SERVIDOR DE CONSULTAS TEMPORALES Y DE FUTUROS ALTERNATIVOS

por

Nora Jacobawsky, Diego Pastoriza, Claudio Pini

LICENCIATURA EN CS. DE LA COMPUTACIÓN
UNIVERSIDAD DE BUENOS AIRES

Director: Alejandro Vaisman

2000

SERVIDOR DE CONSULTAS TEMPORALES Y DE FUTUROS ALTERNATIVOS

POR NORA JACOBAWSKY, DIEGO PASTORIZA Y CLAUDIO PINI

UNIVERSIDAD DE BUENOS AIRES

ABSTRACT

Una Base de Datos Temporal permite el almacenamiento y registro del tiempo de validez de los datos en el mundo real. El manejo de futuros alternativos facilita la toma de decisiones simulando estados posibles de una Base de Datos basados en la ocurrencia hipotética de uno o más eventos definidos a partir de la realidad a modelar.

En este trabajo se presenta un modelo de Base de Datos que permite el manejo de información pasada, presente y futura. El modelo incluye un lenguaje basado en el standard de facto TSQL2 que incorpora características como el manejo de información hipotética futura. Se implementó un servidor de sentencias temporales que soporta dicho lenguaje, incluyendo la posibilidad de embeber sentencias de TSQL2 extendido en programas construidos en lenguajes de tercera generación.

CAPÍTULO 1

Las aplicaciones de hoy en día hacen uso constante de datos variantes en el tiempo. Las Bases de Datos disponibles en el mercado no realizan el manejo automático de la dimensión temporal, obligando a los desarrolladores a implementar soluciones ad-hoc en cada nueva aplicación que se deba construir.

La naturaleza temporal de la información puede ser manejada directamente por la Base de Datos, pero no existe en la actualidad un RDBMS en el mercado que ofrezca esta posibilidad y solo existen algunas implementaciones académicas con diferentes características, enfoques y alcances [B95]. En este trabajo, presentamos un Sistema de Gestión de Bases de Datos (SGBD) con soporte temporal pasado, presente y futuro, junto con un lenguaje compatible con el standard de facto TSQL2.

Una Base de Datos Temporal añade la dimensión temporal a los datos almacenados. La mayoría de las implementaciones existentes solo toman en cuenta el tiempo pasado manteniendo la historia de las modificaciones realizadas a una tupla. Existe un gran rango de casos prácticos en donde es de gran utilidad la posibilidad de manejar diferentes "estados alternativos" de la Base de Datos como lo son típicamente las aplicaciones utilizadas para la planificación. Un sistema SGBD soportando las características mencionadas a través del modelo *Branching Time* fue introducido por N.L. Sarda [SA97] permitiendo la creación de "futuros alternativos" dentro de la Base de Datos. Estas diferentes alternativas dependen de la ocurrencia de eventos que modelan estados posibles en el mundo real. Así, es posible realizar consultas del tipo "*what if...?*". La Base de Datos además provee los mecanismos necesarios para señalar que un evento ha ocurrido en el mundo real y este nuevo estado debe ser reflejado por los datos almacenados como así también la eliminación de ramas que ya no pueden ser alcanzadas. El enfoque original presentado en [SA97] presenta algunas limitaciones que abordaremos más adelante y no existen reportes acerca de los resultados prácticos de su trabajo.

A modo de ejemplo, supongamos que tenemos dos relaciones **EMPLEADOS(EmpId,**

EmpNombre, LocId) y **LOCALES(LocId, LocNombre)**. Una forma posible y muy comunmente utilizada de mantener el tiempo durante el cual un empleado pertenece a un local, sería incorporar dos marcas de tiempo indicando el inicio y el fin del período de relación laboral a la relación EMPLEADO. Lo mismo ocurriría si quisiéramos mantener el período de existencia de un local dado. Por lo tanto, expandimos las relaciones de la siguiente forma:

EMPLEADOS(EmpId, EmpNombre, LocId, ingreso, egreso)

LOCALES(LocId, LocNombre, creacion, eliminacion)

Una consulta simple podría consistir en conocer los nombres de los empleados que trabajaron mientras existió el local 'Cabildo' y sus períodos de existencia. Una forma de escribir dicha consulta en SQL sería la siguiente:

```
SELECT EmpNombre, ingreso, egreso, creacion, eliminacion
FROM empleados, locales
WHERE ( ingreso BETWEEN creacion AND eliminacion OR
Egreso BETWEEN creacion AND eliminacion OR
Creacion BETWEEN ingreso AND egreso ) AND
LocNombre = 'Cabildo' AND
Empleados.LocId = locales.LocId;
```

En un lenguaje soportado por un SGBD temporal la consulta resultaría más sencilla. Por ejemplo, utilizando el standard de facto TSQL2 que se incorpora al propuesto standard SQL:1999, la consulta se escribiría de la siguiente forma:

```
SELECT EmpNombre, Valid(empleados), Valid(locales)
FROM empleados, locales
WHERE LocNombre = 'Cabildo' AND
Empleados.LocId = locales.LocId;
```

Esto se debe a que la junta tiene una semántica tal que solo son unidas aquellas filas que se superponen en el tiempo y por lo tanto no es necesario incluir el análisis de casos utilizado en la sentencia SQL.

Existen distintas formas de implementar un SGBD temporal. En los últimos años se ha investigado la factibilidad de la construcción de un servidor de Bases de Datos temporales mediante una arquitectura basada en "estratos" [TR2 y TR5]. Esta idea consiste en la introducción de un nivel intermedio o estrato entre la aplicación del usuario y el DBMS relacional. La función de este componente consiste en la traducción de las sentencias escritas en algún lenguaje temporal a SQL para luego delegar la ejecución de las mismas al DBMS subyacente. El estrato es también el encargado preprocesar la respuesta generada por la Base de Datos para ser luego entregada a la aplicación.

Hemos denominado BBTSQL2 al lenguaje utilizado en este trabajo ya que se encuentra basado en el TSQL2 y en el modelo Branching Time.

1.1. CONTRIBUCIONES DE LA TESIS

Las contribuciones de la presente Tesis son:

- La definición de un modelo de datos que maneja información histórica, actual y futura permitiendo la definición de estados hipotéticos de una Base de Datos. Este modelo incluye una extensión al standard TSQL2 que soporta la definición de futuros alternativos.

- La implementación del modelo y el lenguaje mencionados en el punto anterior.

- Para soportar el lenguaje de consulta se implementaron dos modalidades: (a) un intérprete de sentencias BBTSQL2; (b) una versión de BBTSQL2 'embebido', que permite su utilización en programas escritos en un lenguaje de tercera generación.

Consideramos que este trabajo constituye una base para el estudio futuro de implementaciones alternativas o para el diseño de un motor de Bases de Datos temporales integrado, es decir, en donde toda la lógica del manejo de la información temporal se encuentra dentro del DBMS.

El resto del trabajo se encuentra organizado de la siguiente forma: el capítulo 2 introduce conceptos y resume el estado del arte en cuanto al soporte temporal en las Bases de Datos. El capítulo 3 presenta el modelo sobre el que hemos basado nuestra implementación como así también las diferencias con los modelos introducidos en el capítulo 2. El capítulo 4 introduce criterios y conceptos que deben ser considerados al implementar un estrato intermedio que aporte el soporte temporal a una Base de Datos relacional. El capítulo 5 muestra detalles de la implementación presentada en este trabajo. Un caso de estudio junto con un análisis de performance de las operaciones más costosas son presentados en el capítulo 6. Finalmente, el capítulo 7 presenta las conclusiones y el trabajo futuro.

CAPITULO 2

DEFINICIONES Y ESTADO DEL ARTE

Las Bases de datos convencionales solo capturan una foto de la realidad, por lo que son insuficientes para aquellas aplicaciones en las que son necesarios los datos del pasado y/o futuro. Una Base de Datos Temporal es una Base de Datos que soporta almacenamiento, borrado y consulta de los datos basados en su validez temporal. Es decir, es una Base de Datos que mantiene datos del pasado, presente y futuro.

En la siguientes tres secciones del presente capitulo se examinarán los conceptos de tiempo de vida, dimensiones de tiempo (tiempo valido, tiempo transaccional, o bitemporal), granularidad y fusión temporal. Estos conceptos serán usados en la descripción del Modelo Conceptual de Datos Bitemporal presentado en la sección 2.4. y en el *Branching Time Conceptual Data Model* presentado en la sección 2.5..

2.1. EL DOMINIO TEMPORAL

2.1.1. LIFESPANS O TIEMPOS DE VIDA

El tiempo de vida de un objeto representa explícitamente aquellos momentos en los cuales el objeto en cuestión es "visible" para los usuarios de la Base de Datos. Las consultas que se refieran a instantes que se encuentran fuera de *lifespan* de un dato x, son tratadas de manera distinta que las Bases de Datos que no poseen manejo temporal ya que las Bases de Datos Temporales no modelan a x fuera de su lifespan. A modo de ejemplo podemos considerar la relación "empleados" de la figura 2.1. Supongamos que en el año 1993 se decidió contratar a `Antonio` para trabajar en la sucursal `Cabildo` para el período 1994 - 1999. El *lifespan* de este objeto sería justamente el período 1994 – 1999. Si en el año 2000 queremos realizar la consulta acerca de que empleados trabajaron en el período 1996 - 1999 veremos que `Antonio` es uno de ellos. En cambio si queremos saber quienes trabajan en el año 2000, no será considerado parte

de la respuesta.

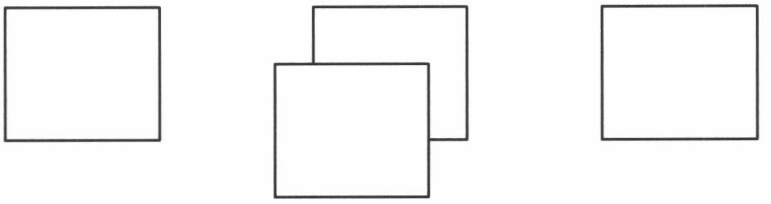
- **Figura 2.1** Tiempo de vida de un objeto.

Surge entonces la pregunta: ¿A nivel de qué objeto conviene asociar dicho *lifespan*?. En particular, los modelos de datos distinguen entre el esquema y la instancia y proveen sentencias para ambos tipos. En el estudio de la Bases de Datos temporales, el mayor foco ha sido puesto sobre la instancia dado que son los *lifespans* de los objetos de datos los que resultan de interés.

Si se asocia el *lifespan* a nivel de Base de Datos lo que resulta es un conjunto de relaciones que son homogeneas en la dimensión temporal (figura 2.2). Es decir, todas las relaciones son contemporáneas ya que todas tienen el mismo *lifespan*. Asociar el *lifespan* a nivel de Base de Datos no ha sido objeto de estudio.

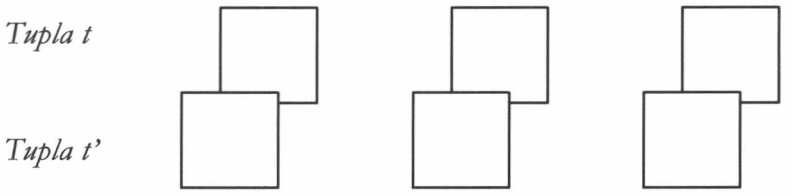
- **Figura 2.2** tres relaciones cuyos lifespans coinciden.

Ahora bien, si se asocia el *lifespan* a cada relación, tendremos una Base de Datos en la que cada relación puede ser definida sobre distintos períodos de tiempo, y donde todas las tuplas de una relación dadas son homogéneas en el tiempo. En la figura 2.3 se puede observar que la relación 2 ha sido modificada en algún instante por lo tanto es válida en dos lifespans distintos.



- **Figura 2.3** Lifespans asociados a las relaciones.

Finalmente, si asociamos el *lifespan* al nivel de tupla, lo que obtenemos es una Base de Datos que se vería como la figura 2.4.



- **Figura 2.4** Lifespans asociados a las tuplas.

La elección del nivel al cual es conveniente asociar el *lifespan* es un compromiso entre costo de mantenimiento, debido a la proliferación de tuplas, y la flexibilidad resultante de *lifespans* de granularidad más fina (a nivel atributo, por ejemplo).

Un *lifespan* simple asociado a todo el esquema de la Base de Datos, podría indicar el período en el cual la base entera estuvo operativa. En el otro extremo, el *lifespan* asociado a nivel atributo puede indicar el período de tiempo en el cual el atributo en cuestión está definido dentro de la relación, permitiendo la posibilidad de esquemas capaces de evolucionar en el tiempo, conocidos como "*evolving schemes*".

2.1.2. TIEMPO VÁLIDO

El tiempo válido o *valid time* de un hecho es el conjunto de instantes o período en los que un hecho es verdadero dentro del *mini-mundo*[1]. El tiempo válido de un evento es el instante en el cual este ocurre en la realidad.

[1] Parte de la realidad modelada por la Base de Datos.

Esta clase de tiempo puede ser futuro si se sabe que un hecho será verdadero en algún instante futuro.

2.1.3. TIEMPO TRANSACCIONAL

Conocemos al tiempo transaccional o *transaction time*, como el tiempo en el cual un hecho esta presente en la Base de Datos como dato almacenado[SNO95].

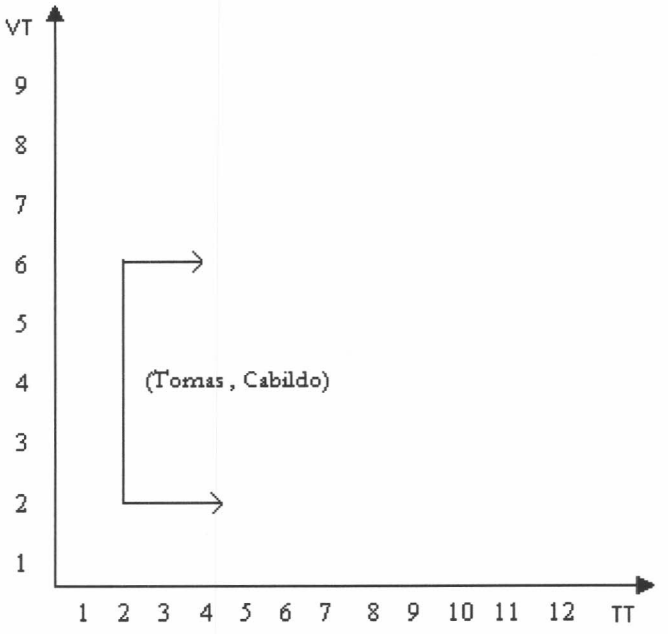
El tiempo transaccional "marcha" en forma monótona hacia adelante y se encuentra limitado por el instante de creación de la Base de Datos y el momento corriente. Este tiempo no puede ser especificado por el usuario y es manejado internamente por la Base de Datos. Debido a la semántica particular del tiempo de transacción, muchos trabajos abordan su estudio por separado.

2.1.3.1. UN EJEMPLO

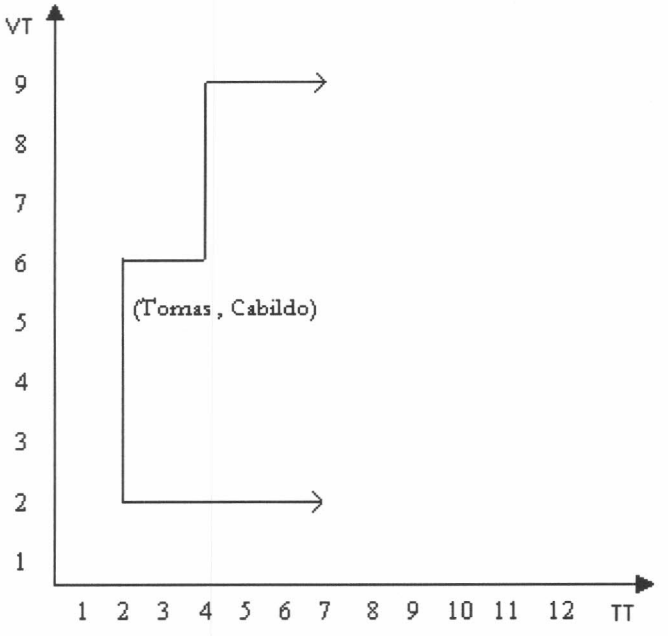
Para ejemplificar el uso de ambas dimensiones, consideremos el caso de la relación EMPLEADOS. Supongamos que en febrero empleamos a Tomás desde febrero hasta junio para trabajar en el local Cabildo. En abril, le extendemos el contrato hasta el mes de septiembre. Finalmente, en julio reducimos las extensión del contrato hasta agosto y lo transferimos al local La Pampa.

La figura 2.5(a) nos muestra que el empleado Tomás fue contratado por la empresa para trabajar en la sucursal Cabildo desde el mes 2 y hasta el mes 6 y este hecho es corriente en la base a partir del mes 2. Las flechas que se prolongan hacia la derecha muestran que las tuplas no han sido lógicamente eliminadas.

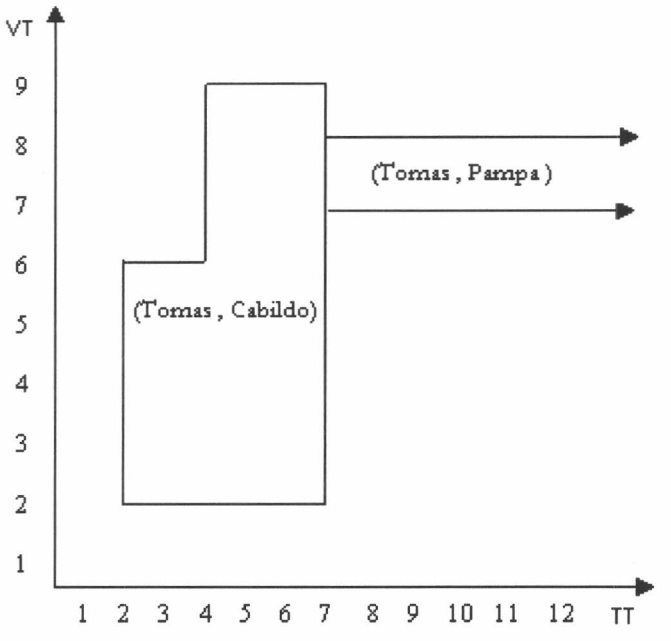
La figura 2.5(b) muestra ciertas correcciones. Se le extiende al empleado Tomás el contrato del mes 6 al mes 9 y este hecho es corriente en la base a partir del mes 4. En el mes 7 transferimos al empleado Tomás a la sucursal La Pampa y se le reduce el contrato hasta el mes 8 como se puede observar en la figura 2.5(c).


VT
9
8
7
6
5
4
3
2
1
(Tomas , Cabildo)
1 2 3 4 5 6 7 8 9 10 11 12 TT

(a)


VT
9
8
7
6
5
4
3
2
1
(Tomas , Cabildo)
1 2 3 4 5 6 7 8 9 10 11 12 TT

(b)



(c)

- **Figura 2.5** : Representación de los tiempos de transacción y validez.

Empleado	Departamento	T
Tomas	Cabildo	{(2,2),...,(2,6),(3,2),...(3,6),(4,2),...,(4,6),(5,2),...,(5,9),(6,2),..., (6,9)}
Tomas	Pampa	{(UC,7),...,(UC,8)}

- **Tabla 2.1**: Instancia de la relación bitemporal

Como veremos en la siguiente sección, el segmento del tiempo puede ser particionado en un número finito de pequeños segmentos discretos llamados gránulos. Un *chronon* es el gránulo más pequeño que se puede representar.

En la tabla 2.1 se ve la instancia de la relación bitemporal en donde el atributo T muestra cada *chronon* del tiempo transaccional asociado con los del tiempo válido como un conjunto de pares ordenados. El valor *Until Change* (UC) indica el instante presente y muestra que la tupla no ha sido eliminada y existe hasta que esta condición sea alterada.

Un modelo de datos que no soporta ninguno de los tiempos mencionados se conoce como *snapshot*. El modelo de datos que soporta solo tiempo válido es llamado *valid-time*. El que soporta

solo tiempo transaccional es llamado *transaction-time* y el que soporta ambos es denominado *bitemporal.*

2.2. LA GRANULARIDAD

El lenguaje de consulta SQL fue diseñado e implementado por IBM Corporation como la interface para una Base de Datos relacional experimental llamada System R. Fue estandarizado por primera vez en 1986 y revisado para crear el standard SQL-92. Actualmente se encuentra en estudio el SQL:1999 anteriormente denominado SQL3 el cual, entre otras características agrega soporte temporal al standard existente.

Las granularidades temporales, por ejemplo, `segundos`, `días`, `semanas` y `meses`, fueron formalizadas inicialmente como particiones de alguna línea de tiempo compuesta por unidades de tiempo indivisibles llamadas *chronons* (denotados usualmente como $\perp$), por ejemplo, `milisegundos`. Un *chronon* puede tener cualquier duración ya sea un segundo, tres días o diez meses. El modelo no determina la duración del *chronon.*

Las Bases de Datos temporales hacen uso intensivo de conversiones de fechas entre granularidades. Este tipo de conversiones y los métodos para realizar estas conversiones fueron estudiados por Hong Lin [TR19]. En este apartado, presentamos definiciones necesarias para abordar una función fundamental en el manejo temporal: la función *Cast.*

Definición 2.1. *Una granularidad α es un conjunto de gránulos contiguos y no superpuestos. A cada gránulo se le asocia un índice entero con el ordenamiento usual de los enteros. Usamos un entero subindexado con la granularidad para identificar los gránulos. La granularidad contiene el gránulo 0_α conocido como el anchor (ancla). Entonces:*

$\alpha = \{ ..., -1_\alpha, 0_\alpha, 1_\alpha, ... \}$

Los *chronons* representan la granularidad mínima. Cada gránulo en una granularidad corresponde a un conjunto de uno o más *chronons.* A fin de facilitar la distinción entre un gránulo de una granularidad cualquiera y la secuencia de *chronons* que lo componen denotaremos a dicho

conjunto como ***CHR***.

