliente busca información sobre la disponibilidad de asientos en varias salidas. Cuando el cliente opta por una salida en particular, la transacción lee el número de asientos de esa salida una segunda vez antes de completar la reserva.

Otra de las razones para ocupar transacciones, es la capacidad de recuperar el sistema ante fallos. Siempre que se envía una transacción a un DBMS para su ejecución, *el sistema es responsable de garantizar que todas las operaciones de la transacción se completen satisfactoriamente* y que *su efecto se grabe permanentemente en la base de*

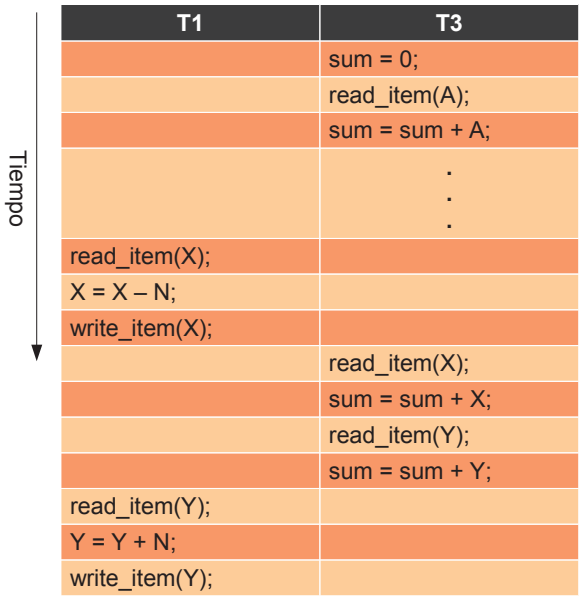

Figura 10: Transacciones - Problema de la suma incorrecta.

datos, o de que *la transacción no afecte a la base de datos o a cualquier otra transacción*. El DBMS **no debe permitir que algunas operaciones de una transacción T se apliquen a la base de datos mientras otras no**. Esto puede ocurrir si una transacción falla después de ejecutar algunas de sus operaciones, pero antes de ejecutar todas ellas.

2.2. Sintaxis

En general, una transacción se compone de las siguientes operaciones:

- **BEGIN_TRANSACTION**: marca el inicio de la ejecución de una transacción.
- **READ o WRITE**: especifican operaciones de lectura o escritura en los elementos de la base de datos que se ejecutan como parte de una transacción. Estas operaciones vienen representadas por las consultas SQL.
- **END_TRANSACTION**: especifica que las operaciones READ y WRITE de la transacción han terminado y marca el final de la ejecución de la transacción. Sin embargo, en este punto puede ser necesario comprobar si los cambios introducidos por la transacción pueden aplicarse de forma permanente a la base de datos (confirmados) o si la transacción se ha cancelado por alguna situación o estado inesperado.
- **COMMIT_TRANSACTION**: señala una *finalización satisfactoria* de la

transacción, por lo que los cambios (actualizaciones) ejecutados por la transacción se pueden **enviar** con seguridad a la base de datos y no se desharán.

- **ROLLBACK (o ABORT)**: señala que la transacción *no ha terminado satisfactoriamente*, por lo que deben **deshacerse** los cambios o efectos que la transacción pudiera haber aplicado a la base de datos.

2.2.1. MySQL

En MySQL, de manera predeterminada las consultas vienen con una opción llamada `AUTO_COMMIT`. Esta variable indica que ante cada consulta realizada, se realiza un `COMMIT`. Para deshabilitar esta opción, se debe escribir:

```
-- Con autocommit = 1 se vuelve a la configuración normal
SET AUTOCOMMIT=0;
```

De esta forma, cuando se quiera confirmar una consulta se debe escribir COMMIT, o ROLLBACK si se desea deshacer. El cambio debe ser realizado por un usuario con privilegios de administración.

Otra forma que se puede iniciar una consulta es a través de **START TRANSACTION**. Esta operación permite realizar consultas en modo transaccional hasta confirmar con un `COMMIT` o `ROLLBACK`. Una alternativa a **START TRANSACTION** es **BEGIN**:

```
START TRANSACTION;
-- Consultas SQL
COMMIT;
```

Estas operaciones son realizadas si el motor de almacenamiento de la tabla es **InnoDB**.

- **SET TRANSACTION**:

  ```
  SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
  { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
  ```

  La operación `SET TRANSACTION` permite preparar el nivel de aislamiento de transacción para la siguiente transacción, globalmente, o para la sesión actual. El nivel de aislamiento predeterminado es `REPEATABLE-READ`. El uso del término `SERIALlZABLE` aquí está basado en no permitir que ocurran ciertos problemas. Si una transacción se ejecuta a un nivel de aislamiento más bajo que `SERIALlZABLE`, entonces se pueden producir una o más de estas violaciones:

  - **Lectura sucia**: una transacción T1 puede leer la actualización de una transacción T2 que todavía no se ha confirmado. Si T2 falla y es cancelada, entonces T1 habría leído un valor que no existe y es incorrecto.

- **Lectura irrepetible**: Una transacción T1 puede leer un valor dado de una tabla. Si otra transacción T2 actualiza más tarde ese valor y T1 lee de nuevo el valor, T1 verá un valor diferente.
- **Fantasmas**: Una transacción T1 puede leer un conjunto de filas de una tabla, quizá basándose en alguna condición especificada en la cláusula `WHERE` de SQL. Ahora, suponga que una transacción T2 inserta una fila nueva que también satisface la condición de la cláusula `WHERE` utilizada en T1 en la tabla utilizada por T1. Si T1 se repite, entonces T1 verá un fantasma, una fila que anteriormente no existía.

La figura 11 resume los tipos de problemas que pueden surgir con cada tipo de transacción:

	Tipo de Problemas		
Nivel de Aislamiento	Lectura Sucia	Lectura Irrepetible	Fantasma
READ UNCOMMITED	**SI**	**SI**	**SI**
READ COMMITED	**NO**	**SI**	**SI**
REPEATABLE READ	**NO**	**NO**	**SI**
SERIALIZABLE	**NO**	**NO**	**NO**

Figura 11: Transacciones - Ocurrencia de errores ante cada tipo de transacción.

2.3. Consejos sobre transacciones

La utilización de transacciones debe considerarse como algo a aplicar en ambientes de sistemas críticos y operaciones en la web. Esto se debe, a que deshacer operaciones incompletas o recuperar el sistema ante una eventualidad inespererada, bajo este metodo resulta ser más eficiente que realizar una recuperación o un retroceso de forma manual. Es por ello, que el tópico debe ser una regla de oro al momento de administrar sistemas.

2.3.1. PostgreSQL

En PostgreSQL, cada acceso comienza a través de **BEGIN**. Esta operación permite realizar cualquier consulta en modo transaccional hasta confirmar con un `COMMIT` o `ROLLBACK`:

```
BEGIN;
-- Consultas SQL
COMMIT;
```

Alternativamente, se pueden definir puntos de guardado, o `SAVEPOINT`. Esto permite, en caso de realizar un `ROLLBACK`, retroceder hasta el punto de guardado, logrando que las operaciones realizadas antes del estado se encuentren aún en memoria.

```
BEGIN;
-- Consultas SQL
SAVEPOINT my_savepoint;
-- Consultas SQL
COMMIT;
```

Todas las transacciones realizadas en este DBMS se encuentran en modo SERIALIZABLE.

2.4. ACID

Uno de los puntos que se deben considerar, al momento de realizar transacciones, es asegurar que estas funciones como se esperan. Para ello, estas deben cumplir con ciertas exigencias minimas para que sean consideradas como optimas. Para ello, se define una regla de oro, denominada la prueba de ácido o **ACID** [2].

ACID es un conjunto de características necesarias para que una serie de instrucciones seab consideradas como una transacción. En concreto ACID es un acrónimo de Atomicity (Atomicidad), Consistency (Consistencia), Isolation (Aislamiento) and Durability (y Durabilidad).