Definición 2.2. $\mathbf{CHR}(i_\alpha) = \{c_\perp \mid c_\perp$ *está en* $i_\alpha\}$, *donde* i_α *es el i-ésimo gránulo en la granularidad y* $c_\perp$ *es un chronon.*

Formalmente, las definiciones 2.1 y 2.2 implican lo siguiente:

- Para *chronons* $c_\perp$, $c'_\perp$ y gránulos i_α, j_α, $c_\perp \in \mathbf{CHR}(i_\alpha)$, $c'_\perp \in \mathbf{CHR}(j_\alpha)$ y $c_\perp < c'_\perp$ implica $i_\alpha \leq j_\alpha$.
- Para *chronons* $c_\perp$, $c'_\perp$ y $c''_\perp$, $c_\perp < c'_\perp < c''_\perp$, $c_\perp \in \mathbf{CHR}(i_\alpha)$ y $c''_\perp \in \mathbf{CHR}(i_\alpha)$ implica que $c'_\perp \in \mathbf{CHR}(i_\alpha)$.
- Para gránulos i_α y j_α, $i_\alpha \neq j_\alpha$ implica $\mathbf{CHR}(i_\alpha) \cap \mathbf{CHR}(j_\alpha) = \varnothing$.

La primer propiedad dice que los *chronons* y los gránulos se encuentran totalmente ordenados. La segunda implica que un gránulo contiene conjuntos de *chronons* contiguos y por último, la tercera asegura que diferentes gránulos no se solapan.

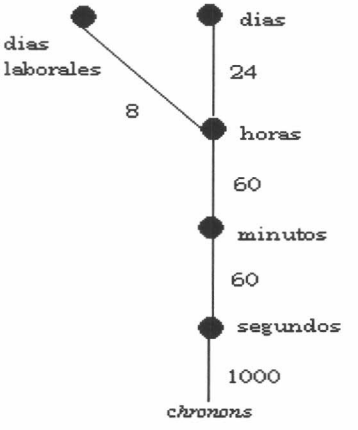
La operación básica que permite la conversión de un gránulo de una granularidad a otro de otra granularidad es la función *Cast*. Otras operaciones pueden escribirse en función de *Cast*.

Definición 2.3. Sean α *y* β *dos granularidades,* $\mathbf{Cast}(i_\alpha, \alpha, \beta)$ *es* $\mathbf{Cast}(i_\alpha, \alpha, \beta) \rightarrow j_\beta$ *tal que i es el i-ésimo gránulo en* α *y* j_β *es el j-ésimo gránulo en* β.

Entre las distintas granularidades existe una relación "más fina que". Esta relación es un orden parcial en donde el ínfimo esta representado por el *chronon* (la granularidad más pequeña posible). Wang [WA93] demostró que este orden define un álgebra de Boole. Relacionando una unidad arbitraria con la más pequeña del conjunto se puede encontrar un camino para llegar desde una granularidad a cualquier otra. Esto implica que es matemáticamente posible encontrar un método para la conversión entre granularidades, aunque no necesariamente práctico. Los sistemas caléndricos desconocen cual es la mínima granularidad representable y estos, generalmente, se encuentran basados en otros calendarios bien conocidos. Se debe notar que si las granularidades que desean ser representadas no forman un álgebra de Boole debe procederse

a incorporar granularidades "artificiales" para poder lograr las conversiones, lo cual puede convertirse en una tarea no trivial.

Según observaron Dyreson y Snodgrass [DYSN94], las interacciones entre la mayoría de las granularidades es normalmente lograda mediante una multiplicación o división más alguna corrección. Esto se debe a que las granularidades utilizadas son, generalmente, subdivisiones unas de otras. *El grafo de granularidades* representa los caminos que deben recorrerse para convertir gránulos de una a otra granularidad. Un arco orientado de *g* a *h* implica que *g* es "más fino" que *h* en términos de granularidad. Los mapeos entre granularidades pueden ser de tres clases: *regulares* son aquellos mapeos que pueden lograrse mediante alguna multiplicación o división simple con una constante de conversión (por ejemplo `horas` a `minutos`), *irregulares* son los que no pueden ser convertidos mediante simples multiplicaciones o divisiones y finalmente se encuentran los mapeos *congruentes* que consisten en aquellos cuyos gránulos son idénticos, pero sus constantes de corrección son diferentes (por ejemplo de días Gregorianos al calendario formado por los días hábiles). En la figura 2.6 se muestra un grafo de granularidades en el que se describe un sendero a partir de la granularidad horas.



- **Figura 2.6** Un grafo de granularidades.

Dyreson y Snodgrass propusieron un método para realizar la operación Cast. El primer paso consiste en encontrar el camino dentro del grafo de granularidades entre la granularidad origen y

la destino, utilizando los mapeos existentes. Ellos conjeturaron que todos los senderos-V descendentes a un ancestro común y luego hacia arriba (debido a esta forma se los conoce como senderos-V o *V-paths*) llevan a resultados equivalentes, pero los caminos pueden diferir en los costos en términos de cálculo, debido a las funciones que deben ser invocadas para la conversión.

2.2.1. CONVERSIÓN

No todos los caminos entre cualquier par de granularidades son buenos para realizar la conversión. Dado que cualquier camino ascendente (de una granularidad más fina a una más gruesa) pierde información, cualquier conversión que realice un camino ascendente y luego descendente puede llevar a un resultado incorrecto. Por ejemplo:

`Cast('01/02/2000',dias,meses)='01/2000'` si se elige el camino `días` a `años` y luego `meses`, mientras que el resultado correcto es '02/2000'.

Definición 2.4. *Si la granularidad β es una subpartición de α, esto es, si $\forall j_\beta$, $CHR(j_\beta) \subseteq CHR(i_\alpha)$, entonces β es más fina que la granularidad α y se expresa como $\beta < \alpha$.*

La relación < define un orden parcial. Por ejemplo `semanas` no es más fino que `meses`, ni viceversa.

Finalmente, se puede dar un mecanismo para realizar el *Cast.*

Si $\alpha > \beta$ entonces $Cast(i_\alpha, \alpha, \beta) \rightarrow j_\beta$ tal que $min(\mathbf{CHR}(i_\alpha)) = min(\mathbf{CHR}(j_\beta))$,

Si $\alpha < \beta$ entonces $Cast(i_\alpha, \alpha, \beta) \rightarrow j_\beta$ tal que $\mathbf{CHR}(i_\alpha) \subseteq \mathbf{CHR}(j_\beta)$

2.3. FUSIÓN TEMPORAL

La fusión temporal o *coalesce* es un operador unario aplicable a las Bases de Datos temporales y su funcionamiento es similar a la eliminación de duplicados [TR9].

No me queda claro porque no dicen que la conversión no siempre es posible. No entiendo porque hacer los descenso

Supongamos que contamos con una relación que mantiene la información de empleados junto a sus marcas de tiempo indicando su período de validez en granularidad `meses/años`.

Nombre	Desde	Hasta
Perez	02/1994	07/1996
Perez	08/1996	04/1998

En álgebra relacional, la consulta sobre qué empleados trabajaron más de tres años en la empresa, produciría un resultado vacío ya que en ninguno de los casos el período de validez supera los tres años. La sentencia para realizar dicha consulta es la siguiente:

```
select nombre
from empleados
where to_number(to_char(desde,'yyyy')-to_number(to_char(hasta,'yyyy'))>3;
```
[2]

Si por otro lado, el resultado estuviese fusionado temporalmente previamente a la consulta, `Perez` formaría parte del resultado ya que trabajó durante más de cinco años en la empresa. Existen diferencias semánticas entre las relaciones fusionadas y las no fusionadas que no pueden ser pasadas por alto por parte de las aplicaciones. La sentencia TSQL2 necesaria para obtener el resultado deseado es la siguiente:

```
SELECT nombre
FROM empleados
WHERE END(VALID(empleados))-BEGIN(VALID(empleados)) > INTERVAL '3' YEAR;
```

La función **`VALID`** retorna el tiempo válido de las uplas de la relación. La función destructora **`BEGIN`** retorna el comienzo de un período, mientras que la función destructora **`END`** retorna el fin de un período. Por ultimo, **`INTERVAL`** `'3'` **`YEAR`** describe un lapso de 3 años no anclado en el tiempo.

Las operaciones como la unión, proyección, inserción y actualización pueden llevar a relaciones no fusionadas. Debido a que estas son operaciones muy frecuentes en las Bases de Datos, se requiere una implementación eficiente para esta operación. El *coalesce* es una operación más costosa que la eliminación de duplicados dado que requiere de la detección de solapamiento temporal y esto representa un predicado de desigualdad basado en los atributos temporales.

[2] Aquí hemos hecho uso de algunas funciones que provee Oracle para ejemplificar la consulta en SQL.

El término *coalesce* fue acuñado por Snodgrass en la descripción del modelo de datos de su lenguaje TQuel [SNO87]. El HSQL [SAR90] incluía una cláusula explícita para producir un conjunto fusionado temporalmente. En TSQL2, el *coalesce* es implícito y el lenguaje no provee ninguna construcción para proveer un conjunto no fusionado.

2.3.1. DEFINICIONES

Para las definiciones que presentamos aquí, consideramos que contamos con relaciones a cuyas tuplas agregamos *timestamps* correspondientes a intervalos abiertos $[t_i, t_j)$, representando su tiempo de validez, es decir, el período de tiempo en el cual el hecho es modelado en el mini-mundo.

Para las definiciones que siguen, consideramos una relación $R = (A_1, ..., A_n | VT)$, en donde, $\{A_1, ..., A_n\}=A$ es el conjunto de atributos explícitos y un período de validez VT=[I, F). Denotaremos con *r* a una instancia de *R* y *x* e *y* como tuplas en *r*.

***Definición 2.5**: dos tuplas son equivalentes en valores (value equivalent) si $x[A] = y[A]$.*

Es decir, que diremos que dos tuplas son *value-equivalent* cuando todos los atributos explícitos son iguales. La siguientes dos definiciones involucran *timestamps*.

***Definición 2.6**: dos timestamps son adyacentes si $([I_1, F_1),[I_2,F_2)) \Rightarrow (F_1 = I_2 \vee I_1 = F_2)$.*

***Definición 2.7**: dos timestamps $[I_1,F_1)$ y $[I_2,F_2)$ se solapan si $(\exists c ((I_1 \leq c < F_1) \wedge (I_2 \leq c < F_2)$*

Podemos decir que una relación se encuentra temporalmente fusionada cuando no contiene ningún par de tuplas *value-equivalent* tal que sus timestamps son adyacentes o se solapan en el tiempo.

***Definición 2.8**: Sea $R= (A_1, ..., A_n | VT)$ una relación y r una instancia de R. Decimos que r está temporalmente fusionada (coalesced) sii:*

$$\forall x \in r, y \in r,\ x \neq y \ (\\ (\neg \ value\_equivalent(x, y)) \\ \vee (\neg \ overlap(x[VT], y[VT]) \wedge \neg \ adjacent(x[VT], y[VT])))$$

2.3.2. EL ÁLGEBRA TEMPORAL

Haciendo uso del concepto de fusión temporal introducido anteriormente, es posible definir un álgebra temporal como una extensión del álgebra relacional propuesta por Codd [COD70].

Definición 2.9*: Las semántica del álgebra basada en períodos se define como sigue:*

(Selección) $\sigma_c^T r \equiv \{(t||[I,F)) / (t||[I,F)) \in r \wedge c((t||[I,F)))\}$

(Proyección) $\Pi_c^T r \equiv \{(t_1||[I,F)) / (t||[I,F)) \in r \wedge t_1 = f((t_2||[I,F)))\}$

(Unión) $r_1 \cup^T r_2 \equiv \{(t||[I,F)) / (t||[I,F)) \in r_1 \vee (t||[I,F)) \in r_2\}$

(Producto) $r_1 \times^T r_2 \equiv \{((t,[A,B)) \circ (t2,[C,D)) ||[I,F)) /$
$(t_1||[A,B)) \in r_1 \wedge (t_2||[C,D)) \in r_2 \wedge$
$I = max(A,C) \wedge F = min(B,D) \wedge I < F\}$

(Resta) $r_1 \setminus^T r_2 \equiv \{(t||[I,F)) / (t||[A,B)) \in r_1 \wedge$
$\exists C ((t||[C,I)) \in r_2 \wedge A \leq I) \wedge$
$\exists D ((t||[F,D)) \in r_2 \wedge B \geq F) \wedge I < F \wedge$
$\neg\exists U,V ((t||[U,V)) \in r_2 \wedge V > I \wedge U < F)\}$

2.3.2.1. EJEMPLO

Sean los siguientes esquemas R_{11} = (A, B||VT) y R_{21} = (A, B||VT) y sean r_{11} ,r_{21} instancias tal que r_{11}={<a, b||[2, 5)>, <a, c||[4, 10) >, <b, c||[8, 12) >} y r_{21}={<a,b||[5,10) >, <b, c||[10, 12) >}.

$\sigma_{A=a}^{(4,5)} r_{11} \equiv$ {<a, b||[2, 5)>, <a, c||[4, 10) >}

$\Pi_B^{(1,20)} r_{11} \equiv$ {< b||[2, 5)> , < c||[4, 10) >}, < c||[8, 12) >}

$r_{11} \cup^{(1,20)} r_{21} \equiv$ { <a, b||[2, 5)> , <a, c||[4, 10) > , <b, c||[8, 12) > ,
<a,b||[5,10) > , <b, c||[10, 12) >}

$r_{11} \times^{(1,20)} r_{21} \equiv$ {<a,b,a,b||[5,5)>, <a,c,a,b||[5,10)>, <b,c,a,b||[8,10)>,
<a,c,b,c||[10,10)>, <b,c,b,c||[10,12) >}

$r_{11} \setminus^{(1,20)} r_{21} \equiv \{<a, b||[2, 5)>, <a, c||[4, 10) >, <b, c||[8, 10) >\}$

Snodgrass demostró que la proyección y la unión pueden potencialmente destruir la fusión temporal. Para mostrar esta posibilidad basta un ejemplo.

Sean los siguientes esquemas R_1 = (A, B||VT) y R_2 = (A, B||VT) y sean r_1 ,r_2 instancias fusionadas tal que r_1={<a, b||[2, 5)>, <a, c||[4, 10) >} y r_2={<a,b||[5,10) >}.

Por la definición de proyección $\prod_{A}{}^{v}(r_1)$ = {<a||[2,5)>,<a||[4,10)} y $r_1 \cup^v r_2$ = {<a,b||[2,5)>,<a,c||[4,10)>,<a,b||[5,10)>}. Ambos resultados no se encuentran temporalmente fusionados ya que contienen filas de valores equivalentes con períodos solapados.

Snodgrass también demostró que el producto cartesiano, la negación y la selección preservan la fusión temporal. Esta demostración puede encontrarse en [TR9].

Debido a que la operación que realiza la fusión temporal es computacionalmente costosa (es al menos el de la eliminación de duplicados lo que implica un ordenamiento de las tuplas) debe evitarse el fusionamiento cada vez que esto sea posible. En el siguiente apartado, mostramos algunas propiedades de la operación coalesce que permiten determinar cuando es necesario llevarla a cabo.

2.3.3. REGLAS PARA LA FUSION TEMPORAL

Las reglas para la aplicación de la función fusión temporal o **coalesce** se basan en las definiciones y propiedades enunciadas en el apartado anterior.

Regla 0: **coalesce**(**coalesce**(r_1)) = **coalesce**(r_1)

Regla 1: **coalesce**($r_1 \times^v r_2$) = **coalesce**(r_1) x **coalesce**(r_2)

Regla 2: **coalesce**($r_1 \setminus^v r_2$) = **coalesce**(r_1) \ **coalesce**(r_2)

Regla 3: **coalesce**(σ_c^p(**coalesce**(r_1)))= σ_c^p(**coalesce**(r_1))

Regla 4: **coalesce**(**coalesce**(r_1) $\cup^v$ **coalesce**(r_2))= **coalesce**($r_1 \cup^v r_2$)

En algunos casos, es posible eliminar el *coalesce* si se conoce que la relación ya se encuentra fusionada previamente (por definición del modelo, por ejemplo). En la implementación presentada en esta Tesis , este es el caso, es decir que toda relación se encuentra fusionada. *Solo es necesario aplicar la función coalesce ante la presencia de alguna operación que pueda destruir dicha fusión.*

Las reglas 0 a 4 son de aplicación incondicional, es decir, que siempre es posible aplicarlas. Este no es el caso de la proyección si se realiza algún cálculo basado en el tiempo válido de la relación. Esto sucede porque el resultado puede variar dependiendo en el formato del *timestamp* de la relación de entrada.

Existen además, algunas reglas de fusión condicional. En algunos casos, es posible retardar la aplicación de la función **coalesce**. Si la condición de selección no restringe el tiempo válido de la relación entonces vale la siguiente regla:

Regla 5: σ^{v}_{c}(**coalesce**(r_1)) = **coalesce**($\sigma_c(r_1)$) si c es independiente del tiempo válido de r_1

Regla 6: **coalesce**($\prod_{c}{}^{v}$ (**coalesce**(r_1))) = **coalesce**($\prod_{c}{}^{v}$ (r_1)) si *c* es independiente del tiempo válido de r_1.

El requisito de que c sea independiente del tiempo válido de la relación se debe a que si la relación no se encuentra fusionada previamente y el predicado incluye a dicho tiempo, existe el potencial de que ocurra una situación como la descripta en el ejemplo del apartado 4.

Debido a que nuestro modelo garantiza que las relaciones temporales siempre se encuentran fusionadas, esto es, **coalesce**(r_1) = r_1, entonces la regla 3, resulta en **coalesce**(σ^{v}_{c}(**coalesce**(r_1)))= $\sigma^{v}_{c}(r_1)$. Esto resulta muy importante ya que reduce en gran medida la aplicación de la función *coalesce* durante las consultas.

2.4. EL MODELO CONCEPTUAL DE DATOS BITEMPORAL

El modelo conceptual de datos bitemporal o BCDM agrega soporte de tiempo al SQL-92, soportando la combinación del tiempo válido y el tiempo transaccional. Comenzamos especificando los aspectos estructurales del dominio del tiempo asumidos por el modelo de

datos para luego describir los objetos del modelo y ciertas consideraciones de cómo se pueden actualizar estos modelos.

2.4.1. DOMINIO DEL TIEMPO

Asumimos un modelo de tiempo lineal y discreto acotado tanto para el dominio de tiempo válido como para el transaccional. Se utilizan los *chronons* en la representación de los *timestamps*. Estos tienen una duración que es múltiplo o fracción de segundos. El dominio temporal es isomorfo a los números naturales ya que los *chronons* son numerables.

2.4.2. OBJETOS EN EL MODELO

Las tupla correspondientes a una instancia de una relación conceptual bitemporal están asociadas con los valores de tiempo tanto para el tiempo válido como para el tiempo transaccional. Como se mencionó anteriormente la unidad de tiempo más pequeña es denominada *chronon*. El dominio del tiempo tiene un orden total y ambos tiempos son isomorfos a subconjuntos del dominio de los números naturales.

Un subconjunto arbitrario del dominio de tiempo válido es asociado con cada tupla, significando que los datos almacenados por las tuplas son verdaderos en la realidad modelada durante cada *chronon* de tiempo válido en el subconjunto. Individualmente cada *chronon* de tiempo válido de una tupla tiene asociado un subconjunto arbitrario del dominio del tiempo transaccional, es decir, el hecho, válido durante ese *chronon* particular, es corriente en la relación durante cada *chronon* del subconjunto del tiempo transaccional. Esto se puede ver en el apartado 2.1.3.1.

Existe un elemento temporal asociado con cada tupla, denotado t_b, consistente de *bitemporal chronons* en el espacio bidimensional cubierto por el tiempo válido y el tiempo transaccional. Se representa a un *chronon* bitemporal como un par ordenado (t,v), con tiempo transaccional t y tiempo válido v. En una relación bitemporal no son permitidas dos tuplas *value-equivalent*, por lo que toda la historia de un hecho esta contenida en un misma tupla.

2.4.2.1. SEMANTICA DE LA INSERCIÓN, BORRADO Y MODIFICACIÓN.

La inserción es la operación por la cual una tupla <a_1,...,a_n> es agregada a una relación bitemporal r, manténiendose durante un cierto período de tiempo. Así una tupla $x=<a_1,...,a_n|t_b>$ en una instancia de una relación conceptual bitemporal, r(R),consiste de un número de valores de atributos asociados con un valor de tiempo.

$$insert(r,(a_1,...,a_n),t_v)=\begin{cases} r \cup\{(a_1,...,a_n|\{UC\}\times t_v)\} \; si \; \neg\exists t_b((a_1,...,a_n|t_b) \in r) \\ r-\{(a_1,...,a_n|t_b)\} \cup \{(a_1,...,a_n|t_b \cup \{\{UC\}\times t_v\})\} \\ \qquad\qquad si \; \exists t_b((a_1,...,a_n|t_b) \in r \wedge \neg\exists(UC,c_v) \in t_b \\ r \; sino \end{cases}$$

Borrar significa a remover lógicamente la tupla, esto es cambiar el estado del tiempo válido de una relación bitemporal. Donde $uc\_ts=\{(UC,c_v)|(UC,c_v) \in t_b\}$ y c_v es el *timestamp* de tiempo valido del algún *chronon* de las tuplas.

$$delete(r,(a_1,...,a_n))=\begin{cases} r-\{(a_1,...,a_n|t_b)\} \cup \{(a_1,...,a_n|t_b - uc\_ts(t_b))\} \\ \qquad\qquad si \exists t_b((a_1,...,a_n|t_b) \in r) \\ r \; sino \end{cases}$$

Finalmente la modificación de una tupla existente está definida por un borrado seguida de una inserción.

$$update(r,(a_1,...,a_n),t_v)=insert(delete(r,(a_1,...,a_n)),(a_1,...,a_n),t_v)$$

La relación conceptual introducida en el apartado 2.1.3.1 es creada por la siguiente secuencia de comandos .