- **Atomicity**: La atomicidad de una transacción garantiza que todas sus acciones sean realizadas o ninguna sea ejecutada , en el caso de la transacción bancaria o se ejecuta tanto el "deposito-deducción" o ninguna acción será realizada.
- **Consistency**: Muy similar a la "Atomicidad", la consistencia garantiza que las reglas que hayan sido declaradas para una transacción sean cumplidas, regresando a la transacción bancaria, supongamos que cada vez que se realice una transferencia inter-bancaria de $100,000 sea necesario notificar a la sucursal del tarjeta-habiente, si no es posible comunicarse y actualizar la información en la sucursal del cliente, toda la transacción será abortada.
- **Isolation**: Esto garantiza que las transacciones que se estén realizando en el sistema sean invisibles a todos los usuarios hasta que estas hayan sido declaradas finales, en la transacción bancaria es posible que el sistema este programado para intentar en 5 o 10 ocasiones más antes de abortar una transacción por completo, a pesar que este ultimo paso no ha sido finalizado ya existen otras modificaciones en el sistema, este aislamiento "Isolation" garantiza que los usuarios del sistema no observen estos cambios intermedios hasta que sea finalizada la ultima acción de actualización.

- **Durability**: La durabilidad de una transacción garantiza que al instante en el que se finaliza la transacción esta perdure a pesar de otras consecuencias, esto es, si el disco duro falla, el sistema aún será capaz de recordar todas la transacciones que han sido realizadas en el sistema.

3. Bases de Datos Remotas y Distribuídas

3.1. Definición

Durante la década de los 70, las *bases de datos* seguían la tendencia de usar sistemas centralizados; en los 80, fue la contraria: más descentralización y autonomía de procesamiento. Con los avances en la computación y el procesamiento distribuido de los sistemas operativos, la comunidad de investigación de las bases de datos trabajó sin descanso para resolver los problemas derivados de la distribución de datos. Sin embargo, un *DBMS distribuído (DDBMS)* totalmente operativo que incluyera la funcionalidad y técnicas propuestas por la investigación de bases de datos distribuídas nunca emergió como un producto comercialmente viable. La mayoría de fabricantes no centró sus esfuerzos en el desarrollo de un producto DDBMS puro, sino en la generación de sistemas basados en cliente-servidor, o a través de tecnologías para el acceso a fuentes de datos distribuidos de manera heterogénea.

Sin embargo, las organizaciones han estado muy interesadas en la descentralización del procesamiento (a nivel de sistema) a la vez que conseguían una integración de las fuentes de información (a nivel lógico) dentro de sus usuarios, aplicaciones y sistemas de bases de datos distribuidos geográficamente. Asociada a los avances en las comunicaciones, ahora existe una aceptación general del esquema cliente-servidor en el desarrollo de una aplicación, el cual asume muchos de los temas de bases de datos distribuídas.

Un **sistema de computación distribuido** consiste en un número de elementos de procesamiento, no necesariamente homogéneos, que están interconectados mediante una red de computadores, y que cooperan para la realización de ciertas tareas asignadas. Como objetivo general, estos sistemas dividen un gran e inmanejable problema en piezas más pequeñas para resolverlo de una manera coordinada.

Se puede definir una **DDB (Base de datos distribuida)** como una colección de múltiples bases de datos distribuidas interrelacionadas de forma lógica sobre una red de computadores, como se muestra en la figura 12; un **DDBMS (Sistema de administración de bases de datos distribuidas)** como el software encargado de administrar la base de datos distribuida mientras hace la distribución transparente para el usuario, y una **Base de Datos Remota** como el acceso, mediante un computador cliente, a un servidor remoto de base de datos mediante la interfaz del DBMS, como se muestra en la figura 13. [7]

Figura 12: Red distribuida de bases de datos.

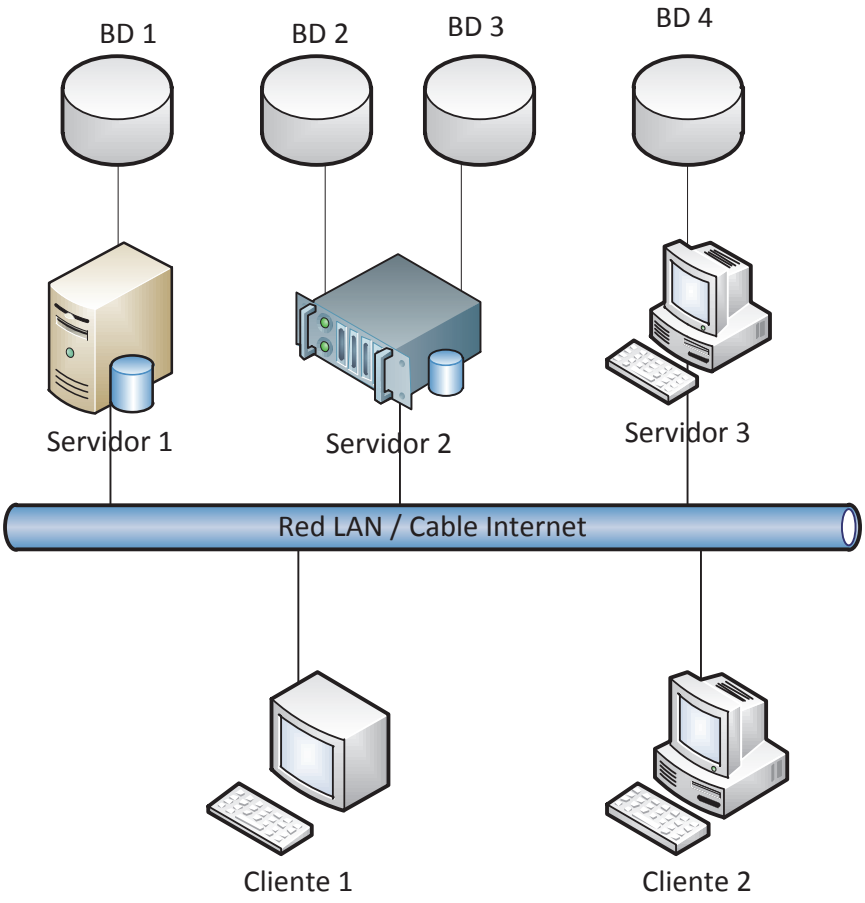

Figura 13: Acceso remoto a diversas bases de datos.

3.1.1. Ventajas

La gestión de bases de datos distribuidas ha sido propuesta por varias razones, entre las cuales se encuentran:

1. **Administración de datos distribuidos con distintos niveles de transparencia**: un DBMS, idealmente, debe ser una distribución transparente en el sentido de ocultar los detalles de dónde está físicamente ubicado cada fichero (tabla, relación) dentro del sistema. En este caso, son posibles los siguientes tipos de transparencias:

   - **Transparencia de red o de distribución:** se refiere a la autonomía del usuario de los detalles operacionales de la red. Puede dividirse en transparencia de localización y de denominación. Por una parte, la *transparencia de localización* hace mención al hecho de que el comando usado para llevar a cabo una tarea es independiente de la ubicación de los datos y del sistema desde el que se ejecutó dicho comando. La *transparencia de denominación* implica que, una vez especificado un nombre, puede accederse a los objetos nombrados sin ambigüedad y sin necesidad de ninguna especificación adicional.
   - **Transparencia de replicación:** permite que el usuario no se entere de la existencia de copias. Se pueden almacenar copias de los datos en distintos lugares para disponer de una mayor disponibilidad, rendimiento y fiabilidad.
   - **Transparencia de fragmentación:** Una consulta global del usuario debe ser transformada en varias consultas fragmentadas. La transparencia de fragmentación permite que el usuario no se entere de la existencia de fragmentos. Existen dos posibles tipos de fragmentación: la horizontal y la vertical. La primera distribuye una relación en conjuntos de tuplas, o filas. La segunda lo hace en subrelaciones, de modo que cada subrelación está definida por un subconjunto de las columnas de la relación original.

2. **Incremento de la fiabilidad y la disponibilidad:**. Éstas son dos de las más importantes ventajas de las bases de datos distribuidas. La **fiabilidad** está definida ampliamente como la probabilidad de que un sistema esté funcionando (no caído) en un momento de tiempo, mientras que la **disponibilidad** es la probabilidad de que el sistema esté continuamente disponible durante un intervalo de tiempo. Cuando los datos y el software DBMS están distribuidos a lo largo de distintas localizaciones, uno de ellos puede fallar, mientras el resto continúa operativo.

   Esto mejora tanto la fiabilidad como la disponibilidad, ya que se logra una optimización al replicar tanto los datos como el software en más de una ubicación. En un sistema centralizado, el fallo de una ubicación provoca la caída del sistema para todos. los usuarios. En una base de datos distribuida, en cambio,

parte de la información puede estar inaccesible, pero sí se podrá acceder a otras partes de la base de datos.