Comando	Tiempo transaccional
`Insert(empleado,("Tomas","Cabildo"),[2,6])`	2
`Update(empleado,("Tomas","Cabildo"),[2,9])`	4
`Delete(empleado,("Tomas","Cabildo"))`	7
`Insert(empleado,("Tomas","Pampa"),[7,8])`	7

2.4.2.2. ESQUEMA

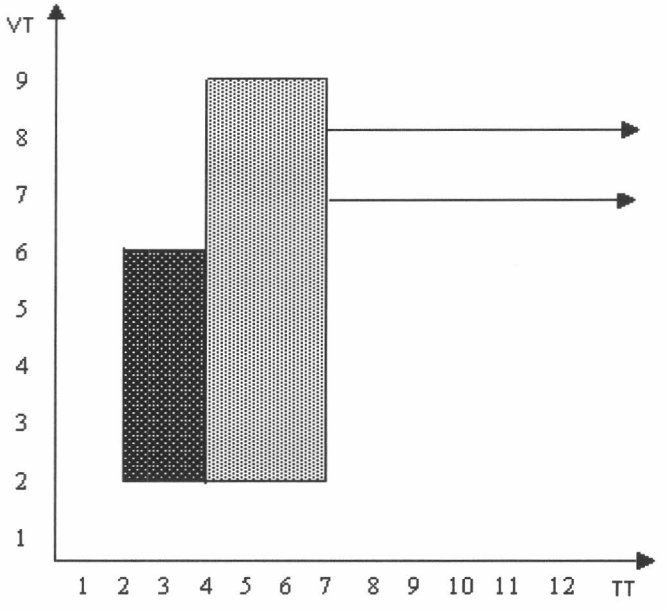
Sea un esquema de relación bitemporal R con atributos a_1, . . ., a_n , T donde T es el atributo

timestamp definido en el dominio de los elementos bitemporales. R está representado por un esquema de relación snapshot R, es decir $R=(a_1,\ldots,a_n,T_s,T_e,V_s,V_e)$.

Los atributos T_s , T_e , V_s , V_e son valores atómicos que contienen el *chronon* de comienzo y fin del tiempo transaccional , y, el *chronon* de comienzo y fin del tiempo válido respectivamente. Estos cuatro valores representan los *chronons* bitemporales en una región rectangular: la idea es cubrir la región representada por el elemento bitemporal de la tupla conceptual en un número de rectángulos y luego representar la tupla conceptual por un conjunto de tuplas, una por cada rectángulo.

Se usa una función de cobertura (que no tiene solapamiento entre los rectángulos) que particiona los elementos bitemporales por el tiempo de transacción como se ve en la figura 2.7.

Emp	Local	T_s	T_e	V_s	V_e
Tomas	Cabildo	2	3	2	6
Tomas	Cabildo	4	6	2	9
Tomas	Pampa	7	UC	7	8

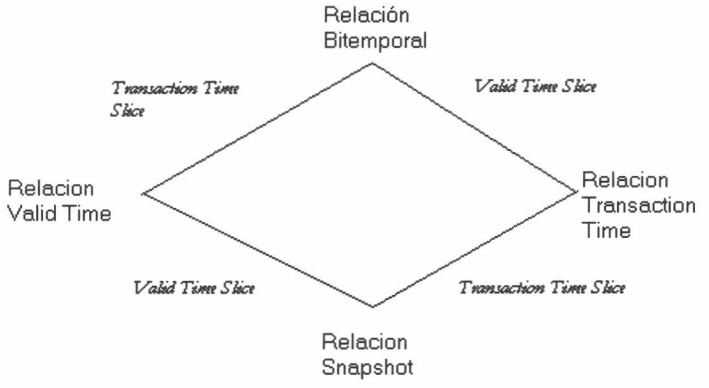


- **Figura 2.7**: Cobertura de los elementos bitemporales particionada por el tiempo transaccional.

Porque el UC aparece explícitamente en la base?
Por otra parte creo que el esquema de representación para
$(tt,tv) \circ ((T_S T_e),(V_S,V_e))$

2.4.3. SEMANTICA DE EQUIVALENCIA

A continuación se muestra la equivalencia entre los objetos del modelo, para esto son consideradas las operaciones *transaction-timeslice* y *valid-timeslice* sobre los objetos.



- **Figura 2.8** Diferentes niveles del soporte temporal

La equivalencia entre dos o más relaciones *snapshot* captura la noción de que las instancias de dos o más relaciones pertenecientes al esquema de representación elegido tengan el mismo contenido de información. Es decir, todas las representaciones de una misma relación conceptual bitemporal son *snapshot equivalent* y dos relaciones bitemporales que son *snapshot equivalent* representan la misma relación conceptual bitemporal.

Definición 2.9: *dos instancias de una relación, r y s son snaphot equivalent* $r \equiv s$ *, si para todo tiempo* t_1 *tal que* t_1 *no exceda el tiempo corriente y para todo tiempo* t_2 *, valid-timeslice* ^{V} t_2 *(transaction-timeslice* ^{B} t_1 *(r)) = valid-timeslice* ^{V} t_2 *(transaction-timeslice* ^{B} t_1 *(s))*

2.4.4. GRANULARIDAD

Este lenguaje construido a partir del modelo presentado anteriormente soporta granularidades mezcladas y calendarios múltiples. La representación interna de estas granularidades está, generalmente, basada en números enteros, pero el lenguaje contiene funciones para realizar conversiones a formatos externos. El TSQL2 soporta conversiones entre granularidades de distintos sistemas de calendarios solo si el usuario provee las funciones de

mapeo correspondientes. En general, esta no es una tarea trivial.

2.5. BRANCHING TIME CONCEPTUAL DATA MODEL

En muchas situaciones es necesario planificar el futuro. Esta planificación consiste en definir acciones alternativas según los distintos eventos que puedan llegar a suceder en el futuro. Un plan consiste en una secuencia de acciones que serán llevadas a cabo para cada salida posible que tenga un evento. Como la salida de estos eventos es incierta necesitamos incorporar las alternativas en el tiempo futuro.

N.L. Sarda y P.V. Siva Prasada Reddy extienden el álgebra temporal relacional y TSQL2 para soportar un nuevo modelo denominado *branching time conceptual data model* [SA97], incorporando alternativas en el tiempo futuro dado que la salida de los eventos es incierta.

El tratamiento del tiempo futuro incorpora los conceptos de "evento" y "acción". Un evento es un hito que ocurre en un momento determinado y genera una salida. Un conjunto de acciones diferente es llevado a cabo dependiendo de la ocurrencia o no de dicho evento. En un entorno de Base de Datos, las acciones posibles son: la inserción, la eliminación o la actualización de sus filas. Desde el punto de vista de la planificación, los eventos son aquellas decisiones críticas que introducen incertidumbres en el futuro y las acciones son las actividades propuestas para manejar estas posibles salidas.

Para representar situaciones como las descriptas anteriormente se usa un árbol de eventos o *event tree* que puede ser visto como un árbol binario en donde los nodos son eventos rotulados y los arcos representan la dependencia entre los eventos y el paso del tiempo.

2.5.1. OBJETOS EN EL MODELO

2.5.1.1. EVENTOS

La ocurrencia de un evento puede depender de la ocurrencia de otros eventos. Esta dependencia puede definirse usando expresiones de eventos. Todo evento posible tiene dos salidas *true* y *false*. Se asocia un único identificador con cada evento y se asocia un valor con éste para representar su salida.

Si el evento aún no ha ocurrido se le asocia un valor especial "**not_yet_known**", si el evento ocurre el valor será "**true**" y si no ocurre será "**false**".

Un evento es caracterizado por la terna <*Id del evento, tiempo de ocurrencia, Expresión de evento*> en donde el tiempo de ocurrencia es el momento en el cual la salida del evento se hace efectiva. Las expresiones de eventos deben contener los eventos en el orden cronológico de ocurrencia de los mismos. Un evento puede ocurrir si su expresión de evento es verdadera ("**true**") y no ocurre si su expresión de evento es falsa ("**false**"). No toda combinación de expresiones de eventos modela el estado de un objeto. La tabla 2.2 muestra tres eventos junto con sus propiedades características.

La ocurrencia de un evento es externa a la Base de Datos y debe ser señalada de forma explícita.

ID.	DESCRIPCION	TIEMPO DE OCURRENCIA	EXPRESIÓN DE EVENTO
E1	Aumento de salario	10/1	True
E2	Nuevo producto	20/1	True
E3	Modernización	30/1	Not E2

- **Tabla 2.2** Ejemplo de Eventos

2.5.1.2. ACCIONES

Una acción es una operación en la Base de Datos que deberá ser ejecutada sólo si ocurre una combinación de eventos.

Un acción se caracteriza por <*Expresión de evento, Operación*> en donde, expresión de evento es la combinación de las salidas de los eventos bajo las cuales tiene validez la acción y operación es la operación de inserción, borrado o actualización de la Base de Datos resultante de llevar a cabo la acción.

En el ejemplo de la tabla 2.3 se muestra que la acción `A1` actualizará el campo `salario` cuando la expresión de eventos `E1` sea "**true**".

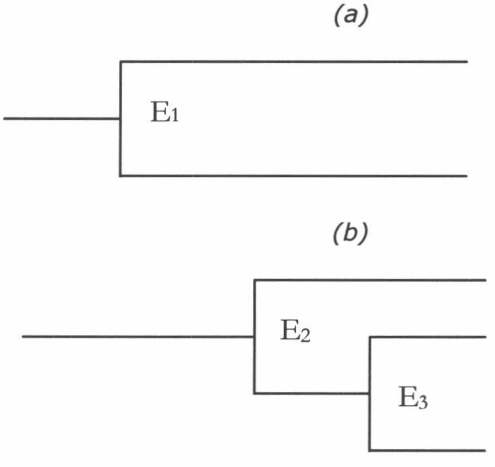
ID.	OPERACIÓN	EXPRESIÓN DE EVENTO
A1	Salario=Salario*1.1	E1

- **Tabla 2.3** Ejemplo de Acciones

Se ejecuta una acción cuando su correspondiente expresión de eventos es "**true**". Una acción es ejecutada tan pronto como es definida, dando origen inmediatamente a la apertura de futuros alternativos. Cabe aclarar, que los datos que reflejan el efecto de esta acción solo tendrán validez temporal cuando su expresión de evento sea verdadera.

2.5.1.3. ARBOL DE EVENTOS

El árbol de eventos (*event tree*) puede ser visto como un árbol binario en donde los nodos son eventos rotulados. En cada nodo, uno de los subárboles contiene eventos que pueden ocurrir dada la ocurrencia del evento asociado y el otro contiene los eventos que pueden ocurrir dada la no ocurrencia de este. Es decir, cada subárbol representa una alternativa en el futuro. Los arcos representan la dependencia entre los eventos y el paso del tiempo. Un arco representa una expresión de evento y una expresión de evento corresponde a un arco.

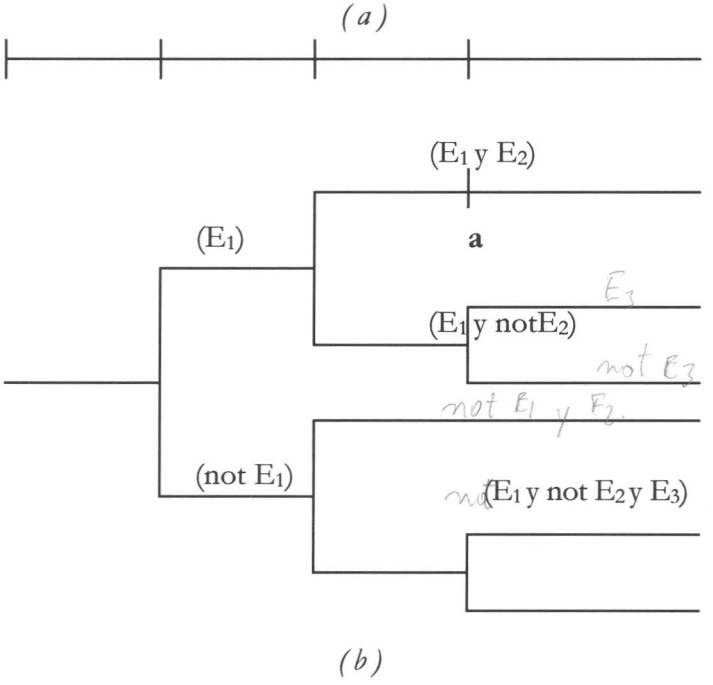


- **Figura 2.9.** Arbol de eventos

2.5.2. DOMINIO TEMPORAL

El árbol de eventos nos muestra que pueden ocurrir diferentes conjuntos de acciones dependiendo de las diferentes salidas de los eventos. En cierto punto del tiempo el curso futuro de un evento puede seguir dos o más caminos alternativos. En otras palabras, hay más de un futuro y cada

uno de estos está calificado con una expresión de evento. Para considerar las diferentes alternativas en el futuro se superpone al árbol de eventos un tiempo válido lineal y discreto. Esta superimposición nos permite ver el tiempo como *branching time*, en donde el tiempo se bifurca en cada evento en dos caminos, uno por cada salida. En la representación gráfica de la figura 2.10(b) se muestra la naturaleza ramificada del tiempo y en la figura 2.10(a) el tiempo lineal correspondiente.



- **Figura 2.10:** Branching time tree

En el modelo propuesto, un *chronon* es una combinación de intervalos de tiempo y una expresión de evento. En la figura 2.10 se puede ver el *chronon* **"a"** (4,(E_1 y E_2)), donde 4 es el tiempo válido y (E_1 y E_2) su expresión de eventos. Distintos *chronons* pueden tener igual valor de tiempo y distintas expresiones de eventos.

2.5.2.1. DEFINICIONES DE LA SEMÁNTICA DEL DOMINIO DEL TIEMPO *BRANCHING*

Se representa a un *branching time chronon bt* como un par (*v*,*e*), donde *v* corresponde al tiempo válido del *chronon* y *e* representa una expresión de evento.

El tiempo transaccional no tiene hijos?

Sean $bt(v,e)$, $bt_1(v_1,e_1)$ y $bt_2(v_2,e_2)$ *branching chronons*. Los siguientes predicados caracterizan a los *bt*

- *propio(bt)* : si *v* es mayor o igual que el tiempo de ocurrencia del último evento (o expresión de evento) contenido en *e*, ordenados cronológicamente.
- *alcanzable*(bt_1,bt_2) : bt_1 es alcanzable desde bt_2 si hay un posible camino desde bt_1 a bt_2.
- *sucesor* (bt_1,bt_2): bt_2 es un sucesor de bt_1 si $v_2 = v_1 + 1$ y bt_2 es alcanzable por bt_1.
- *igual*(bt_1,bt_2): bt_1 es igual a bt_2 si $v_1 = v_2$ y $e_1 = e_2$.
- *equivalentes_en_el_tiempo*(bt_1,bt_2) : bt_1 es equivalente en el tiempo a bt_2 si $v_1 = v_2$.
- *menor*(bt_1,bt_2): bt_1 es menor que bt_2 si $v_1 < v_2$ y bt_2 es alcanzable desde bt_1. El < impone un orden parcial en el conjunto de los *branching chronons*.
- *cumplido(bt)* : *bt* es cumplido si *e* es *verdadero*.

2.5.2.2. DEFINICIÓN DE *BRANCHING ELEMENT* Y *BRANCHING INTERVALS*.

Un *branching element* es un conjunto de *branching chronons*. Estos *chronons* no tienen que ser sucesivos. Un conjunto de *branching chronons* sucesivos $bt_1,\ldots,bt_n$ representada por [bt_1, bt_n] es un *branching interval*. Un intervalo es homogéneo si todos sus *chronons* tienen la misma expresión de evento. Un intervalo está cumplido si todos sus *chronons* están cumplidos.

Extendemos las operaciones convencionales sobre los intervalos y elementos como *overlaps*, *meets*, *intersects* a intervalos y elementos *branching time*. Las operaciones aritméticas no cambian la expresión de eventos. El *chronon* resultante puede ser o no propio. El tiempo transaccional es lineal y discreto y es tratado como en el modelo conceptual de datos bitemporal.

2.5.3. SEMANTICA DE EQUIVALENCIA

Las siguientes son las estructuras de las relaciones válidas en el modelo:

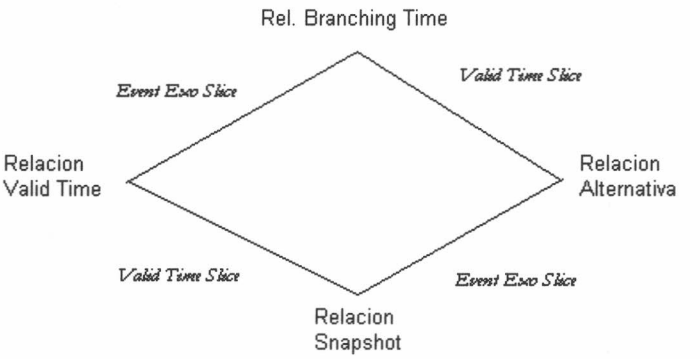
a) Relación *snapshot*, $R(A_{1,\ldots}, A_n)$: Es una relación convencional sin ningún soporte de tiempo implícito.

b) Relación *valid time*, $R(A_{1,...}, A_n \mid V)$: Permite a un tupla $<a_1,\ldots, a_n>$ ser almacenado a lo largo de un tiempo válido v, indicando donde los hechos son verdaderos en el mundo real modelado.

c) Relación alternativa, $R(A_{1,...}, A_n \mid E)$: Permite a una tupla $<a_1,\ldots, a_n>$ ser almacenado y tener validez en el mundo real bajo cierta expresión de eventos e.

d) Relación *Branching time*, $R(A_{1,...}, A_n \mid E \mid V)$: Permite a un hecho, $<a_1,\ldots, a_n>$ ser almacenado a lo largo de un tiempo válido v y tener validez en el mundo real bajo cierta expresión de eventos e.

A continuación se muestra la equivalencia entre los objetos del modelo, para esto son consideradas las operaciones *event expression slice operator* y *valid time slice operator* sobre los objetos.



- **Figura 2.11:** Diferentes niveles del soporte temporal

2.5.4. RELACIONES BRANCHING

2.5.4.1. RELACIÓN CONCEPTUAL *BRANCHING TIME*

El estado de los objetos en el mundo real está dado por el valor de sus atributos. Debido a las diferentes acciones basadas en las diferentes salidas de los eventos, un objeto puede tener más de un estado en el futuro, cada uno calificado con un expresión de evento. Este estado representa el futuro alternativo de este evento. Solo uno de estos estados es cumplido, dependiendo del camino que se recorre en el *branching tree* a lo largo de la línea representada por el tiempo válido.

Sean $A = \{A_1, A_2,..., A_n\}$ un conjunto de nombres de atributos y $D = \{D_1, D_2,..., D_n\}$ un

conjunto de dominios de atributos. Un esquema de relación conceptual *branching time* R consiste de un número arbitrario de atributos explícitos de A, con dominios en D y un elemento temporal implícito, V_b, el cual es un conjunto de *branching chronons*.

Así, una tupla $x^{(n+1)}$ en una instancia r (R) es de la forma $<a_1,a_2,\ldots,a_n \mid v_b>$.

2.5.4.2. COMPARACIÓN DE ELEMENTOS TEMPORALES BRANCHING

Los *temporal branching elements* son utilizados para modelar la validez temporal de los hechos. Para poder comparar la validez temporal de la realidad modelada se necesita comparar sus elementos temporales, los cuales pueden incluir diferentes conjuntos de eventos en la expresión de eventos de sus *chronons*.

Para facilitar la comparación, se expanden los dos elementos para que ambos incluyan el mismo conjunto de eventos sin cambiar su validez temporal.

2.5.4.3. CASOS ESPECIALES DE UNA RELACIÓN BRANCHING

Se pueden definir casos especiales de relaciones *branching time* restringiendo la característica temporal de los datos.

a) *Relación válida en el tiempo cumplido.* Esta relación almacena datos que son verdaderos en el mundo real para un intervalo específico de tiempo válido. Si el tiempo válido corresponde al futuro, este se conocerá con eventos y grado de certeza. Los datos almacenados en esta relación corresponden al camino seguido por el tiempo válido en el *branching tree.*

b) *Posible relación válida en el tiempo bajo la expresión de eventos* ***e***: Esta relación almacena datos y su tiempo válido si la expresión de eventos es verdadera. No se conoce aún la salida del evento *e* en el mundo real pero se quiere examinar el dato en el caso que se satisfaga *e*. Los datos en esta relación corresponden al camino que conduce y que incluye el branching *e*.

c) *Relación alternativa en el tiempo válido* ***v.*** Esta relación contiene los datos y eventos asociados tal que los datos serán válidos en el tiempo *v*. En esta relación los datos están asociados con las expresiones de eventos.

d) *Relación* ***snapshot*** *cumplida.* Esta relación almacena datos que han sido válidos o serán válidos en el mundo real en un cierto tiempo válido *v*.

e) *Posible relación* ***snapshot*** *bajo la expresión de eventos* ***e*** *y tiempo válido* ***v.*** Esta relación almacena datos que pueden ser válidos en un cierto tiempo válido *v* y bajo una expresión de eventos *e*. La expresión *e* puede contener eventos cuyas salidas estén aún indefinidos (su salida es "**not yet known**").

2.5.4.4 REPRESENTACION DE UN INTERVALO BRANCHING

Sea R_b una relación *branching* con atributos (A_1,..., A_n | V_b), donde V_b es el atributo *timestamp* definido en el dominio de los *branching elements*. R_b es representada por una relación convencional R con atributos (A_1,..., A_n | V_s ,V_e, E). Los atributos V_s y V_e son valores atómicos que representan el intervalo [V_s ,V_e). E es una expresión de eventos atómica almacenada como una cadena de caracteres. Los *timestamps* corresponden a la representación de intervalos homogéneos.

El *timestamp* V_b en la relación conceptual corresponde a más de una terna < V_s ,V_e, E>. Cada < V_s ,V_e, E> corresponde a un intervalo homogéneo. Para esta representación se requiere que las tuplas se encuentren temporalmente fusionadas. Las tuplas que van a fusionarse temporalmente deben tener la misma expresión de eventos. Se asume que una relación para un intervalo *branching* siempre esta fusionada.

2.5.5. ACTUALIZACIONES

2.5.5.1. SEMANTICA DE LAS TRES FORMAS DE ACTUALIZACIÓN.

Una inserción consiste en agregar a una relación r, un hecho (a_1,..., a_n) actualmente no registrado para un conjunto de valores de tiempo válido v_v y una expresión de eventos e. Recordemos que v_b

es un conjunto de *branching chronons.*

$$insert(r,(a_1,...,a_n),e,v_v)= \left\{ \begin{array}{l} r-\{<(a,...,a_n|\ v_b),e_1>\} \cup \{<(a_1,...,a_n|\ v_b \cup v_v),e>\} \\ \qquad\qquad si \exists\, t(<(a_1...,a_n|\ v_b),e_1>) \in r) \wedge e=e_1 \\ \\ r \cup \{<(a_1,...,a_n|v_v),e>\}\ sino \end{array} \right\}$$

El borrado consiste en remover el estado $(a_{1,...,}\ a_n)$ de un objeto para un conjunto de valores de tiempo válido v_v.

$$delete(r,(a_1,...,a_n),e,v_V)= \left\{ \begin{array}{l} r-\{<(a,...,a_n|v_b),e_1>\} \cup \{<(a_1,...,a_n|v_b - v_V),e>\} \\ \qquad\qquad si \exists t(<(a_1...,a_n|v_b),e_1> \in r) \wedge e=e_1 \\ \\ r\ sino \end{array} \right\}$$

En una relación *branching time*, delete(r,$(a_1,...,a_n)$,e,v_V) implica que las tuplas seleccionadas dejen de ser corrientes para un conjunto de valores de tiempo v_V.

Los cambios en los valores de los atributos de una tupla $(a_1,\ldots,\ a_n)$ por $(b_1,\ldots,\ b_n)$ para un conjunto de valores de tiempo v_V son logrados borrando la tupla $(a_1,\ldots,\ a_n)$ primero y luego insertando la tupla $(b_1,\ldots,\ b_n)$.

$$update(r,\ (a_1,...,a_n),(b_1,...,b_n),e,v_V)=insert(delete(r,(a_1,...,a_n),e,v_V),(b_1,...,b_n),e,v_V)$$

2.5.5.1.1. EJEMPLO

Consideremos el caso de la relación EMPLEADOS. Supongamos que en febrero empleamos a Tomás desde febrero hasta junio para trabajar en el local Cabildo si se confirma la apertura del local. En abril, le extendemos el contrato hasta el mes de septiembre si se confirma la apertura del local Cabildo. Finalmente, en julio reducimos las extensión del contrato hasta agosto siempre que se confirmo la apertura del local Cabildo y lo transferimos al local La Pampa. Llamemos E_1 (evento 1) a "si se confirma la apertura del local Cabildo ".

La relación conceptual introducida es creada por la siguiente secuencia de comandos.

Comando
Insert(empleado,("Tomas","Cabildo"),E_1,[2,6])
Update(empleado,("Tomas","Cabildo"),E_1,[2,9])
Delete(empleado,("Tomas","Cabildo"),E_1,[9,forever], E_1)
Insert(empleado,("Tomas","Pampa"),E_1,[7,8])

2.6. SUMARIO

En este capítulo se examinaron el Modelo Conceptual de Datos Bitemporal y el *Branching Time Conceptual Data Model* . Para la descripción de estos modelos tuvimos en cuenta los siguientes puntos: dominio del tiempo, objetos en el modelo, semántica de equivalencia y granularidad.

Ambos modelos soportan la combinación de tiempo válido y tiempo transaccional y difieren en la representación de sus *chronons*. El *Branching Time Conceptual Data Model* incorpora los conceptos de evento, acción y árbol de eventos para poder manejar alternativas en el tiempo lo que hace que tenga un esquema de relación distinto al Modelo Conceptual de Datos Bitemporal. Las operaciones de inserción, borrado y modificación en los modelos vistos no sólo difieren en la cantidad de parámetros, sino también en el tratamiento que por la incorporación de la expresión de eventos en el BTCDM sufren las tuplas de la relación.

CAPITULO 3

EL MODELO PROPUESTO - BBTSQL2

Basado en lo expresado en los capítulos anteriores, en el presente capítulo introducimos un modelo de datos que incorpora soporte temporal a SQL-92. El modelo admite la combinación de tiempo válido y tiempo transaccional e incorpora la semántica de alternativas futuras dentro del modelo relacional [SA97]. Denominamos BBTSQL2 al lenguaje de consulta de este modelo de datos.

Nuestra implementación de BBTSQL2 es un subconjunto de TSQL2 en lo que hace a información pasada, presente y futura, soportando un solo calendario[3]. Por otro lado, incluye extensiones que le permiten el soporte a futuros posibles a través del lenguaje BTSQL2 basado en *Branching Time Conceptual Data Model* (BTCDM).

Desde el punto de vista del soporte a futuros posibles, los elementos más importantes de este modelo son los eventos y las acciones. Como mencionamos en el capítulo anterior, un evento es un suceso que produce un resultado. Por otro lado, una acción es una operación sobre la Base de Datos a ser ejecutada solo cuando cierta combinación de eventos ocurra. En el presente trabajo asumimos que la ocurrencia de los eventos es externa a la Base de Datos y por lo tanto debe ser indicada de algún modo al sistema.

3.1. EL EJEMPLO

En lo que resta de la Tesis ejemplificaremos varios conceptos basándonos en el ejemplo que presentamos a continuación.

Consideraremos el caso de una empresa que cuenta con locales distribuidos en regiones y empleados asignados a cada uno de los locales. La empresa comercializa productos cuyas ventas

[3] TSQL2 tiene soporte a múltiples sistemas de calendario, pero es el usuario el que debe incluir todas las funciones de mapeo temporal entre los distintos sistemas.

son registradas por local y vendedor. Para registrar esta información contamos con el siguiente esquema:

Una relación `EMPLEADOS` cuyos atributos son: `EmpId` , `EmpNombre`, `LocId`, `Turno` (de tipo interval con granularidad '`hour to second`'. La granularidad de la relación es '`Year To Day`'.

`LOCALES` esta formada por los siguientes atributos: `LocId`, `LocNombre`, `RegId` y su granularidad es '`YEAR TO DAY`'.

La tabla de `REGIONES` cuenta con dos atributos: `RegId` y `RegNombre`.

`PRODUCTOS` mantiene información acerca del código del producto (`ProdID`) y su nombre (`ProdNombre`).

Una tabla de `PRECIOS` mantiene las variaciones pasadas y posibles mediante los siguientes atributos: `ProdID` y `Precio`. La granularidad de esta relación es '`YEAR TO SECOND`'.

Finalmente, la relación de `VENTAS` registra: `ProdID`, `LocID`, `EmpID` y `Cantidad`. La granularidad de esta relación es '`YEAR TO SECOND`'.

Tanto `EMPLEADOS` como `LOCALES` y `PRECIOS` son tablas del tipo *Branching*. La relación `VENTAS` captura solamente el tiempo válido y el resto carece de soporte temporal.

3.2. CARACTERÍSTICAS DEL MODELO

3.2.1. TIPOS DE RELACIONES

Dado que no siempre es necesario guardar información temporal pasada, presente o futura, la implementación presentada en este trabajo permite definir tres tipos de relaciones: *Snapshot*, *Valid* y *Branching Time*, correspondientes a los distintos tipos de relaciones introducidos en el Capitulo 2.

Una relación de tipo *snapshot* corresponde a la concepción usual en el modelo relacional.

Está compuesta por atributos y valores para los mismos, organizados en lo que se conoce como n-uplas.

Una relación de tipo *valid* contiene información histórica acerca de la "vida" de las tuplas. Es decir que en todo momento se mantiene el tiempo de validez de los valores de las uplas. Esto equivale a decir que el *lifespan* se encuentra a nivel tupla. Esto añade una primera instancia de dimensión temporal a los datos.

Por último, la relación de tipo *branching time* permite la realización de consultas a futuro, es decir, alcanza una nueva dimensión ya que permiten realizar consultas a estados posibles de la Base de Datos en algún momento no ocurrido aún. Esta clase es un superconjunto de las relaciones *valid* ya que también mantiene la información histórica acerca de las tuplas.

3.2.2. OPERADORES TEMPORALES

En esta sección definimos la semántica de los tipos de datos, las operaciones temporales y constructores soportados en BBTSQL2.

Cada tipo de dato temporal tiene una semántica bien definida en la cual una fecha y/u hora (*datetime*) describe un punto dado en el tiempo con una granularidad específica, los períodos son necesarios para especificar lapsos de tiempo con comienzo y fin y los intervalos describen duraciones de tiempo sin un punto específico de comienzo. Es decir, describen lapsos de tiempo en alguna granularidad específica, pero sin anclaje en el tiempo.

Para familiarizarnos un poco más con estos tipos de datos vamos a mostrar algunos ejemplos de consultas temporales.

TIPO DE DATO	EJEMPLO DE CONSULTA
Datetime	¿Desde cuando es valida la promoción de café?
Interval	¿Hace cuanto tiempo se inauguró la sucursal Martínez?
Period	¿Trabajó Julieta en la sucursal Belgrano entre Junio y diciembre de 1999?

- **Tabla 3.1** Ejemplos de consultas temporales

3.2.2.1. FUNCIONES BUILT-IN

Existe una serie de funciones *built-in* para manipular y realizar conversiones entre los distintos tipos de datos. Así las funciones destructoras `begin` y `end` retornan el comienzo y el final de un período respectivamente. Un período se encuentra compuesto por dos puntos en el tiempo (*datetimes*), ambos especificados con idéntica granularidad. La función destructora retorna un *datetime* que tiene la granularidad de los elementos del período. Por ejemplo:

```
Begin(period[date '10/07/1999' year to day, date '25/11/2010' year to day])=
                                        date '10/07/1999' year to day

End(period[date '10/07/1999' year to day, date '25/11/2010' year to day])=
                                        date '25/11/2000' year to day
```

La función constructora `period` retorna un período limitado por dos *datetimes*, los cuales deben tener la misma granularidad. Este constructor fue utilizado en el ejemplo de uso de las funciones `begin` y `end`. La función `intersect` retorna el período representado por el solapamiento de los períodos dados. La función `interval` construye un intervalo que representa una duración de tiempo. Ejemplos de uso de estas funciones son:

```
Intersect(period[time '10' hour,time '20' hour],
          period[time '5' hour,time '17' hour])=
                                Period [time '10' hour, time '17' hour]

Interval '10:20' hour to minute
```

Funciones como `first` y `last` aceptan dos *datetimes* como argumentos y retornan el primer y último *datetime* respectivamente.

3.2.2.2. EXPRESIONES ARITMÉTICAS

Una de las principales características de una Base de Datos que ofrece soporte temporal es la posibilidad de realizar operaciones aritméticas con valores temporales como argumentos. Por ejemplo, si se desea obtener la cantidad de días trabajados por un empleado y disponemos de la fecha de ingreso.

No todas las combinaciones posibles de operaciones aritméticas con operandos cuyos tipos de datos sean temporales tienen sentido en BBTSQL2. En este apartado describimos las operaciones válidas dentro del lenguaje.

Es posible sumar o restar dos o más intervalos, por ejemplo: 10 años - 5 meses + 12 horas, lo que daría como resultado 9 años 7 meses 12 horas. Para poder realizar estas operaciones los intervalos son convertidos a días o fracciones de los mismos. Al intervalo resultante de esta operación se le asigna una nueva granularidad a partir de las granularidades de los operandos. Para el ejemplo mencionado anteriormente la granularidad de 10 es `YEAR` la de 5 es `MONTH` y la de 12 es `HOUR` , por lo que la granularidad del intervalo generado será `YEAR TO HOUR`.

Otra operación posible entre dos intervalos es la división, que a diferencia de la suma y la resta retorna un número, por ejemplo 10 años/2 años retorna 5. También es posible dividir o multiplicar un operando de tipo interval por un número, esto es 10 años/2 devuelve 5 años.

Es posible añadir o quitar intervalos a un *datetime*, por ejemplo sumar '10 de julio del 2000' más 10 días devuelve como resultado 20 de Julio del 2000. **Requerimos para esta clase de operaciones que el componente más grueso de la granularidad sea el mismo para ambos operandos**. En nuestro modelo, un cálculo como '7 de Julio' + 5 años retorna un error de granularidad.

La suma entre un operando de tipo *datetime* y otro de tipo *interval* es conmutativa, no así la resta entre ambos operandos ya que carece de sentido.

Si se desea saber cuanto tiempo transcurre entre dos fechas y/u horas o alguna combinación válida de las mismas debemos restar dos operandos de tipo *datetime*. Por ejemplo '20 de Junio del 2000' – '1 de junio del 2000' se obtiene 19 días como resultado.

La suma o resta de un período y un intervalo, agrega o quita lapsos de tiempo a cada uno de los datetimes que forman el período, esto es, si al período que comienza el '1 de agosto del 2000' y termina el '31 de diciembre del 2000', se le suma un intervalo de un año el resultado sería un nuevo período comprendido entre el 1 de agosto del 2001 y el 31 de diciembre del 2001.

La suma entre un operando de tipo `interval` y otro de tipo `period` es conmutativa, no así la resta entre ambos operandos.

Como regla general, podemos decir que sólo tienen sentido aquellas operaciones que involucran intervalos, períodos, fechas u horas (y sus combinaciones válidas) cuyos elementos comienzan con el mismo tipo de gránulos.

3.2.2.3. OPERADORES DE COMPARACIÓN

Los operadores básicos de comparación temporal "=, <, >, <=, >=, <>" utilizan *datetimes* en el calendario Gregoriano como argumentos.

El operador de comparación ">" se corresponde con preguntas del tipo: ¿Es 10 de mayo del 1996 posterior al 5 de abril del 1996?. Razonamientos similares pueden utilizarse para el resto de los operadores.

Como hemos visto anteriormente, cada fecha y/u hora tiene asociada una granularidad. Para que las comparaciones tengan sentido, se pide que el componente mas grueso de la granularidad de ambos sea el mismo, por lo que no está permitido comparar la fecha '10/5/1999' (día/mes/ /año) con '14/6' (mes/día) y sí la fecha '05' (mes) con '3/4 21:05' (mes/dia hora:minuto).

Durante una comparación válida cuyos argumentos no tienen la misma granularidad realizamos una expansión del argumento con granularidad más gruesa a la granularidad más fina para así poder realizar la comparación. El operador "<" verifica que el o los elementos del primer operando sean menores que el o los elementos del segundo operando. De la misma manera se pueden interpretar la semántica de los demás operadores. Por ejemplo, es posible comparar `DATE 20/7/2000' > DATE '6/2000' YEAR TO MONTH`. El significado de esta comparación se traduce a que la fecha 20 de julio del 2000 sea mayor que todas las fechas

comprendidas entre el 1 de junio del 2000 y el 30 de junio del 2000.

Otra posibilidad consiste en la pregunta acerca de la igualdad de períodos. El resto de las operaciones temporales básicas no están permitidas para comparar dos períodos ya que carecen de sentido (Por ejemplo, preguntar si el período comprendido entre abril de 1999 y junio de 1999 es mayor que el período comprendido entre abril del 2000 y agosto del 2000).

Es posible utilizar todos los operadores de comparación temporales básicos entre dos intervalos. Aquí realizamos una conversión de los intervalos a días o fracciones de días para luego poder realizar la operación (Ejemplo: `INTERVAL '2/2000' YEAR TO MONTH > INTERVAL '28' DAYS`).

Sean un período un intervalo compuesto por dos *datetimes* indicando su comienzo y fin respectivamente. Si p_1 y p_2 son períodos, tenemos:

p_1 **precedes** p_2 si $End(p_1) < Begin(p_2)$

$p_1 = p_2$ si $Begin(p_1) = Begin(p_2) \wedge End(p_1) = End(p_2)$

p_1 **meets** p_2 si $Begin(p_1) = End(p_2) + 1 \vee Begin(p_2) = End(p_1) + 1$ en donde "+" es el operador de suma de *datetimes* con intervalos (es decir, se suma un gránulo en la granularidad correspondiente).

p_1 **overlaps** p_2 si $\exists$ un chronon t, tal que $t \in$ al $p_1 \wedge t \in p_2$.

p_1 **contains** p_2 si $(Begin(p_2) > Begin(p_1) \vee Begin(p_2) = Begin(p_1) \wedge (End(p_2) < End(p_1) \vee End(p_2) = End(p_1))$.

Porque no x incluye los operadores de Allen?

3.2.3. CONSISTENCIA DE LAS RAMAS

Del análisis de "*Branching Time Conceptual Model*" [SA97] surge que como consecuencia de crear las ramas alternativas exactamente en el momento en el que una acción es definida, se produce un problema de inconsistencia con respecto a los datos presentes en una relación. Para explicar una situación típica, consideremos el siguiente escenario:

Se decide dar un aumento del 5% a todos los empleados a partir del año 2001 si aumentan las ventas. Creamos entonces un evento aumentan_ventas y su acción asociada (una síntesis de la sintaxis para realizar esta operación puede encontrarse en el apartado final de este capítulo):

```
CREATE EVENT aumentan_ventas "Aumentan las ventas";
CREATE ACTION FOR aumentan_ventas IS 'update sueldo set sueldo = sueldo*1.05';
```

La ejecución de la acción provocará que una nueva tupla por cada tupla existente hoy sea insertada en la relación empleados con una expresión de eventos "aumentan_ventas". Adicionalmente, las tuplas presentes serán marcadas con la expresión de eventos "**¬aumentan_ventas**" constituyéndose así, en la rama alternativa. A continuación, decidimos darle un aumento del 10% a un empleado x. Debido a que la acción ya fue ejecutada, este aumento reciente no se verá reflejado en la rama alternativa futura.

Este problema tiene graves efectos en el resultado de las consultas que hagan uso del evento "**aumentan_ventas**". Supongamos que contamos con tres empleados: José, Juan y Pedro con sueldos de 100, 200 y 300 pesos respectivamente. Al crear la acción mencionada, la rama alternativa correspondiente a la salida afirmativa del evento "**aumentan_ventas**" contendrá tres tuplas. Por simplicidad, consideraremos que la relación `EMPLEADOS` consta solo de dos atributos explícitos: `nombre` y `sueldo`. Entonces, la rama alternativa correspondiente a "**aumentan_ventas**" de la tabla de empleados se vería como:

Nombre	Sueldo
Jose	105
Juan	210
Pedro	315

Si ahora procedemos a otorgarle un aumento del 10% a José, el nuevo sueldo sería de $115,50. Una consulta por la rama futura que contiene al evento "aumentan_ventas" mostraría que José tendría un sueldo de $105 si la salida del evento fuese verdadera, lo que es inconsistente. **El resultado correcto es que José tendría un sueldo de $121,27**.

Esta deficiencia en el modelo quita en gran medida la utilidad y el poder que ofrece, dado que no podrían realizarse cambios a los datos presentes una vez creadas acciones que se basan en valores presentes, algo que es sumamente frecuente. Existe la posibilidad de hacer efectivas

No entiendo el problema como una deficiencia del modelo sino como una complicación de las transiciones.

las acciones ante una consulta que haga uso de un evento en particular, pero desde ya esta solución es sumamente costosa sobre todo ante la posibilidad de consultas simultáneas para el mismo evento.

Nuestra solución a este problema consiste en regenerar todas las tuplas provenientes de acciones ante la solicitud del usuario. En el caso del ejemplo, una vez que se ha determinado un aumento para un empleado con posterioridad a la creación de una acción, el usuario podría solicitar la consistencia de las ramas futuras. De no realizar esta consistencia, el problema mencionado sería visible. El algoritmo que realiza la regeneración de las ramas alternativas puede encontrarse en el capítulo 5 sección 6.

Resulta de interés destacar que existen dos formas de crear n-uplas *branching*: una es que como resultado de un evento se dispare una acción que realice la inserción correspondiente y una segunda posibilidad es la de realizar una inserción *branching*. A pesar de que el lenguaje permite la posibilidad de generar alternativas *branching* explícitamente, la forma más natural de hacerlo es a través de la especificación de eventos y sus acciones asociadas.

3.2.4. NIVELES DE EVALUACIÓN DE EXPRESIONES

En el modelo introducido por Sarda se permite la definición de eventos a partir de otros eventos previos. Por ejemplo, si $e_3 = e_1 \vee e_2$, al momento de la evaluación, pueden existir tuplas con cualquier combinación factible entre los dos eventos en función de los cuales se encuentra definido e_3. La evaluación requiere de un primer paso de reescritura de todas las expresiones obtenidas desde la Base de Datos. Una vez hecho esto, se puede proceder a evaluar la expresión resultante.