3. **Rendimiento mejorado**: Un DBMS distribuido fragmenta la base de datos manteniendo la información lo más cerca posible del punto donde es más necesaria. La localización de datos reduce el enfrentamiento por la CPU y los servicios de E/S, a la vez que atenúa los retardos en el acceso implícito a las redes de área extendida. Cuando se distribuye una base de datos a lo largo de varias localizaciones, lo que obtenemos son bases de datos más pequeñas. Como resultado, se obtiene lo siguiente:

   - Las consultas locales y las transacciones de acceso a los datos de uno de estos sitios tienen un mayor rendimiento debido al menor tamaño de esas bases de datos.
   - Cada sitio tiene que ejecutar un menor número de transacciones que si todas ellas fueran llevadas a cabo por una base de datos centralizada.
   - El paralelismo interconsultas e intraconsultas puede conseguirse ejecutando múltiples consultas de diferentes sitios, o dividiendo esa consulta en otras más pequeñas que se ejecutan en paralelo.

4. **Expansión más sencilla**: Se simplifica la expansión del sistema en términos de incorporación de más datos, incremento del tamaño de las bases de datos o la adición de más procesadores.

3.2. Configuraciones de bases de datos remotas

Acceder a una base de datos de manera remota, de manera preestablecida, se encuentra bloqueado. Es por que, para habilitar el el acceso de manera externa, se deben realizar ciertas configuraciones al DBMS. Acá se presentan los pasos para habilitar el acceso remoto a una base de datos.

3.2.1. MySQL

En MySQL, se debe realizar el siguiente procedimiento:

- Para **Linux**:

  1. Primero se debe configurar el archivo my.cnf. Este provee las configuraciones del DBMS. La ruta donde se puede encontrar el archivo son las siguientes:
     - Para **Debian Linux**: `/etc/mysql/my.cnf`.
     - Para **Red Hat Linux/Fedora/Centos Linux**: `/etc/my.cnf`.
     - Para **FreeBSD**: se necesita crear el archivo en `/var/db/mysql/my.cnf`.
  2. Luego, se debe localizar la siguiente linea: `[mysqld]`. En esta parte, se configura el acceso del servidor.
  3. En esta parte, hay que localizar la linea `skip-networking` y se debe reemplazar por `# skip-networking`.
  4. Debajo de esta linea, se debe añadir lo siguiente: `bind-address=YOUR-SERVER-IP`, donde `YOUR-SERVER-IP` es la IP del servidor.

     Por ejemplo, si el cliente tiene la IP 65.55.55.2, debe quedar de la siguiente manera:

     ```
     [mysqld]
     ...
     bind-address    = 65.55.55.2
     # skip-networking
     ....
     ```

     Si se desea dejar que mysql acepte todas las conexiones, debe reemplazar por 0.0.0.0.

     ```
     [mysqld]
     ...
     bind-address    = 0.0.0.0
     # skip-networking
     ....
     ```

5. Una vez hecho lo anterior, se graba la configuración realizada.
6. Luego, hay que reiniciar el servidor. Para ello, se ingresa a la terminal y se ingresa lo siguiente: `/etc/init.d/mysql restart`
7. Una vez hecho esto, hay que otorgar privilegios al usuario. Para ello hay que ingresar a la consola de MySQL.
8. Luego, se debe otorgar el permiso de usuario a la base de datos. Para ello, se escribe en la consola lo siguiente:

   ```
   GRANT <PRIVILEGIOS> ON <DB>.<TAB> TO <USER>@<HOST>
   IDENTIFIED BY <PASS>;
   ```

   - `GRANT <PRIVILEGIOS>` otorga las autorizaciones a realizar por el usuario. Para otorgar todos los permisos al usuario, se debe escribir `GRANT ALL`
   - `ON <DB>.<TAB>` indica a que tabla y base de datos tendrá autorización el usuario. Para otorgar permiso a todas las bases de datos o tablas se debe escribir *. Por ejemplo: `db.*` otorga permiso a todas las tablas en la base de datos *db*, mientras que `*.*` otorga permiso a todas las bases de datos y todas las tablas.
   - `TO <USER>@<HOST>IDENTIFIED BY <PASS>` indica el usuario que tendrá autorización a conectarse remotamente. `<USER>` indica el nombre del usuario. `<HOST>` indica la IP del cliente remoto. `<PASS>` es la contraseña del usuario.
9. Una vez hecho lo anterior, se debe salir de la consola MySQL.
10. Para finalizar, se debe habilitar el puerto de MySQL. De forma predeterminada, se ocupa el puerto 3306. Para ello, se abre una nueva terminal y se ingresa el siguiente comando:

    ```
    /sbin/iptables -A INPUT -i eth0 -p tcp --destination-port 3306
    -j ACCEPT
    ```

    .
11. Finalmente, se reinicia el servicio, mediante el comando: `service iptables save`.
12. Para ingresar de manera remota a una base de datos, se escribe el comando `mysql -u <USER>-h <IP>-p`, donde `<USER>` es el usuario remoto de la base de datos e `<IP>` es la dirección IP del servidor remoto.

- Para **Windows**:

1. En caso de estar ocupando el Firewall de Windows, se debe habilitar su acceso. Para ello, se debe ir a *Panel de Control* y luego abrir `Firewall de Windows`.
2. Luego, se debe ir al manejo de excepciones. Para ello, se debe ir a la pestaña *Excepciones*.

3. Para agregar una nueva excepción, se presiona el botón `Agregar Puerto`. Se debe ingresar el nombre de la aplicación (MySQL por ejemplo), luego el puerto a habiliar (de manera predeterminada se ocupa el puerto 3306) y luego el protocolo, en donde se debe seleccionar TCP. item Una vez hecho esto, hay que otorgar privilegios al usuario. Para ello hay que ingresar a la consola de MySQL.
4. Luego, se debe otorgar el permiso de usuario a la base de datos. Para ello, se escribe en la consola lo siguiente:

   ```
   GRANT <PRIVILEGIOS> ON <DB>.<TAB> TO <USER>@<HOST>
   IDENTIFIED BY <PASS>;
   ```

   - `GRANT <PRIVILEGIOS>` otorga las autorizaciones a realizar por el usuario. Para otorgar todos los permisos al usuario, se debe escribir `GRANT ALL`
   - `ON <DB>.<TAB>` indica a que tabla y base de datos tendrá autorización el usuario. Para otorgar permiso a todas las bases de datos o tablas se debe escribir *. Por ejemplo: `db.*` otorga permiso a todas las tablas en la base de datos *db*, mientras que `*.*` otorga permiso a todas las bases de datos y todas las tablas.
   - `TO <USER>@<HOST>IDENTIFIED BY <PASS>` indica el usuario que tendrá autorización a conectarse remotamente. `<USER>` indica el nombre del usuario. `<HOST>` indica la IP del cliente remoto. `<PASS>` es la contraseña del usuario.
5. Una vez hecho lo anterior, se debe salir de la consola MySQL.
6. Para ingresar de manera remota a una base de datos, se escribe el comando `mysql -u <USER>-h <IP>-p`, donde `<USER>` es el usuario remoto de la base de datos e `<IP>` es la dirección IP del servidor remoto.

3.2.2. PostgreSQL

En PostgreSQL, se debe realizar el siguiente procedimiento:

1. Primero se debe abrir el archivo `postgresql.conf`:

   - En **Windows** se ubica en `C:\Archivos de Programa\PostgreSQL\<VERSION>\data`, donde `<VERSION>` es la versión de PostgreSQL que se encuentra instalado (por ejemplo 9.0).
   - En **Linux** se ubica en `/etc/postgresql/<VERSION>/main`, donde `<VERSION>` es la versión de PostgreSQL que se encuentra instalado (por ejemplo 9.0).

2. Luego se debe reemplazar la linea `#listen_addresses = 'localhost'` por `listen_addresses = '*'`. Con esta linea, indicamos que de ahora en adelante se escuchará todas las solicitudes externas. Guarde la modificación realizada.