En el momento de la evaluación, podríamos encontrar expresiones como "$\neg e_1 \vee \neg e_2$" teniendo que considerar a esta fila como verdadera, dado que el requisito es que "e_1" o "e_2" sea verdadero. Si únicamente uno de ellos es verdadero, la expresión "$\neg e_1 \vee \neg e_2$" también lo sería. Pero también serían ciertas expresiones compuestas solamente por "e_1" o por "e_2". En esta situación tendríamos como resultado una relación inconsistente, dado que si encontramos tuplas con expresiones de eventos "e_1" y otras con "e_2", una tupla con expresión de eventos "$\neg e_1 \vee \neg e_2$" sería inconsistente.

Evitamos esta clase de posibilidades, restringiendo los tipos de expresiones de eventos sólo a conjunciones. Por otra parte, el lenguaje no permite la definición de eventos a partir de otros ya existentes.

3.2.5. DEPENDENCIAS ENTRE EVENTOS

De acuerdo a lo expresado en el apartado anterior, no es factible implementar la posibilidad de ocurrencia de un evento que dependa de otro.

3.2.6. SEÑALIZACION DE EVENTOS

Hemos mencionado que la definición de eventos y acciones permiten a la Base de Datos realizar consultas a diferentes futuros alternativos. Es necesario entonces definir la forma de indicar a la Base de Datos que un evento pasa a ser cierto o falso y por lo tanto su estado deja de ser "**not yet known**". Una serie de acciones deben ser llevadas a cabo cuando un usuario indica que un evento ha ocurrido en la realidad: en principio toda la información contenida en las ramas futuras debe ser "movida" al presente.

BTSQL2 permite la señalización de un evento manteniendo la información ramificada de los futuros que ya no son alcanzables debido a que el caso hipotético se ha convertido en un hecho y por lo tanto la rama positiva o la negativa (dependiendo de la salida del evento) dejarán de ser alcanzables en el mundo real.

En nuestro modelo, la indicación de que el evento se ha cumplido elimina la ramas que involucran la no ocurrencia del evento si la salida del evento es verdadera (lo mismo ocurre en el caso contrario cuando la salida es falsa). BTSQL2 posee una sentencia especifica (PURGE) para la eliminación de estas ramas inalcanzables para un evento dado. Nuestro lenguaje carece de dicha sentencia debido a que es la señalización del evento misma es la encargada de dicha eliminación.

3.2.7. GRANULARIDAD

En nuestra implementación, consideramos un subconjunto del conjunto de todas las granularidades posibles. El conjunto considerado incluye las siguientes granularidades: Años,

meses, días, horas, minutos y segundos, y cualquier combinación válida de estas. Obviamente existirán combinaciones que si bien pueden darse, no tienen sentido como tales. La lista completa de granularidades validas se muestra a continuación.

MONTH TO MONTH	YEAR TO MONTH
DAY TO DAY	YEAR TO DAY
HOUR TO HOUR	YEAR TO HOUR
MINUTE TO MINUTE	YEAR TO MINUTE
SECOND TO SECOND	YEAR TO SECOND
MONTH TO DAY	MONTH TO HOUR
YEAR TO YEAR	MONTH TO MINUTE
MONTH TO SECOND	DAY TO SECOND
DAY TO MINUTE	DAY TO HOUR
HOUR TO MINUTE	HOUR TO SECOND
MINUTE TO SECOND	

- **Tabla 3.2** Granularidades soportadas por el Modelo

Debido a que nuestro conjunto de granularidades es reducido, la implementación de la función cast es bastante más simple que la presentada por Dyreson y Snodgrass [SNO95]. La simplificación del algoritmo radica en que nuestro grafo de granularidades carece de V-Paths. Las conversiones son siempre lineales. A pesar de esto, se mantiene el concepto de orden parcial, por ejemplo: '01/1999' en granularidad `YEAR TO MONTH` no es comparable a 12 en granularidad `MINUTE TO MINUTE`.

3.2.8. LA OPERACIÓN *COALESCE*

Como mencionamos en el Capítulo 2, el *coalesce* o fusión temporal es una operación aplicable a relaciones temporales. En particular, los modelos de Snodgrass y Sarda no admiten la posibilidad de obtener un resultado no fusionado. Sin embargo, en ocasiones, el usuario puede contar con información que le permite saber que el resultado no requiere ninguna clase de fusión temporal pudiendo eliminar el costo de la operación *coalesce*. Para aprovechar esta característica hemos expandido la sintaxis del lenguaje incluyendo una cláusula para instruir al estrato medio a que no realice ninguna fusión sobre el conjunto obtenido. Esta cláusula es **`NOCOALESCE`** y debe ser la última de una sentencia de selección. Una síntesis de la sintaxis de la sentencia **`SELECT`** puede encontrarse al final de este capítulo.

Esta posibilidad no está disponible para las demás operaciones como inserciones, actualizaciones o borrados debido a que el modelo no soporta relaciones no fusionadas.

3.2.9. TABLAS DE EVENTOS

En el lenguaje HSQL [SA90] se incorpora un tipo de tabla *event* que permite registrar el instante de ocurrencia de eventos[4] en forma automática. En nuestro modelo, este tipo de tabla carece de sentido gracias a su bitemporalidad implícita, es decir, que el tiempo válido y transaccional son mantenidos en todo momento.

3.2.10. CUADRO COMPARATIVO

A modo de resumen, incluimos aquí un cuadro que establece las diferencias y características comunes entre los modelos en los que hemos basado el trabajo presentado en esta Tesis.

[4] No se deben confundir los eventos dentro del modelo de HSQL con los eventos presentados en el Capítulo 2. En HSQL los eventos son simplemente datos cuyo tiempo de ocurrencia, es decir, cuando son informados a la base de datos, es automáticamente registrado por la BD.

MODELO DE DATOS	TSQL2	HSQL	BTSQL2	BBTSQL2
Dimensión de tiempo	*Valid* & *Transaction*	*Valid*	*Valid* & *Transaction*	*Valid* & *Transaction*
Soporte Caléndrico	Múltiples	Unico	Unico	Unico
Coalesce automático	✓	×	✓	✓
Consistencia de ramas	❶	❶	×	✓
Branching (Tiempo futuro)	×	×	✓	✓
Historia de cambios	×	×	×	✓
Tablas de eventos	❷	✓	×	×
Dependencia entre eventos	❸	❸	✓	×
Manejo de Probabilidades en eventos	✓	×	✓	×
Posibilidad de obtener un conjunto no fusionado	×	✓	×	✓

- **Tabla 3.3** Comparación entre los distintos modelos.

3.3. CARACTERISTICAS DEL LENGUAJE

En este apartado damos una síntesis de las construcciones más comunes dentro del lenguaje que soporta el modelo implementado en este trabajo y que se utilizan en los capítulos siguientes. A modo de convención, utilizamos letras mayúsculas y **negritas** para indicar las palabras reservadas del lenguaje y "<>" para indicar listas. Usamos "[]" para mostrar que una cláusula o construcción es opcional.

❶ La consistencia de las ramas alternativas es una necesidad del modelo que es capaz de mantener información futura y probable. Debido a que estos modelos no mantienen esta clase de información, esto no es necesario.

❷ TSQL no requiere de tablas que registren el momento en el que un "evento" ocurre debido a que el manejo implícito del tiempo transaccional realiza una tarea semejante.

❸ La dependencia entre eventos o también la reescritura de eventos, es decir, la posibilidad de definir eventos a partir de la ocurrencia de otros no es posible en HSQL o TSQL2 dado que carecen de la posibilidad de crear eventos.

3.3.1. CREACIÓN DE TABLAS

Para crear una tabla se utiliza la siguiente sintaxis:

```
CREATE TABLE nombre_tabla ( < lista_campos> )
[AS VALID | AS BRANCHING TIME] [granularidad]
```

En donde:

```
<lista_campos>: < nombre_de_campo tipo [NOT NULL] [PRIMARY KEY] >
```

3.3.2. SELECCIÓN

La sintaxis de la sentencia SELECT tiene la siguiente forma:

```
SELECT <lista_columnas> [,VALID(tabla)][,TRANSACTION(tabla)
FROM <lista_tablas>
[WHERE <lista_predicados>]
[GROUP BY <lista_columnas>
[HAVING <lista_predicados_de_grupo>]]
[BRANCHING <lista_eventos>]
[NOCOALESCE];
```

En donde:

`<lista_columnas>`: columnas pertenecientes a las tablas especificadas en `<lista_tablas>`.

`<lista_tablas>`: lista de una o mas tablas.

`<lista_predicados>`: uno o mas predicados conectados por `AND` y `OR`.

`<lista_predicados_de_grupo>`: idem lista_predicados con funciones de grupo (`AVG`, `MAX`, `COUNT`, etc.)

`<lista_eventos>`: una lista de uno o mas eventos o su negación.

Como ya se explicó, la cláusula `NOCOALESCE` tiene el efecto de pedir que el resultado de una consulta no se encuentre temporalmente fusionado.

3.3.3. INSERCIÓN, MODIFICACIÓN Y BORRADO

Las sentencias insert, update y delete tienen la siguiente sintaxis:

```
INSERT INTO nombre_tabla (<lista_campos>) VALUES (<lista_valores>)
[VALID periodo_validez]
[BRANCHING expresion_eventos]


UPDATE nombre_tabla SET ( <lista_asignaciones> )
[VALID periodo_validez]
[BRANCHING expression_de_eventos]
[WHERE <lista_predicados>]


DELETE FROM nombre_tabla
[VALID periodo_validez]
[BRANCHING expression_de_eventos]
[WHERE <lista_predicados>]
```

3.3.4. CREACION DE EVENTOS Y ACCIONES

Para definir eventos y sus acciones asociadas existen dos sentencias. La primera define un evento:

```
CREATE EVENT <nombre_evento> <comentario>;
```

Para asociar una acción a un evento, la sentencia es:

```
CREATE ACTION FOR <nombre_evento> IS <sentencia_BBTSQL2>;
```

3.3.5. SEÑALIZACIÓN DE EVENTOS

Como mencionamos en el apartado 3.4, la operación de señalización, permite indicar que un evento ha ocurrido o ya no ocurrirá en el mundo real. El comando para hacer efectiva la salida de un evento es:

```
SIGNAL <nombre_evento>=<valor_de_verdad>;
```

En donde:

<valor_de_verdad>: T o F.

En los capítulos siguientes, se muestran ejemplos que hacen uso de esta sintaxis para mostrar el uso y los efectos de las distintas cláusulas.

3.4. SUMARIO

En este capítulo hemos presentado el ejemplo que utilizaremos en lo que resta de la Tesis y mostramos las principales características del modelo de datos definido y del lenguaje de consulta BBTSQL2.

Se presentaron aquellos tipos de datos y funciones nativas del lenguaje que facilitan el manejo de la dimensión temporal. La semántica de los operadores de comparación como también de las funciones temporales también fueron introducidos.

Hemos realizado una breve comparación entre los distintos modelos de Bases de Datos temporales a fin de poder establecer importantes diferencias en lo que hace a los distintos enfoques con los que cada modelo aborda la temporalidad de los datos. También puede apreciarse el caracter integrador que tiene el trabajo presente.

Finalmente, hemos introducido una sintaxis reducida a fin de poder facilitar los ejemplos que se incorporan en los capítulos siguientes.

CAPITULO 4

ARQUITECTURA

La implementación del motor presentado en este trabajo, se ha realizado siguiendo el modelo de estratos. La idea, como se mencionó en la introducción, es la de superimponer al motor de Bases de Datos relacionales, un nuevo nivel encargado de proveer el soporte temporal a la base subyacente. En este capitulo, presentamos la arquitectura adoptada junto con una serie de criterios y técnicas que han sido objeto de estudios previos y que condujeron a nuestra elección.

4.1. CRITERIOS PARA LA IMPLEMENTACIÓN

Una de las principales virtudes de los lenguajes temporales es la de derivar a la Base de Datos la complejidad que representa el manejo de datos variantes en el tiempo. Normalmente, cuando se piensa en esta clase de desarrollos se consideran *arquitecturas integradas*, es decir que toda la implementación de las operaciones temporales expresables en el lenguaje correspondiente se encuentran integradas en un único nivel. Aquí, la Base de Datos contiene toda la lógica para soportar los datos temporales. Una alternativa a esta clase de arquitectura, es la de *estratos*, en la cual la lógica temporal se encuentra en un nivel intermedio que encapsula al nivel inferior representado por un DBMS relacional. Es de esperar que una implementación integrada exhiba una performance no inferior a su equivalente de estratos.

Snodgrass y otros [TR5] definen ocho criterios para la implementación de un modelo temporal utilizando estratos. Estos son:

- **No requerir modificaciones al DBMS subyacente**.
- **Impacto mínimo en el middleware**.

- **Independencia de las aplicaciones**: el estrato debe encapsular al DBMS para todas las aplicaciones.

- **Máximo reuso de la tecnología existente**: el estrato no debe reimplementar operaciones que ya son provistas por el DBMS.

- **Disponibilidad gradual del la funcionalidad temporal**

- **Retención de las propiedades del DBMS subyacente**.

- **Performance adecuada**: las sentencias que no hacen uso de operaciones temporales deben exhibir una performance igual a la que tenían cuando no existía el soporte temporal. Además, las sentencias que se ejecuten a través del nuevo estrato deben ser tan rápidas como la sentencia equivalente a la generada por el estrato lo sería si se ejecutase directamente sobre el DBMS.

- **Independencia del DBMS**: las técnicas utilizadas para la implementación del estrato no deben ser específicas de un DBMS en particular.

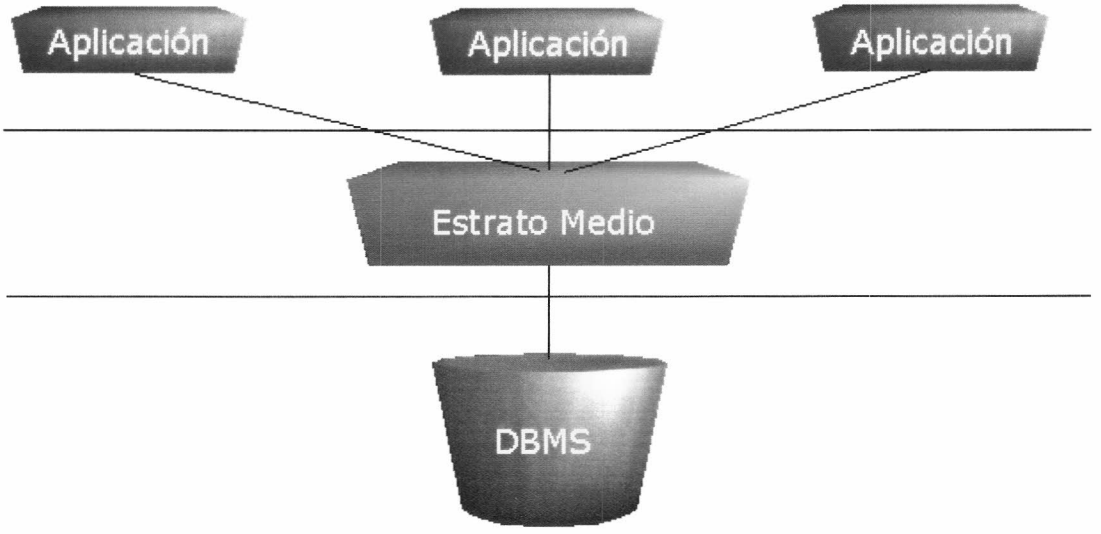
Varios de los criterios enunciados resultan conflictivos entre sí y se deben adoptar soluciones de compromiso, particularmente entre, performance, reuso de tecnología y orientación a una Base de Datos en particular.

En varios puntos del diseño es posible que una solución pueda ser menos portable, orientando la misma a cierto DBMS en particular para obtener una mejor performance de lo que se lograría si se implementase una solución genérica dentro del estrato mismo. Por otra parte, en algunos casos la implementación de ciertas operaciones dentro del DBMS mismo puede aumentar el reuso de la tecnología provista.

4.2. LA ARQUITECTURA EN NIVELES

La función del nuevo nivel es la de proveer el soporte temporal del que el DBMS subyacente no es capaz. Para ello, este nuevo nivel debe intermediar en la comunicación entre una aplicación y el DBMS para inspeccionar las sentencias que van a ejecutarse y convertirlas a

SQL-92. Gráficamente, podríamos verlo de la siguiente forma:



- **Figura 4.1**: Esquema de la arquitectura mediante el uso de estratos o niveles intermedios.

El Estrato Medio requiere de metadatos para realizar las conversiones apropiadas: información sobre las tablas temporales, nuevos tipos de columnas, etc. Accede a esta información durante la traducción, pero no interviene en la ejecución de la sentencia una vez que esta ha sido enviada al DBMS, funcionando como un típico *Mediator* [WIE92].

Si las solicitudes involucran operaciones temporales, el nuevo nivel intervendrá en la respuesta para completarla, llevando a cabo las tareas de las que no es capaz el DBMS, por ejemplo, la presentación de las mismas o tareas más complejas como la fusión temporal conocida como *coalesce*. El estrato agregado debe utilizar a la Base de Datos como una caja negra y la interacción entre ambos debe ser la menor posible a la hora de ejecutar sentencias de usuarios.

4.3. METAS GENERALES DE DISEÑO

A la hora de extender un lenguaje/modelo para añadir funcionalidades temporales, debe tenerse en cuenta cómo estas extensiones pueden afectar a las aplicaciones que no hacen uso de ellas. Estas aplicaciones se conocen como *legacy code* y debe garantizarse que estas puedan seguir haciendo uso del modelo sin ser afectadas por las nuevas posibilidades que ofrece la Base de Datos.

4.3.1. COMPATIBILIDAD ASCENDENTE

En primer lugar para que un lenguaje temporal sea aplicable en la práctica, debe poder hacer frente al desafío que presentan las aplicaciones ya existentes escritas sobre algún lenguaje no temporal. Esto es, el código existente no debería tener que ser modificado cuando se añade el soporte temporal (*upward compatibility*).

La arquitectura de estratos permite explotar características y funcionalidades provistas por el DBMS subyacente (por ejemplo: manejo de transacciones, control de concurrencia, etc.) y adicionalmente, se puede lograr la compatibilidad ascendente a un costo bastante bajo. Por otra parte debemos tener en cuenta que esta clase de implementaciones requiere de una conexión a la Base de Datos y el nivel intermedio no puede acceder a las estructuras internas del DBMS lo cual, por supuesto, afecta la implementación.

Existen dos requisitos básicos para que un nuevo modelo sea ascendentemente compatible con respecto a otro. El primero requiere que la estructura de datos que extiende al modelo sea un superconjunto de la estructura utilizada por el anterior. En segundo lugar, se requiere que una sentencia válida en el viejo modelo, también lo sea en el nuevo. Adicionalmente, la semántica también debe ser la misma, es decir, el resultado debe ser idéntico en ambos casos.

La Compatibilidad Temporal Ascendente (*Temporal Upward Compatibility*, en adelante TUC) [TR2] es un requerimiento mucho más restrictivo. Para que un nuevo modelo sea TUC con respecto a otro, se pide que todos los sistemas existentes previamente a la existencia del nuevo modelo, trabajen sin ningún cambio aunque las tablas que eran usadas por ellos cambien para

proveer soporte para los tiempos de transacción o válido. Esta propiedad garantiza que los viejos y nuevos sistemas pueden coexistir sin problemas.

4.3.2. INDEPENDENCIA DE LA PLATAFORMA

Es también destacable la importancia que tiene que el nuevo nivel que aporta la funcionalidad temporal sea independiente del DBMS. Si el nuevo nivel genera SQL a partir de una sentencia temporal, la portabilidad del sistema aumenta en gran medida, debiendo realizar solo cambios al módulo encargado de la comunicación directa con la Base de Datos.

4.3.3. DOMINIO TEMPORAL

La inclusión del soporte temporal implica la extensión de las relaciones que desean hacer uso de esta posibilidad con marcas de tiempo o *timestamps*. La elección del dominio temporal a utilizar para registrar estas marcas temporales es el tema de discusión de este apartado.

La primera posibilidad es la de utilizar el tipo `DATE` o similar provisto por el DBMS subyacente. Esta opción ofrece varias ventajas, entre ellas la de que el DBMS provee las funciones para operar con este dominio, evitando de este forma su reimplementación dentro del estrato. Por otra parte, tiene la desventaja de la limitación del rango de fechas soportado por el DBMS (los años se encuentran en el rango 1-9999).

Otra alternativa sería la de implementar los *timestamps* como atributos numéricos. Si bien esta posibilidad no tiene la restricción de los atributos de tipo `DATE`, obliga a la reimplementación de todas las operaciones necesarias para su manejo dentro del nivel intermedio. Esto atenta en cierta forma contra el criterio de reuso de tecnología existente y en segundo término, aumenta la complejidad del nivel intermedio.

Finalmente, se puede hacer uso de la posibilidad que ofrecen ciertos DBMS de crear tipos abstractos de datos u objetos. Esto reduciría la portabilidad al sistema, ya que no todas las Bases de Datos poseen esta característica.

Para nuestra implementación utilizamos dos tipos de marcas temporales a) los atributos implícitos que soportan el tiempo válido y el de transacción están basados en el tipo `DATE`

ofrecido por ORACLE; b) dos nuevos tipos de datos `INTERVAL` y `PERIOD` son agregados por el estrato. El primero está basado en un atributo numérico y permite almacenar intervalos de tiempo en cualquiera de las granularidades ofrecidas por el estrato (Por ejemplo: 3 días y 4 horas). La unidad con la que se almacena este tipo es de "`día`", realizando las conversiones necesarias para la presentación al usuario. El tipo `PERIOD`, que se presenta al usuario como una unidad, está basado en dos atributos de tipo `DATE` (Por ejemplo: [12 de marzo de 1999, 23 de octubre de 2000)).