3. Ahora se debe abrir el archivo `pg_hba.conf`, el cual se encuentra ubicado en la misma carpeta.

4. Al final del archivo, se encuentra la configuración de los equipos que se encuentran autorizados a utilizar postgres. En este caso, la configuración sigue el siguiente patrón: `TYPE DATABASE USER CIDR-ADDRESS METHOD`

   - `TYPE` identifica el tipo de conexión. La variable `local` indica que la conexión es del mismo equipo, `host` es una conexión TCP/IP (ya sea con o sin SSL), `hostssl` es una conexión TCP/IP con SSL y `hostnossl` es una conexión TCP/IP sin SSL.
   - `DATABASE` indica la base de datos que se puede acceder. Para autorizar a todas las bases de datos se ocupa `ALL`.
   - `USER` indica el usuario que puede acceder a PostgreSQL. Para autorizar a todos los usuarios se ocupa `ALL`.
   - `USER` indica el o los equipos que pueden acceder a PostgreSQL. La dirección se describe como la dirección IP y el CIDR, el cual es un número entre 0 y 32 (IPv4) o 0 y 128 (IPv6) e indica el rango de IP's autorizados. Por ejemplo: 172.20.143.89/32 indica una IP de un computador, mientras que 172.20.143.0/24 indica una red de computadores (que se encuentran autorizados desde la IP 172.20.143.1 a la IP 172.20.143.255). Para autorizar a todas las direcciones se ocupa `0.0.0.0/0`.
   - `METHOD` indica el método a utilizar para la autentificación de los usuarios. Ocupando el valor `trust` autorizamos al usuario sin ingresar una contraseña. Con *password* el usuario ingresa una contraseña no encriptada, mientras que con *md5* indicamos que ingrese una contraseña encriptada. Para usos en redes externas, el *md5* resulta ser el más seguro.

   Por ejemplo: `host all all 0.0.0.0/0 trust` autoriza a todos los usuarios de todas las direcciones a conectarse a cualquier base de datos sin autentificarse. Es el método más inseguro, por lo que solo se debe usar en redes pequeñas y solo para propósitos de prueba.

5. Una vez realizada las modificaciones correspondientes, se guarda el archivo.

6. Finalmente, se debe reiniciar el servicio de PostgreSQL para aplicar los cambios realizados.

   - En **Windows** se debe ir a Inicio >Ejecutar (Tecla Win + R) y escribir `C:\WINDOWS\system32\cscript.exe //NoLogo "C:\Archivos de programa\PostgreSQL\<VERSION>\scripts\serverctl.vbs" restart wait`, donde `<VERSION>` es la versión de PostgreSQL que se encuentra instalado (por ejemplo 9.0).
   - En **Linux** se debe abrir la terminal y escribir `/etc/init.d/postgresql restart`.

3.3. Configuraciones de bases de datos distribuídas

3.3.1. MySQL

A contar de la versión 5.0, MySQL provee un tipo de almacenamiento denominado `FEDERATED`. Este motor provee al DBA la posibilidad de crear punteros a tablas que se encuentran en otros servidores MySQL, enlazando datos separados, o *islas de datos*, para formar una o más bases de datos lógicas. Este sistema es similar al ocupado en bases de datos empresariales, como *Oracle* o *SQL Server*. [3]

Para que se pueda ocupar el motor FEDERATED, previamente hay que realizar una pequeña modificación en el archivo **my.cfg**. Dentro del archivo, se debe buscar la siguiente linea:

```
[mysqld]
```

Y debajo de esta, agregar la instrucción **federated**. Así:

```
[mysqld]
federated
```

Para utilizar este tipo de almacenamiento en MySQL, tiene que ser definido desde el momento en que se crea una tabla. La sintaxis a ocupar es el siguiente:

```
CREATE TABLE tbl_name ([(create_definition,...)])
  ENGINE=FEDERATED
  CONNECTION='scheme://user_name[:password]@host[:port]/db_name/tbl_name';
```

Los parámetros que se agregan a la operación son el `ENGINE` y `CONNECTION`. En `ENGINE=FEDERATED` indicamos que el tipo de almacenamiento a ocupar es una base de datos federada. En `CONNECTION` se definen los parámetros para conectar a los datos que se encuentran en el servidor remoto. Los parámetros que se configuran son los siguientes:

- `scheme`: indica el driver a ocupar para la conexión. En este caso, el valor es `mysql`.
- `user_name`: indica el nombre de usuario a ocupar para la conexión.
- `password`: indica la contraseña de la cuenta, en caso de tener una.
- `host`: indica la dirección IP donde se conectará remotamente a MySQL.
- `port`: indica el puerto para conectarse a MySQL. Por omisión, se considera el puerto 3306.
- `db_name`: indica el nombre de la base de datos a ocupar para rescatar los datos.
- `tbl_name`: indica el nombre de la tabla a ocupar para rescatar los datos.