4.3.4. ELECCIÓN DE LA REPRESENTACIÓN DEL VALOR "*FOREVER*"

Las Bases de Datos temporales hacen uso intensivo de un valor conocido como ***forever***. Este representa el tiempo corriente y en la literatura recibe otros nombres como *now* o *Until Change* (utilizado en Capitulo 2 apartado 2.1.3.1). Un hecho que es corriente para la Base de Datos (tiempo de transacción), estará definido desde algún tiempo *t* hasta el momento actual y se representa con el período *[t,* ***forever****)*.

La elección del valor más apropiado para la representación de "**forever**" depende del dominio seleccionado para la representación de los *timestamps*. En nuestro caso, habiendo decidido que estas marcas temporales están representadas por valores del tipo `DATE`, solo quedan tres posibilidades: el valor mínimo del rango, el máximo o `NULL`.

Torp, Jensen y Böhlen [TR2] han estudiado empíricamente cual de estos valores es el que ofrece más ventajas. Algunas características a considerar son:

a) si se selecciona el máximo o el mínimo valor representable entonces existirá una potencial sobrecarga y esta debe ser tratada por el estrato.

b) La elección de NULL para la representación de *forever* impediría la posibilidad de la utilización de índices sobre los *timestamps*. Esto se convierte en una desventaja importante a la hora de mejorar la performance de las operaciones que hacen uso de los atributos implícitos.

El mencionado trabajo campara tres consultas temporales sobre distintos *timeslices*. Las consultas favorecían el estado presente de la base debido a que es supuesto que esta es la

sección de los datos almacenados que se accede con mayor frecuencia en la operatoria normal. Los resultados más relevantes fueron: a) *el valor mínimo del rango no debe utilizarse dado que ofrece la performance más pobre de las tres posibilidades*; b) *no se puede afirmar si es mejor utilizar* `NULL` *o el máximo valor*. Un criterio planteado para la selección del valor a utilizar es que si cerca de un 10% de la información almacenada es corriente, es conveniente el uso del máximo valor y `NULL` si este porcentaje asciende a 40% o más. Esto ocurre porque recorrer una tabla mediante el uso de un índice requiere de dos accesos por cada lectura: uno al índice y otro a la tabla. En cambio, un barrido de una tabla puede requerir menos lecturas ya que puede realizarse con otras técnicas, como por ejemplo, lecturas multibloque.

Suponemos que a medida que el tiempo avanza, la cantidad de datos históricos es la que aumenta y la cantidad de datos corrientes aumenta en menor relación. Por esa razón, **seleccionamos al máximo valor del rango como la representación de "forever"**. Esto es, en nuestro caso particular, el '31/12/9999 23:59:59'.

4.4. PROCESAMIENTO DE UNA CONSULTA

El camino recorrido por una sentencia tiene tres partes:

1) La primer sección es la del envío de la sentencia al estrato medio por parte de la aplicación.

2) Luego de la traducción a SQL, la nueva sentencia debe ser enviada al DBMS subyacente.

3) El último paso ocurre cuando el estrato medio recupera la respuesta generada por el DBMS. Eventualmente, puede realizar algunas operaciones adicionales y finalmente esta respuesta es enviada a la aplicación solicitante.

- **Figura 4.2**: Pasos en la ejecución de una sentencia temporal en BTSQL2.

Resulta claro que uno de los componentes más importantes del estrato medio es el traductor que debe encargarse del análisis sintáctico y por lo tanto, del chequeo de los tipos temporales, de las sentencias enviadas por las aplicaciones, generando la consulta equivalente en SQL. Independientemente de las distintas formas que pueda adoptar el estrato medio [TR5], existen varias estrategias para crear un un analizador sintáctico.

4.4.1. ANALIZADOR SINTACTICO

Siendo este un componente tan importante dentro de la arquitectura, nos extenderemos a efectos de explicar las opciones de diseño disponibles para su implementación y justificar la elección realizada.

La primera opción es la de un *parser* completo, capaz de procesar cualquier sentencia del lenguaje. Aquí, una sentencia es enviada al estrato, este analiza la sentencia accediendo a la información que mantiene sobre los datos temporales (un pequeño diccionario que guarda información acerca de las columnas y tipos que no brinda el DBMS). Una vez efectuada la conversión, la nueva sentencia es enviada al DBMS. Finalmente, la respuesta es interceptada para hacer las transformaciones necesarias, que veremos con mayor detalle más adelante. Esta arquitectura es conocida como *parser completo* y ofrece la posibilidad de alcanzar la compatibilidad ascendente y temporal. Otra ventaja es la de encapsular totalmente el funcionamiento. Como desventaja encontramos que no se está haciendo uso de la tecnología que ofrece el DBMS ya que este provee un *parser* de SQL.

Otra posibilidad es la de un *parser parcial.* En este caso, cuando el estrato obtiene una sentencia para ser procesada, trata de analizarla. Si falla, entonces la sentencia se envía directamente a la Base de Datos. Si ocurre un nuevo error, entonces existe un error en la sentencia. El problema aquí es que no es fácil distinguir a qué clase de error responde el problema. Supongamos que una sentencia temporal está sintácticamente mal formada, provocando que la misma sea enviada al DBMS sin una conversión. Claramente, el DBMS subyacente tampoco será capaz de analizar la sentencia en cuestión lo que provocará un segundo error de sintaxis quedando a cargo del estrato distinguir cual es el error exactamente. Otro

problema, es que enviando una sentencia válida en SQL, el estrato la pasará directamente al DBMS provocando una potencial exposición de los atributos implícitos que soportan los tiempos válidos y de transacción.

Finalmente, consideramos la posibilidad de un *parser* que incluya alguna clase de palabra clave que permita distinguir sentencias temporales de aquellas que no los son. Aquí, un preprocesador de sentencias descubre de qué tipo es, llevando a cabo el procesamiento correspondiente o enviando la sentencia directamente al DBMS. El problema con este tipo de arquitectura es que en *legacy systems* una sentencia no temporal sería doblemente procesada, ya que el preprocesador asume que la sentencia es temporal al no encontrar un comentario indicando lo contrario. Una variante de esta arquitectura es que las sentencias temporales tengan obligatoriamente una palabra reservada a modo de comentario que permita distinguirlas. La gran desventaja que presenta este método es que no se puede alcanzar la compatibilidad temporal ascendente. Cuando una tabla es extendida para agregar el soporte temporal, todas las sentencias que hagan uso del mismo deberán agregar la palabra reservada, introduciendo modificaciones en el código existente.

Otras formas posibles incluyen la de "enmascarar" las funciones ofrecidas por APIs (Application Program Interface) como ODBC (Open DataBase Connectivity). Aquí cada función es encapsulada por otra que hace la conversión cuando es requerido. Si bien, esta es una forma bastante simple de hacerlo, requeriría de un *Wrapper*[5] [GA95] para cada tipo de API que utilicen los *legacy systems* para acceder a la Base de Datos. Si esto no se hiciera así, las aplicaciones que no hicieran uso de la nueva API que encapsula la anterior funcionalidad, estarán posibilitadas de acceder a los atributos implícitos quedando expuestos detalles de implementación que no deben ser visibles por los usuarios.

Nos hemos inclinado por una implementación de *parser completo* que si bien demanda mucho más trabajo en la construcción del estrato permite más flexibilidad para la implementación del

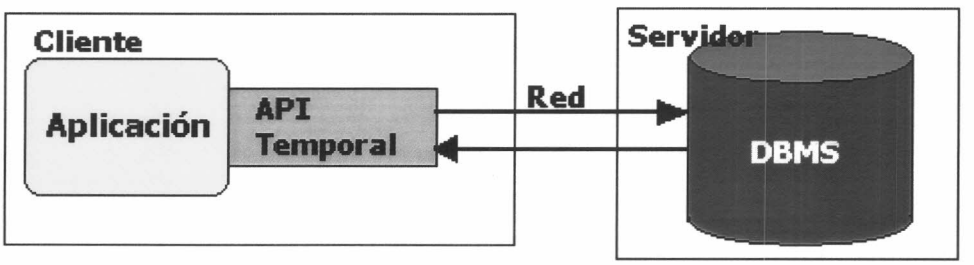
[5] Wrapper: una técnica de diseño que permite cambiar una interface de un objeto en otra que espera el cliente que hace uso de la misma. Adicionalmente, la técnica de Decorator, permite agregar nueva funcionalidad a un objeto. Ambas se utilizan para extender la funcionalidad de un objeto.

lenguaje y mayor manejo y control de errores. Toda la funcionalidad temporal, se encuentra encapsulada dentro del nivel intermedio.

4.5. SERVIDOR

Hasta el momento, hemos mencionado la existencia de un nivel intermedio encargado de encapsular al DBMS aportando la funcionalidad temporal, pero nada ha sido dicho acerca de la disposición de los componentes. Diferentes alternativas existen para la implementación del traductor de sentencias temporales a SQL. A continuación, resumimos estas alternativas y explicamos nuestra decisión.

Uno de los diseños más simples consiste en la escritura de una API que a su vez hará uso de alguna librería específica para realizar la conexión al DBMS subyacente. Una desventaja de este diseño es que existen operaciones que requieren un intercambio de información entre el nivel intermedio integrado al cliente y el DBMS. El tiempo de 'round trip' de los datos agravará la performance del estrato y por otra parte, el nivel intermedio podría recibir información de la cual no hará uso. Para ejemplificar esto, tomemos una situación como la que se presenta durante la evaluación de eventos en una consulta. Es el estrato el encargado de determinar si una tupla pertenece o no al resultado. Si fuera la API integrada al cliente la que realizara esta tarea, solo un conjunto reducido de las filas recibidas debe ser entregada a la aplicación. Toda esta información debe ser transferida a través de la red para luego ser descartada en gran parte. Un segundo problema que ofrece este tipo de solución es que los requerimientos de la aplicación del usuario pueden aumentar notablemente debido a la memoria destinada a *cache* para realizar esta clase de operaciones.



- **Figura 4.3**: Arquitectura en donde la API que utiliza el cliente contiene toda la lógica

temporal.

Otra alternativa es la de crear una API que se conecte con una aplicación que está en constante ejecución sobre el servidor en donde se encuentra la Base de Datos o en un servidor de aplicaciones (constituyendo una arquitectura de tres capas o '*three tier*'), convirtiéndose en un servidor de consultas temporales. Si bien, este diseño es más difícil de implementar, tiene la ventaja de reducir en tráfico de información entre el cliente y el servidor. El cliente nunca requiere datos de los que no hará uso explícitamente. En segundo lugar, las operaciones como el *coalesce* son beneficiadas dado que los datos permanecen dentro del mismo servidor y no existe tráfico de red. Finalmente, este diseño provee un mejor aprovechamiento de los recursos dado que es el servidor es el que realiza la mayor parte de las operaciones mientras la API utilizada por la aplicación permanece simple. Consideramos que este tipo de estrategia puede servir como esqueleto y base para una arquitectura integrada. Creemos además que se trata de una arquitectura mucho más flexible debido a que cambios realizados en la implementación y nuevas funcionalidades estarán inmediatamente disponibles para todas las aplicaciones que hacen uso del servidor. Por estas razones mencionadas, hemos optado por la creación de un servidor de consultas temporales y no utilizar el esquema de una API que contenga toda la lógica temporal.

4.6. SUMARIO

Hemos visto los diferentes tipos de arquitecturas existentes para la construcción de un estrato intermedio encargado de la traducción de sentencias BTSQL a SQL. Cada una ofrece ventajas y desventajas que deben ser evaluadas y analizadas durante la etapa de diseño tanto en lo que hace a sus capacidades como a su complejidad a la hora de la implementación. La decisión que hemos tomado para la construcción del estrato es a nuestro criterio, la que ofrece mayor flexibilidad en cuanto a la incorporación de nuevas posibilidades, ya que estas no deben ser redistribuidas por todas las estaciones de trabajo de los clientes como ocurriría en el caso de implementarlo en una API que contenga toda la lógica temporal. El estrato intermedio separado de los clientes, permite la incorporación de nuevas funcionalidades de manera transparente.

También, hemos explicado nuestra elección en lo que hace al análisis sintáctico de las sentencias: el *parser* completo. Este debe poseer la capacidad de analizar completamente el

lenguaje ya que es el que debe detectar cualquier error que pueda presentarse durante el análisis e informarlo al usuario. De esta forma, la escritura del *parser* se hace más compleja, pero permite un mayor control.

CAPITULO 5

DISEÑO E IMPLEMENTACIÓN

El estrato cuenta con objetos que colaboran en la traducción de sentencias como en las tareas administrativas que hacen al manejo de las conexiones de los clientes. Además, mantiene un diccionario que le permite conservar información acerca de los distintos tipos de datos y tablas que no son soportados por el DBMS subyacente. En este capítulo describimos los principales componentes, su función y detalles de implementación.

5.1. PREMISAS DE DISEÑO DEL SERVIDOR

Para el diseño del servidor propusimos las siguientes premisas:

- El servidor podrá atender a más de un cliente simultáneamente.
- Respetar el concepto de *sesión*, es decir, que los clientes una vez que realizan una operación mantienen su conexión activa hasta que se solicite la desconexión explícitamente.
- El motor tendrá un diseño portable para poder trabajar sobre distintos DBMSs existentes en el mercado.
- El diseño e implementación estarán basados en componentes intercambiables.

Estas características básicas requirieron del diseño y construcción de distintos componentes con responsabilidades bien definidas. Estos componentes son accesibles a través de una interface simple haciendo que la implementación de cada uno de estos sea intercambiable sin afectar al resto del sistema.

Debido a que se decidió utilizar el concepto de sesión, se hizo también necesaria la definición de objetos que cumplen con las tareas administrativas relacionadas con el mantenimiento de la información asociada a las conexiones activas. Si bien estos objetos no

realizan tareas explícitamente relacionadas con el manejo de información temporal, cumplen un rol fundamental en el correcto funcionamiento del motor construido.

En lo que hace a la utilización de información temporal desde programas escritos en lenguaje de alto nivel como Pascal, C/C++, Visual Basic, etc., se definió una interface consistente en funciones que cubren el manejo de la conexión desde el cliente en donde son utilizadas, la ejecución de comandos BBTSQL2 y la definición de *buffers* en donde se mantendrán los resultados de las consultas.

La factibilidad de poder trabajar sobre distintas Bases de Datos existentes en el mercado se logra haciendo que los accesos a la misma se realicen a través del uso de objetos que ocultan los detalles que esto implica, ofreciendo a los demás componentes del sistema una interface única, independiente del vendedor del RDBMS subyacente. Para esto, hemos utilizado un patrón de diseño conocido como *Abstract Factory* [GA95] que permite desacoplar a un cliente de un componente en particular.

Hemos utilizado distintas herramientas para la construcción del servidor que detallamos a continuación:

- C++ para la escritura de los componentes que hacen a la arquitectura básica del sistema.
- Librerías OCI (Oracle Call Interface) para realizar la interface entre el motor y el DBMS.
- Lenguaje C y las herramientas standard *lex* y *yacc* para la generación del parser de sentencias en BBTSQL2.
- PL/SQL para la escritura de algunas funciones que realizan el soporte temporal dentro de la Base de Datos.
- El servidor de consultas está construido sobre una Base de Datos ORACLE 8.0.5 u 8.1.6.

5.2. COMPONENTES

Como se mencionó antes, la división del trabajo en el sistema está basada en el modelo

Cliente/Servidor, siendo el último el que realiza todo el procesamiento de una sentencia y delegando en el cliente solo las tareas de mantener una sesión con el servidor y recibir los datos enviados por este.

5.2.1. SERVIDOR

Este es el componente principal del sistema y lleva a cabo el soporte temporal de la Base de Datos.

Para la comunicación entre un cliente y el servidor, definimos un protocolo simple de intercambio de mensajes. Existen tres grupos de mensajes básicos: de *sentencias*, de *control* y de *datos*. Los primeros son los que se envían desde el cliente hacia el servidor para hacer que este procese una sentencia escrita en BBTSQL2. Las sentencias de control, proveen al cliente información acerca del estado del servidor, como por ejemplo la cantidad de conexiones abiertas en un momento dado. Finalmente, los mensajes de datos son enviados desde el servidor hacia el cliente como resultado de la ejecución exitosa de una sentencia o un mensaje de error en caso contrario.

5.2.2. CLIENTES

Los programas clientes deben hacer uso de una API (**A**pplication **P**rogram **I**nterface) que permite comunicarse con el servidor. Mediante el uso de funciones implementadas en esta API, un programa escrito en algún lenguaje de alto nivel puede acceder al estrato intermedio para crear objetos, realizar consultas y actualizaciones. La interface definida es extremadamente simple y fácil de usar. Presentamos aquí algunas líneas de código en lenguaje C para ejemplificar el uso de la API.

```
main() {
    char   lpszNombre[50];

    TSQLInitialize();
    retCode = TSQLExecuteStmt( "select nombre from empleado, depto \
              where valid(empleado) contains valid(departamento) and \
              empleado.coddept = departamento.coddept" );

    TSQLBind( 1, lpszNombre );
```

```
    while ( !TSQLFetchNextRow() )
        printf( "Nombre: %s\n", lpszNombre );

    TSQLDisconnect();
}
```

El programa mostrado envía una consulta mediante el uso de la función `TSQLExecuteStmt()`. A continuación se define que la primer columna de la respuesta será almacenada en la variable `lpszNombre` mediante la invocación a `TSQLBind()`. Luego, accede a todas las filas correspondientes al resultado utilizando `TSQLFetchNextRow()`. Las funciones `TSQLInitialize()` y `TSQLDisconnect()` tienen el objetivo de iniciar y terminar una sesión con el servidor respectivamente.

El trabajo principal de la API es la de transformar en paquetes de TCP (Transmission Control Protocol) las solicitudes realizadas por el programa enviándolas al servidor, comenzando una espera por la respuesta. Cuando esta es obtenida, el procesamiento puede continuar de distinta forma, dependiendo de si se trata de una consulta o una sentencia DDL/DML.

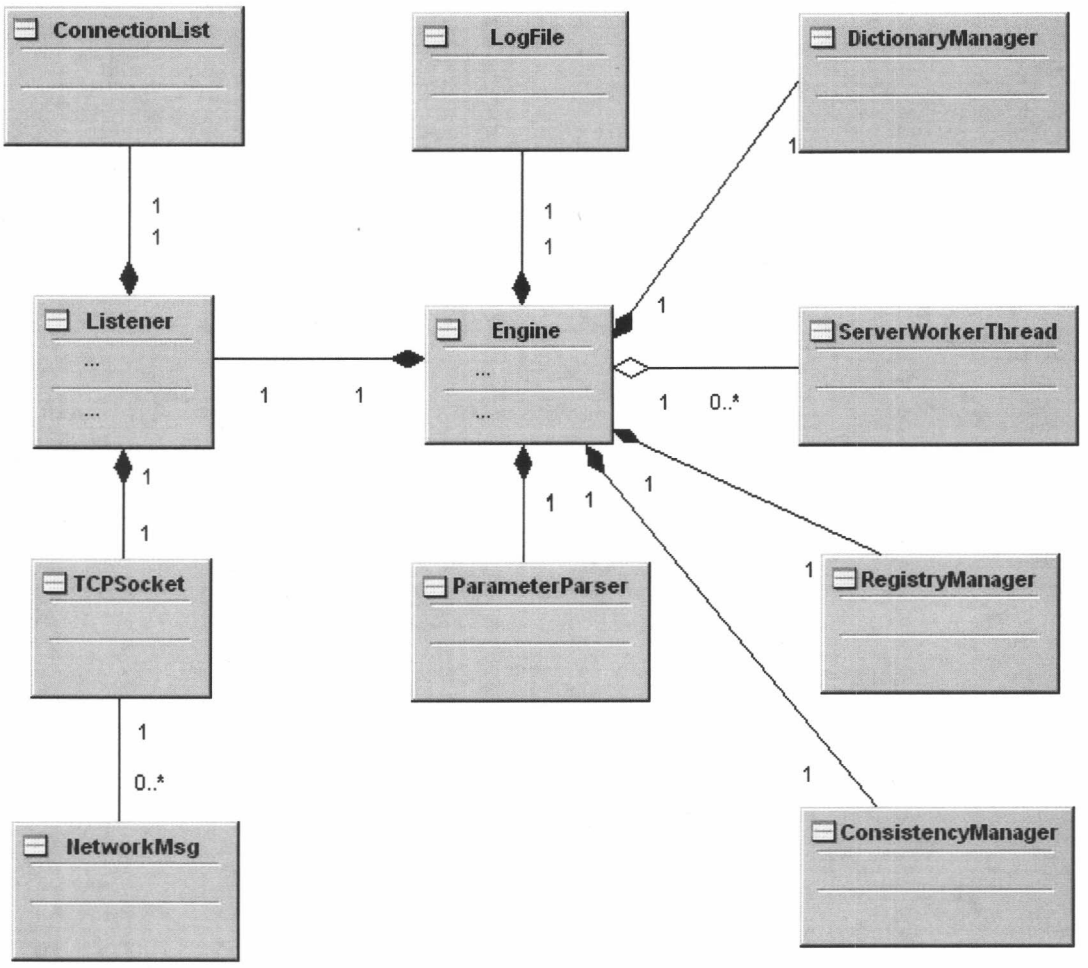
Esta forma de división de trabajo parte de la idea de disminuir la cantidad de paquetes que deberían enviarse entre el motor y los clientes si la "inteligencia" del sistema se distribuyese.

5.3. LA ARQUITECTURA INTERNA DEL ESTRATO

Subdividimos la arquitectura del estrato en dos niveles: la primera sirve como esqueleto del servidor y contiene todas aquellas funcionalidades que permiten la administración de conexiones, registración de eventos en archivos de log, obtención de valores de parámetros, etc. La segunda se encuentra gobernada por la primera y su función es la de efectivamente, procesar solicitudes escritas en BBTSQL2.

Todos los componentes se encuentran coordinados o son creados por un objeto que es creado como primera actividad del sistema. El objeto en cuestión pertenece a la clase *Engine* y las relaciones que esta clase mantiene con las demás se presentan en la figura 5.2 en un diagrama

basado en la notación OMT (Object Modelling Tool) [BOO94].



- **Figura 5.1**. Componentes del estrato medio. Las relaciones corresponden al tipo "tiene un" o "conoce a un".

La fig. 5.1 muestra algunas de las clases que conforman el estrato intermedio como así también las relaciones de uso que existen entre ellas. Sobre la izquierda de gráfico, vemos las clases encargadas de la administración de la comunicación con los clientes: *ConnectionList*, *Listener*, *TCPSocket* y *NetworkMsg*. El resto de las clases están orientadas a la obtención de resultados y comunicación con la Base de Datos subyacente.

El diseño de la fig. 5.1 no es completo, debido a que la clase *DictionaryManager* no es directamente instanciada por *Engine*. En su lugar, hemos utilizado un patrón de diseño conocido

como *Abstract Factory* [GA95] de manera tal de que las instancias creadas estén más desacopladas del cliente (en este caso, quien hace uso de ellas). A continuación incluimos una tabla de los componentes del sistema y sus responsabilidades básicas:

#	Nombre	He-ren-cia	Tipo	Responsabilidad
1	ActionExecutor		A	Ejecutar las acciones asociadas a un evento particular.
2	ConnectionList		C	Mantener la lista de conexiones abiertas con el servidor. La información registrada es: hora de comienzo, operación que esta realizando, port sobre el cual se esta realizando la comunicación.
3	ConsistencyManager		C	Mantener la consistencia de las ramas alternativas a partir de las acciones definidas por los usuarios. Una instancia de esta clase puede ser creada dentro del servidor ante una invocación de la función TSQLBranchConsistency() definida en la API.
4	DatabaseAccess		A	Describir la interface de una clase que debe acceder a un DBMS.
5	DataBuffer		C	Buffer de datos que contiene la información sobre columnas, longitudes de estas, tipos de dato y datos mismos una vez ejecutada una consulta.
6	DBAccess	4	C	Interface especifica para Oracle de una sesión.
7	DBAccessFactory		A	Describe la interface de un constructor de conexiones a Bases de Datos.
8	DBExpressionEvaluator	13	C	Evaluador de expresiones de eventos provenientes de las filas de la Base de Datos. Es capaz de evaluar mas de una expresión cuando se trata de resultados de juntas.
9	DictionaryManager	11	C	Escribir y obtener información del diccionario de metadatos. Ninguna otra clase puede accederlo directamente ya que carecen de la "inteligencia" para hacerlo. Este es específico para Oracle.
10	DictionaryManagerFactory		A	Describir la interface de un creador de fábricas de DictionaryManagers.
11	DictionaryMgr		A	Describir la interface que debe ofrecer un manejador del diccionario de metadatos genérico.
12	Engine		C	Mantener el ciclo de funcionamiento del servidor: recibir conexiones, disparar procesos para la atención de conexiones y cerrarlas utilizando para esto servicios provistos por otras clases.
13	ExpressionEvaluator		C	Evaluador de expresiones genéricas.
14	Instant		C	Conocer las operaciones que pueden realizarse sobre un instante. Un instante es un punto en el tiempo como por ejemplo los extremos de un

				período.
15	Listener		C	Recibir solicitudes de conexión.
16	LogFile		C	Escribir en archivos de log las actividades del motor y sus sesiones. Se crea un archivo de log para el motor y uno por cada nueva sesión abierta. En estos se registran todas las operaciones realizadas como así también los errores obtenidos durante la ejecución.
17	MessageReader		C	Obtener los mensajes de error a partir de sus códigos.
18	NetworkMessage		C	Mantener la información correspondiente a un paquete de TCP enviado desde y hacia el servidor.
19	OracleActionExecutor	1	C	Ejecutor de acciones específico para Oracle.
20	OracleDBAccessFactory	7	C	Crear e inicializar conexiones con la Base de Datos Oracle.
21	OracleDictionarMgrFactory	9	C	Crear e inicializar un DictionayManager.
22	ParameterParser		C	Leer y obtener los parámetros a partir del archivo de configuración del servidor.
23	Period	13	C	Manejar las operaciones propias de los períodos: Meets, Overlaps, etc. Un *Period* está compuesto por dos *Instants*.
24	RegistryManager		C	Obtener parámetros globales que se encuentran en la Registry de MS Windows NT.
25	TCPSocket		C	Recibir y enviar paquetes de TCP a través de un socket específico.

- **Tabla 5.1**: las clases que componen el servido. La columna herencia muestra a partir de que clase *x* una clase *y* obtiene su comportamiento básico. La columna TIPO indica si la clase es Abstracta o Concreta[6].

La tabla 5.1 muestra las clases que componen el servidor. Las clases abstractas permiten hacer mas genérico al diseño de manera de facilitar su portabilidad a otros DBMSs.

Ante cada solicitud de conexión, el estrato medio crea un nuevo *ServerWorkerThread*, el cual estará dedicado a la atención de la nueva conexión.

Los accesos a la Base de Datos por parte del estrato se encuentran separados. *Las sesiones no pueden hacer acceso directo a los metadatos almacenados en el diccionario del estrato. Es solo la clase DictionaryManager la que puede realizar esta clase de accesos y para eso cuenta con una conexión exclusiva.* El

[6] Una clase abstracta no es instanciable y se utiliza para describir la interface de algún componente generico mientras que si se pueden encontrar instancias de una clase concreta.

resto de las sesiones, también cuentan con una sesión exclusiva, por lo tanto, en el caso de nuestra implementación sobre ORACLE, la cantidad de conexiones a la base es igual a la *cantidad de sesiones abiertas más uno (durante la consistencia de ramas y signals, también se crean conexiones adicionales).*

5.3.1. API

La API con la que cuentan los clientes para realizar solicitudes al estrato medio posee las siguientes funciones:

FUNCIÓN	DESCRIPCIÓN
`TSQLInitialize()`	Esta función realiza la inicialización interna de la comunicación a nivel TCP enviando un mensaje al servidor.
`TSQLDisconnect()`	Realiza la desconexión y terminación de la sesión en el servidor.
`TSQLExecuteStmt()`	Provoca la ejecución de la sentencias enviada como parámetro, ya sea, DDL o DML.
`TSQLCommitTrans()`	Realiza la confirmación de los cambios en la Base de Datos.
`TSQLRollbackTrans()`	Deshace los cambios realizados por la sesión desde el último "commit".
`TSQLFetchNextRow()`	Si se ejecutó una sentencia `SELECT`, con esta función se pueden obtener las filas que conforman el resultado. Se debe llamar a `TSQLFetchNextRow()` para cada fila que exista en el resultado.
`TSQLBindValue()`	Para la obtención de los valores de las columnas en un resultado, se deben definir áreas de memoria/variables en donde la API guardará dichos valores. Mediante esta función se realiza una asociación uno a uno entre las columnas y las variables del programa. El llamado a esta función debe ser previo a la invocación de `TSQLFetchNextRow()`.
`TSQLGetColCount()`	Retorna la cantidad de columnas con las que cuenta el resultado.
`TSQLGetColLength()`	Devuelve la longitud de una columna dada.
`GetLastTSQLError()`	Devuelve un string con la descripción de último error reportado.
`TSQLGetJoinValidTime()`	Debido a que el lenguaje carece de sintaxis para obtener el tiempo en que dos o más filas se solapan, proveemos esta función para obtener dicho período de superposición de cada fila en el resultado (resultado del coalesce).
`TSQLCancelQuery()`	Un query que se encuentra en ejecución puede ser interrumpido invocando a esta función.
`TSQLBranchConsistency()`	Provoca la consistencia de las ramas de futuros alternativas sean consistentes con el estado actual de la información presente[7].

- **Tabla 5.2**: funciones de la API.

5.4. SOPORTE TEMPORAL

Para llevar a cabo el soporte temporal sobre una Base de Datos relacional, definimos estructuras para poder mantener la información asociada a las nuevas construcciones del lenguaje.

[7] Ver: Capítulo 2, punto 10 "La consistencia de los estados alternativos".

5.4.1. METADATOS

Hemos creado un diccionario encargado de almacenar los tipos de tablas y sus granularidades como así la de las columnas y los nuevos tipos. Este diccionario está compuesto por ocho tablas:

NOMBRE DE LA TABLA	DESCRIPCION
BTSQL_ACTIONS	Acciones asociadas a eventos definidos por los usuarios
BTSQL_COL_TYPES	Tipos posibles de columnas
BTSQL_EVENTS	Eventos definidos por los usuarios y sus salidas.
BTSQL_FIELDS_DEFS	Columnas de las tablas definidas por los usuarios. Almacena la longitud, el tipo, la granularidad en caso que corresponda, como así también si es o no clave de la tabla.
BTSQL_GRANULARITIES	Contiene todas las granularidades posibles.
BTSQL_QUERIES	Sentencias que se están o fueron recientemente ejecutadas por thread.
BTSQL_TABLE_DEFS	Descripciones de tablas
BTSQL_TABLE_TYPES	Tipos de tablas posibles

- **Tabla 5.3**: Tablas del diccionario de metadatos.

Estas tablas son mantenidas por un objeto de la clase *DictionaryManager*. El *parser*, durante el análisis de una sentencia, solicita a esta clase descripciones e información sobre tablas y columnas para realizar el chequeo de tipos y granularidades.

Una vez terminada la traducción de la sentencia de BBTSQL2 a SQL, el parser almacena el resultado en la tabla `BTSQL_QUERIES` junto con aquellas opciones que deben ser ejecutadas posteriormente por el estrato. De aquí, el encargado de atender a la sesión obtendrá la nueva sentencia y la ejecutará.

El diccionario permite obtener el formato al que deben ser convertidas las fechas dependiendo de la granularidad. Esta información se obtiene de la tabla `BTSQL_GRANULARITIES`. Además, se hace uso intensivo de esta tabla en todas las funciones relacionadas con el manejo temporal (`Cast`, `Overlaps`, `Meets`, etc.).

5.4.2. OBJETOS ADICIONALES

Existen tres tipos de tablas manejados por el estrato: *snapshot*, *valid* y *branching*. El primero no requiere de ningún soporte temporal. *Valid* y *branching* requieren de soporte temporal y por lo

tanto detallaremos como son soportados estos tipos de tablas.

Se puede ver a las tablas que requieren de soporte temporal como separadas en secciones: la sección *corriente* y la sección *histórica* (aquí se encontrarían las filas que fueron reemplazadas por actualizaciones o eliminaciones). En el caso de las tablas *branching* una nueva sección puede agregarse y corresponde a los datos que dependen de la salida de algún evento no ocurrido aún y por lo tanto no son corrientes.

La implementación aquí presentada mantiene tanto a los datos corrientes como a los datos históricos en la misma sección. Es decir, que existe una sola tabla en donde conviven datos pasados, presentes y futuros que son independientes de la ocurrencia de eventos. Las tablas *branching* se encuentran divididas y los datos cuya existencia es aún dependiente de la ocurrencia de un evento se encuentran separados en una tabla adicional.

5.4.3. FUSIÓN TEMPORAL

Uno de los requisitos que impone el lenguaje TSQL2 es que las relaciones deben estar temporalmente fusionadas (*coalesced*). La decisión acerca de cuándo realizar el *coalesce* es dependiente del implementador ya que si bien el usuario debe tener dicha visión fusionada de las filas, esto no implica que estas deban ser almacenadas de esa forma. Esta flexibilidad permite decidir en que momento se hará la operación. Nuestra implementación almacena las filas ya fusionadas. Para eso, creamos un *trigger* encargado de realizar la fusión temporal para cada tabla que se crea. También es necesario un *trigger* para el caso de las actualizaciones.

Debido a que ORACLE impide que se modifique la tabla sobre la que se está trabajando durante la ejecución del trigger, cada tabla posee una tabla temporal asociada adonde se insertan los resultados intermedios. Finalmente, la ejecución de un trigger sobre la tabla temporal hace que las filas allí almacenadas reemplacen a las anteriores en la sección corriente.

La elección de usar *triggers* para realizar el *coalesce* tiene ventajas y desventajas que deben ser analizadas. Una ventaja es que el tiempo de ejecución se reduce gracias a que no hay tiempo de transferencia hacia el estrato medio para realizar la operación. Por otra parte, se hace un buen uso de la tecnología existente. La desventaja es que el *sistema pierde portabilidad* debido a que el

código de los *triggers* no sería compatible en otros DBMSs y debería ser escrito para cada una específicamente.

A modo de resumen, mostramos los objetos que son creados cuando se agrega una tabla valid o branching (la columna "**VALID?**" indica si el objeto es o no creado para una tabla de tipo `VALID`):

OBJETO	TIPO	VALID?	DESCRIPCIÓN
`<tabla>_TEMP`	TB	✓	Tabla que es utilizada durante la ejecución de los triggers para colocar filas temporarias. Estas luego reemplazarán a las filas originales.
`<tabla>_BRANCH`	TB		Aquí se almacenan aquellas filas que dependen de algún evento no ocurrido aún.
`INSERT_<tabla>`	TG	✓	Realiza el Coalesce de filas sobre la tabla.
`UPDATE_<tabla>`	TG	✓	Idem. Este trigger es más complejo que el anterior debido a que más casos pueden darse: el cambio de una expresión de eventos implica que las nuevas filas deben ser insertadas en la sección BRANCH de la tabla dando origen a las alternativas futuras.
`AUPD_<tabla>_TEMP`	TG	✓	Trigger de la tabla temporal asociada a la tabla branching o valid. La ejecución de este trigger hace que las filas alojadas en ella sean movidas a las secciones corrientes o branching.
`<tabla>_SEQ`	SQ	✓	Esta secuencia se utiliza para numerar a las filas dentro de la relación para permitir el encadenamiento histórico de las actualizaciones.
`<tabla>_VW`	VW	✓	Unión de todas las secciones de la tabla exceptuando a la tabla temporal.
`<tabla>`	TB	✓	La tabla que contiene los datos corrientes.

- **Tabla 5.4**. Objetos creados para el soporte temporal en donde TB: Tabla, TG: Trigger, SQ: Sequence y VW: View.

La administración de estos objetos y su creación es absolutamente transparente para los usuarios y nunca deben hacer referencias explícitas a ninguno de ellos.

5.4.4. EXPANSION DE LAS RELACIONES

Cada tabla *valid* o *branching* tiene campos adicionales que permiten el soporte temporal:

NOMBRE DEL CAMPO	DESCRIPCIÓN
TT_FROM	**Inicio del período del tiempo de transacción**
TT_TO	**Fin del período del tiempo de transacción**
VT_FROM	**Inicio del período del tiempo válido**
VT_TO	**Fin del período del tiempo válido**
BTSQL_EVENTEXPR	**Expresión de eventos de la fila. Solo existente en tablas branching.**
BTSQL_UPDATE	**Indica si la fila proviene de un update y solo se utiliza como control durante la ejecución de los triggers de insert y update.**
BTSQL_MYID	**Identficicación de la fila para mantener la línea histórica de actualizaciones**
BTSQL_ACTION	**Indica si la fila es producto de una acción asociada a un evento o de una sentencia directa de un usuario (un insert branching, por ejemplo).**

- **Tabla 5.5**. Campos agregados a las tablas temporales.

Se debe destacar que la eficiencia de los triggers depende en gran medida de la performance en la búsqueda de las filas que hacen *coalesce*. Para esto, aconsejamos la creación de índices sobre las columnas más selectivas. Es importante notar que el hecho de almacenar todas las filas corrientes en la misma tabla haga que se pierda el concepto de clave como tal, ya que si se actualiza un campo que no forma parte de la misma, aparecerán más filas con los mismos valores dentro de la tabla. La clave única no puede existir en estas tablas. La cláusula `PRIMARY KEY` que hemos agregado en la sintaxis (ver Capítulo 3.4), permite la creación de un índice no único sobre la/s columna/s seleccionadas.

5.4.5. SOPORTE TEMPORAL DENTRO DEL DBMS

Hemos creado una serie de procedimientos dentro de la Base de Datos permiten realizar ciertas operaciones temporales directamente cuando la consulta es enviada al DBMS. Estos procedimientos implementan operadores como *Meets*, *Contains*, *Overlaps*, *Cast*, etc. Al igual que en el caso de los *triggers*, estos ofrecen la ventaja de que se aprovecha al máximo la tecnología subyacente (en este caso, el manejo de fechas) y por otra parte la migración a otro tipo de DBMS podría verse desfavorecida. Creemos, de todos modos, que las ganancias a nivel

performance justifican la reescritura de los procedimientos y simplifican la implementación del estrato.

El principal objetivo de estos procedimientos es reducir el tamaño de las sentencias generadas por el *parser* al realizar la traducción de sentencias BBTSQL2 a SQL-92, utilizando funciones en los predicados. Estos podrían haberse convertido en uno o más predicados equivalentes en la sentencia resultante, pero reduciría la legibilidad de las mismas. Adicionalmente, nos permitieron probar funcionalidades aisladamente y facilitar la construcción.

5.4.6. OPERACIONES ADICIONALES QUE REALIZA EL ESTRATO

El estrato es el encargado de obtener las filas y enviarlas al cliente cuando este las requiere. El hecho de almacenar las filas temporalmente fusionadas, reduce principalmente la cantidad de filas corrientes que son tenidas en cuenta, debido a que una sola fila puede reemplazar a varias. Esto favorece la performance de las consultas a pesar de que aumenta el costo de las inserciones y actualizaciones en gran medida. De todas formas, la consulta es la operación más realizada en una Base de Datos y creemos que en ese sentido es mejor sacrificar algo de velocidad en una inserción o actualización.

Como se mostró en el capítulo 2, mantener una relación temporalmente fusionada no garantiza que el resultado de una consulta sea correcto (ver Cap. 2.4).

Para esto, fue necesario implementar dentro del estrato la operación *Coalesce* y junto con ella, clases necesarias para dar soporte temporal a dicha operación. El estrato, se encarga de determinar si es o no necesario realizar esta operación en función de los tipos de tablas intervinientes en una consulta. Las clases intervinientes en esta operación son: *Instant*, *Period* y *DBAccess*. El método `CoalesceResult()` definido en la clase *DBAccess* es el encargado llevar a cabo la operación y determinar el período de la fila fusionada. El cálculo del tiempo de superposición de dos o mas filas en una junta requiere de la obtención de cada uno de los períodos de validez de cada una de ellas. Para esto el parser durante la generación de la nueva sentencia, proyecta columnas adicionales. Estas columnas agregadas no son accesibles por el usuario.

Para el manejo de alternativas o estados posibles (*branching time*) hemos diseñado clases que realizan la evaluación de las expresiones almacenadas en las tuplas de las relaciones branching. Si bien el evaluador es simple, está preparado para evaluar expresiones más complejas que las que se encuentran en la Base de Datos. Las clases que realizan la evaluación de expresiones son *ExpressionEvaluator* y *DBExpressionEvaluator* (ver tabla 5.1).

Es necesario hacer algunas observaciones con respecto a la evaluación de expresiones que provienen de juntas (*joins*). Sea *e* un evento cuyo resultado aún es desconocido (“***not yet known***”) y E una expresión de eventos que proviene de una tupla t_a de la relación branching r, en donde *current*(r) representa a las tuplas que no dependen de la ocurrencia de un evento y *branching*(r) a los estados posibles contenidos en r y por lo tanto dependientes de la ocurrencia de uno o más eventos. Supongamos que E involucra solamente a *e*(es decir, E=‘e’ o E=’ ¬ e’) y se realiza la siguiente consulta:

```
select * from A;
```

Si una tupla $t_a \in$ *current*(r), el resultado de la consulta debería contener a t_a. Por el contrario si $t_a \in$ *branching*(r) entonces t_a no debería pertenecer a conjunto resultante dado que su existencia depende de *e* y su salida es aún incierta.

Para resolver esta potencial ambigüedad, definimos una función $Eval(t, E_t)$, en donde t es una tupla y E_t su expresión de eventos, de la siguiente forma:

$$Eval(t, E_t) = \begin{cases} verdadero\ si\ t \in current(r) \vee (t \in branching(r) \wedge E_t\ verdadero) \\ not\_yet\_known\ sino \end{cases}$$

De la definición de la función *Eval* se desprende que cualquier hecho registrado en la sección corriente de una relación es siempre verdadero por defecto.

Expandimos la función *Eval* para aquellos casos en los que la evaluación requiere que ciertos eventos sean tomados como ciertos o falsos. Este caso se presenta cuando se realizan consultas

especificando la cláusula `BRANCHING`. La inclusión de esta cláusula en una sentencia BBTSQL2 provoca que los eventos allí especificados sean tomados como verdaderos o falsos durante la evaluación, más allá del resultado desconocido que estos tengan en la realidad, provocando el efecto de estados alternativos de los datos.

Entonces, *Eval* es:

$$Eval(C, t, Et) = \begin{cases} E't, \text{ donde } E't = hacer\_verdadero(Et, C) \text{ si } t \in current(r) \\ E't, \text{ donde } E't = hacer\_falso(Et, C) \text{ si } t \in branching(r) \end{cases}$$

En donde C representa al conjunto de eventos proporcionados en la cláusula branching.