De esta manera, los datos que se encuentran en la base de datos serán referenciadas de la que se encuentra remotamente. En caso de que se tenga replicada una tabla en múltiples base de datos, se puede ocupar las vistas para concatenar los resultados. Así para el usuario será transparente la distribución de datos.

Por ejemplo, si se realiza la siguiente operación:

```
mysql> use gim2
Database changed
mysql> CREATE TABLE client_transaction_history (
         client_transaction_id int(11) NOT NULL default '0',
         client_id int(11) NOT NULL default '0',
         investment_id int(11) NOT NULL default '0',
         action varchar(10) NOT NULL default '',
         price decimal(12,2) NOT NULL default '0.00',
         number_of_units int(11) NOT NULL default '0',
         transaction_status varchar(10) NOT NULL default '',
         description varchar(200) default NULL,
         broker_id bigint(10) default NULL,
         broker_commission decimal(10,2) default NULL
       )
       ENGINE=FEDERATED
       DEFAULT CHARSET=latin1
       CONNECTION='mysql://robin:mypassword@192.168.32.69:3306/
       gim2/client_transaction_history';
```

En esta operación, se obtiene el puntero a la tabla `client_transaction_history`, de la base de datos remota gim2, ubicada en 192.168.32.69. Esta conexión remota a la tabla se hará con el usuario `robin` y contraseña `mypassword`.

Ahora, se tiene las tablas `client_transaction` y `client_transaction_history`, ambas tablas son iguales, pero con datos diferentes. Si se desea unir ambas tablas en una, se puede crea una vista para concatenar los resultados. La siguiente operación, realiza lo anterior:

```
mysql> CREATE VIEW
         client_transaction_all
       AS
       SELECT
         client_transaction_id,
         client_id,
         investment_id,
         action,
         price,
         number_of_units,
```

```
        transaction_status,
        transaction_sub_timestamp,
        transaction_comp_timestamp,
        description,
        broker_id,
        broker_commission
      FROM
        client_transaction
      UNION ALL
      SELECT
        client_transaction_id,
        client_id,
        investment_id,
        action,
        price,
        number_of_units,
        transaction_status,
        transaction_sub_timestamp,
        transaction_comp_timestamp,
        description,
        broker_id,
        broker_commission
      FROM
        client_transaction_history;
Query OK, 0 rows affected (0.00 sec)
```

De esta manera, al llamar a `client_transaction_all`, realizará la operación tanto en las tablas `client_transaction` como en `client_transaction_history`.

Bibliografía

[1] Juan Pablo Isasmendi Díaz. *Taller de Bases de Datos Avanzadas*, Trabajo de Título de Ingeniería en Informática Aplicada , Departamento de Computación, Universidad de Valparaíso, 2009.

[2] Osmosis Latinas. *Bases de Datos*. `http://www.osmosislatina.com/aplicaciones/bases_de_datos.htm`. Último acceso: 31 de Mayo del 2011.

[3] MySQL. *Bases de Datos Federadas*. `http://dev.mysql.com/tech-resources/articles/mysql-federated-storage.html`. Último acceso: 31 de Mayo del 2011.

[4] MySQL. *Indices*. `http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html`. Último acceso: 30 de Mayo del 2011.

[5] PostgreSQL. *Indices*. `http://www.postgresql.org/docs/9.0/static/indexes.html`. Último acceso: 30 de Mayo del 2011.

[6] Shamkant B. Navathe Ramez Elmasri. *Fundamentos de Sistemas de Bases de Datos*, chapter 17, pages 517–540. Pearson Educación S.A., 5th. edition, 2007.

[7] Shamkant B. Navathe Ramez Elmasri. *Fundamentos de Sistemas de Bases de Datos*, chapter 23, pages 749–775. Pearson Educación S.A., 5th. edition, 2007.