La función *hacer_verdadero*(E_p C) devuelve una expresión de eventos en donde aquellos eventos que no tienen una salida definida aun y no se encuentran en el conjunto C, son reemplazados por verdadero. *Hacer_falso* realiza una operación similar, pero reemplazando por falso los eventos indefinidos y que no se encuentran en el conjunto C. En la implementación, no realizamos este reemplazo de antemano, sino que lo hacemos a medida que la evaluación de la expresión avanza.

En función de lo dicho, se desprende que es necesario determinar la procedencia de las distintas expresiones de evento que obtiene el evaluador.

Cuando un usuario realiza una consulta, la traducción hecha por el *parser* agrega la concatenación de las expresiones de eventos de cada fila interviniente de manera que esta sea visible por el estrato[8] formando una sola expresión.

Supongamos que un usuario realiza la siguiente consulta:

```
Select Empnombre, LocNombre, valid(Empleados), valid(Locales)
from Empleados, Locales
Where Empleados.LocId=Locales.LocId and Locales.LocId="LPA"
Branching E1;
```

La lista de campos que produce el estrato en la nueva sentencia que es enviada a la base de datos se transforma en:

[8] Recordemos que de acuerdo con los criterios de diseño, el estrato medio no debe intervenir en la consulta y por lo tanto debe poder acceder solo a los resultados finales de la evaluación, pudiendo si, realizar operaciones posteriores.

```
SELECT  DISTINCT t1.Empnombre, t1.LocNombre, BT.QC(t1.vt_from,t1.vt_to,56),
BT.QC(t2.vt_from,t2.vt_to,56)  ,
BT.overlapped('/'||to_char(t1.vt_from, 'yyyymmddhh24miss')
||to_char(t1.vt_to,'yyyymmddhh24miss') || '/'||56||'/'||
to_char(t2.vt_from, 'yyyymmddhh24miss')
||to_char(t2.vt_to,'yyyymmddhh24miss')||'/'||56)
AS Valid,
BT.overlapped('/'||to_char(t1.tt_from, 'yyyymmddhh24miss')
||to_char(t1.tt_to,'yyyymmddhh24miss') ||'/'||63||'/'||
to_char(t2.tt_from,
'yyyymmddhh24miss')||to_char(t2.tt_to,'yyyymmddhh24miss') ||'/'||63)
AS Trans,
t1.Origen ||t1.Btsql_EventExpr || '/' || t2.Origen ||t2.Btsql_EventExpr
FROM ...
```

La última expresión (resaltada en **negrita**) proyecta las expresiones de eventos de cada una de las tuplas provenientes de las tablas intervinientes en la consulta. Los atributos `t1.Origen` y `t2.Origen` indican si la procedencia de la fila: corriente o *branching*. Una expresion resultante de esta consulta tendrá el aspecto siguiente:

```
1^( E1 )/2^!( E4 )
```

Cada expresión se encuentra precedida por un valor de un dígito: '1' o '2'. Basado en estos valores, el evaluador decide que hacer cuando encuentra un evento cuya salida es aún desconocida. Si la expresión se encuentra precedida por un '1' entonces cualquier valor '*not yet known*' hará que la evaluación continue asumiendo un valor afirmativo para la salida del evento. Si el dígito es '2', cuando el evaluador detecte una salida aún desconocida, detendrá la evaluación lo que produce un resultado falso. *El resultado de la evaluación es el resultado de la conjunción de cada evaluación parcial.* El código encargado de estas evaluaciones se puede encontrar en la clase *DBExpressionEvaluator.*

5.6. CONSISTENCIA DE LAS RAMAS ALTERNATIVAS

Como mencionamos anteriormente, la posibilidad de consultar futuros alternativos se encuentra basada en BTCDM. Este modelo presenta un problema que relacionado con la consistencia de las tuplas que se encuentran en ramas alternativas cuando estas son el resultado de una acción que se ejecuta sobre información corriente. *¿Qué pasa cuando se actualizan atributos cuyos valores están basados en el valor presente?* El problema se hace evidente cuando se definen acciones/futuros posibles basados en valores de atributos presentes. Esto fue analizado en el Capítulo 3.

Por esta razón, debimos introducir durante la implementación una solución para corregir este defecto en el modelo. Nuestra solución cuenta con un componente adicional, el *ConsistencyManager* (ver tabla 5.1) que se encarga de realizar la consistencia de las ramas alternativas con respecto a los valores actuales. El *ConsistencyManager* realiza su trabajo basado en un algoritmo simple que detallamos a continuación:

Sean r_b una relación branching, t_e la expresión de eventos asociada a la tupla t.

Para cada relación branching r_b *hacer*
    $t_e = \varnothing$ *para cada tupla que proviene de una acción asociada a un evento*[9] *en donde* $t \in current(r_b)$.
    $r_b = r_b - \{t\}$ *en donde* $t \in current(r_b)$
fin para

Para cada evento e hacer
    ejecutar(a) en donde a es una acción asociada a e.
fin para

La tabla que mantiene las acciones almacenadas (`BTSQL_ACTIONS`) almacena dos versiones de la acción. La primera es la acción exactamente como fue definida por el usuario en el momento de su creación. Una segunda versión de esta, es aquella que generó el parser en el momento de realizar su análisis sintáctico, es decir, su traducción a SQL. Es esta última acción la que se ejecuta en el momento de realizar la consistencia de las ramas (la primera es utilizada durante la señalización de un evento).

5.7. EJEMPLOS DE TRADUCCIÓN BBTSQL2 A SQL

En este apartado mostramos un par de ejemplos sobre el tipo de traducción que realiza el estrato medio para obtener una sentencia SQL a partir de una BBTSQL2. La sentencia obtenida como resultado es luego enviada al DBMS para ser procesada. Una vez que el DBMS termina su ejecución se obtiene un *cursor*[10] que es recorrido para obtener las filas correspondientes al

[9] Para distinguir las filas que provienen de acciones asociadas a eventos y aquellas que son el resultado de la ejecución de una sentencia BBTSQL2 escrita especificamente por un usuario, expandimos las relaciones con un atributo cuyos valores posibles son A y U.

[10] Un mecanismo provisto por el DBMS para poder recorrer las filas que resultan de la ejecución de una consulta de forma programática.

resultado de la consulta. Durante la obtención de filas ocurre la última etapa de filtrado dado que las expresiones de eventos son evaluadas por el estrato medio. Es decir que es el estrato el que decide que tuplas son enviadas a la API.

EJEMPLO 1

Sentencia BBTSQL2

```
Select EmpNombre, LocId from Empleados where EmpNombre="Pablo" and
period [date '1990' year, date '2020' year] contains valid(Empleados);
```

Traducción SQL standard

```
SELECT  DISTINCT t1.EmpNombre  , t1.LocId  ,
BT.overlapped('/'||to_char(t1.vt_from,'yyyymmddhh24miss')||
to_char(t1.vt_to,'yyyymmddhh24miss') || '/'||56) AS Valid,
BT.overlapped('/'||to_char(t1.tt_from, 'yyyymmddhh24miss')||
to_char(t1.tt_to,'yyyymmddhh24miss')||'/'||63) AS Trans
FROM EMPLEADOS t1
WHERE t1.EmpNombre = 'Pablo' AND
BT.ContainsPeriod( to_date( '1990', 'yyyy') , to_date( '2020', 'yyyy')
,32,t1.vt_from,t1.vt_to,56)=1  AND  to_char(t1.TT_TO,'yyyy')='9999'
```

EJEMPLO 2

Sentencia BBTSQL2

```
Select Empnombre, LocNombre, valid(Empleados), valid(Locales)
from Empleados, Locales
Where Empleados.LocId=Locales.LocId and Locales.LocId="LPA"
Branching E1;
```

Traducción SQL standard

```
SELECT  DISTINCT t1.Empnombre, t1.LocNombre,
BT.QC(t1.vt_from,t1.vt_to,56),
BT.QC(t2.vt_from,t2.vt_to,56)  ,
BT.overlapped('/'||to_char(t1.vt_from, 'yyyymmddhh24miss')
||to_char(t1.vt_to,'yyyymmddhh24miss') || '/'||56||'/'||
to_char(t2.vt_from, 'yyyymmddhh24miss')
||to_char(t2.vt_to,'yyyymmddhh24miss')||'/'||56)
AS Valid,
BT.overlapped('/'||to_char(t1.tt_from, 'yyyymmddhh24miss')
||to_char(t1.tt_to,'yyyymmddhh24miss') ||'/'||63||'/'||
to_char(t2.tt_from,
'yyyymmddhh24miss')||to_char(t2.tt_to,'yyyymmddhh24miss') ||'/'||63)
AS Trans,
t1.Origen ||t1.Btsql_EventExpr || '/' || t2.Origen ||t2.Btsql_EventExpr
FROM EMPLEADOS_vw t1, LOCALES_vw t2
     WHERE t1.LocId = t2.LocId AND t2.LocId = 'LPA' AND
     to_char(t1.TT_TO,'yyyy')='9999' AND  to_char(t2.TT_TO,'yyyy')='9999'
     AND
```

```
BT.OVERLAPSPERIOD(t1.vt_from,t1.vt_to,56,t2.vt_from,t2.vt_to,56)=1 AND
BT.OVERLAPSPERIOD(t1.tt_from,t1.tt_to,63,t2.tt_from,t2.tt_to,63)=1;
```

Las construcciones sintácticas de BBTSQL2 como '`contains valid(empleados)`' son traducidas a funciones como `OVERLAPSPERIOD`, `CONTAINSPERIOD` que no forman parte del conjunto de funciones soportadas por el DBMS. Estas contienen la lógica de las operaciones temporales y fueron escritas directamente como funciones dentro de la Base de Datos para reducir el tamaño de los predicados que deberían ser agregados para verificar la validez temporal. Si bien esta no es la única forma posible, tienen la gran ventaja de reducir la longitud de las sentencias generadas dado que reemplazan a uno o más predicados. Por otra parte, son escritas en el lenguaje nativo de la base de datos y por lo tanto resultan muy eficientes.

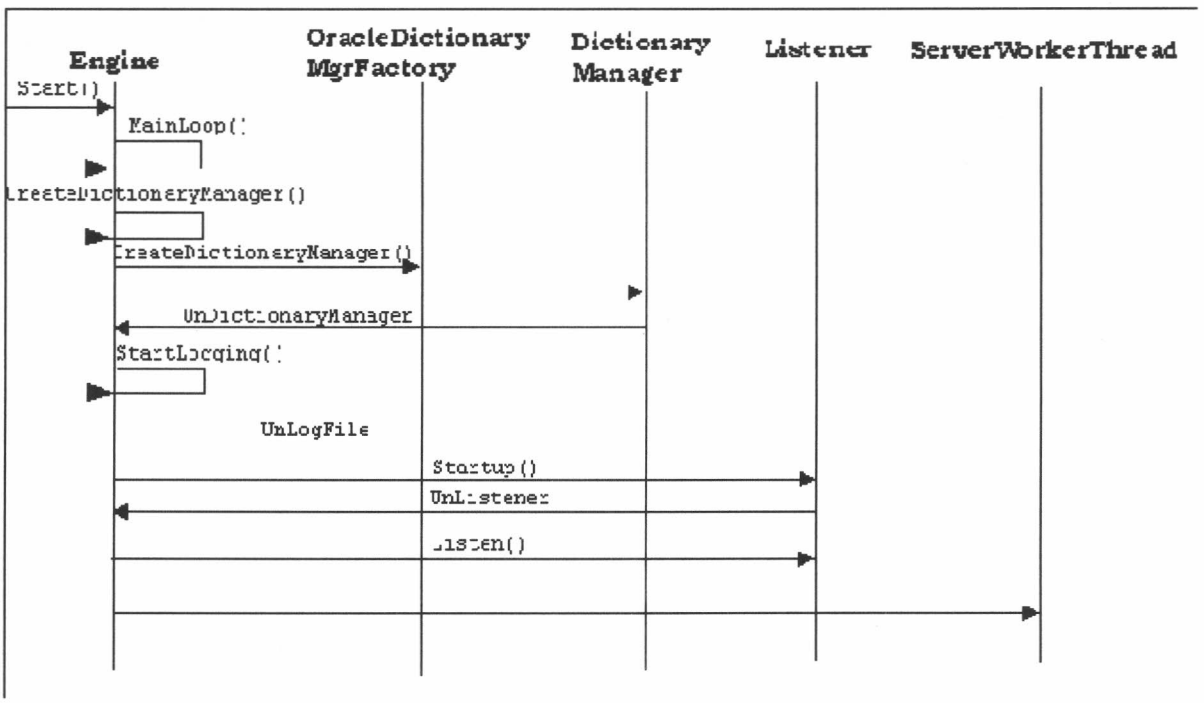
En el ejemplo 2 pueden observarse más modificaciones realizadas por el estrato en lo que hace a la lista de columnas a seleccionar. En primer lugar, se ve como las columnas con expresiones del tipo '`valid(tabla)`' fueron reemplazadas por nuevas expresiones '`BT.QC(<principio_periodo>, <fin_periodo>)`'. Estas funciones dan formato al período para su presentación posterior, convirtiendo a ambas columnas en un solo período[11].

Adicionalmente, el *parser* ha agregado nuevas columnas para obtener las expresiones de eventos de las tuplas resultantes para poder realizar un filtrado posterior basándose en los valores de verdad de dichas expresiones (ver sección 5.5).

5.8. EL PROCESAMIENTO DE LAS SENTENCIAS

A lo largo de los apartados anteriores, hemos repasado las diferentes estructuras que conforman el servidor de consultas temporales, los componentes/clases y las tablas que mantienen los metadatos necesarios para la traducción y ejecución de sentencias escritas en BBTSQL2. En este apartado mostramos la interacción entre distintos componentes en dos instantes de la vida del servidor: el comienzo de una conexión por solicitud de un cliente y la ejecución de una consulta.

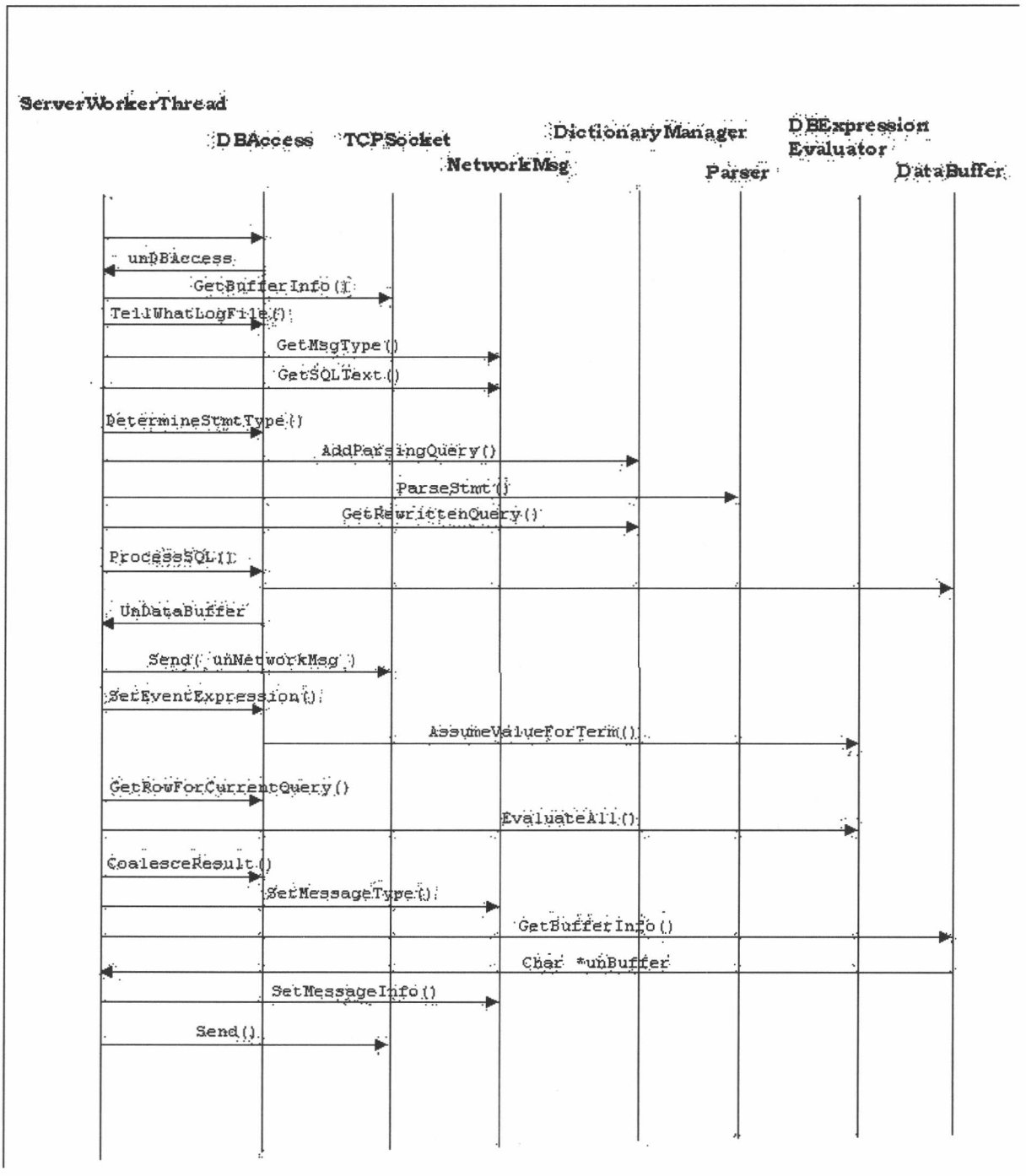
[11] Es necesario recordar que internamente los periodos se encuentran representados por dos columnas de tipo DATE.



- **Figura 5.2**: Interacción entre los componentes durante la iniciación de una comunicación Cliente/Servidor.

En la figura 5.2, vemos la secuencia de inicialización del servidor hasta el momento en el motor recibe una solicitud de conexión. Las flechas indican el método que está siendo invocado en cada momento. Una vez que se ha realizado la invocación al método `Listen()`, se comienza una espera por una solicitud de conexión. Cuando esto ocurre, un nuevo *thread* cuyo punto de entrada es la función `ServerWorkerThread` es iniciado.

En la figura 5.3, mostramos la secuencia de invocaciones que ocurre una vez que el nuevo *thread* fue iniciado. La secuencia de invocaciones no es única y aquí solo mostramos la correspondiente a la ejecución de una sentencia `SELECT` que incluye la opción `BRANCHING` (invocaciones a `SetEventExpression` y `EvaluateAll` en `DBExpressionEvaluator`) y con fusión temporal (`CoalesceResult` en la clase `DBAccess`).



- **Figura 5.3**: Colaboraciones que se dan cuando se solicita la ejecución de una consulta.

En primer lugar, una sesión posee una conexión exclusiva con la Base de Datos subyacente y por ese motivo se crea una instancia de *DBAccess* para administrar dicha

conexión. Una vez que esto se ha realizado con éxito, procedemos a obtener el buffer que procede de la capa de TCP a través del método `GetMsgType()` de la clase *TCPSocket*. `GetSQLText()` devuelve un cadena de caracteres consistente en la sentencia enviada por la sesión del usuario. Aquí el camino de ejecución puede dividirse en dos y esta división está dada por el tipo de sentencia que se ha recibido: DML/DDL o una consulta. En los dos primeros casos el camino es idéntico. En el caso de una consulta, el procesamiento sigue la linea mostrada en el gráfico. La determinación del tipo de sentencia es realizada por el método `DetermineStmtType()` de la clase DBAccess. La sentencia es luego agregada a una tabla de donde será posteriormente leída por el *parser*. Luego de ser invocado a través de la función `ParseStmt()`. En caso de un análisis sintáctico correcto, el *parser* almacena la sentencia resultante en una tabla del diccionario. El thread obtiene esta nueva sentencia invocando al método `GetRewrittenQuery()` de la clase *DictionaryManager*. La sentencia es ejecutada por el DBMS a través de la invocación a `ProcessSQL()` definida en la clase *DBAccess*. Este método define el *buffer* en donde se alojarán de a una las filas obtenidas. El resultado de la operación es indicado al cliente enviando un mensaje (`Send()`). Debido a que consideramos el caso de una consulta branching, se almacenan los eventos definidos por el usuario en la cláusula branching como una propiedad del evaluador (`AssumeValueForTerm()`): este paso define que eventos serán asumidos como verdaderos por el evaluador (*DBExpressionEvaluator*) durante la evaluación del resultado. Una vez que el cliente solicita una fila del resultado, el *thread* comienza a acceder a estas a través del método `GetRowForCurrentQuery()`. Aquí debe evaluarse la expresión de eventos contenida en el último campo de la selección: esta tarea es llevada a cabo por una instancia de la clase *DBExpressionEvaluator* invocando al método `EvaluateAll()`. El resultado debe ser temporalmente fusionado y por esa razón se invoca al método `CoalesceResult()` definido en la clase *DBAccess*. Las invocaciones restantes se encargan de enviar la fila obtenida al cliente. El próximo requerimiento de una fila por parte del cliente disparará un nuevo ciclo que comienza con la invocación al método `GetRowForCurrentQuery()`.

5.9. LIMITACIONES IMPUESTAS POR LA IMPLEMENTACIÓN

5.9.1. LIMITACIONES DEL ALCANCE DE LAS CONSULTAS BRANCHING

Si bien la posibilidad de implementar un modelo temporal mediante la estrategia de estratos reduce los tiempos de desarrollo y permite el estudio de lenguajes temporales, introduce algunas limitaciones.

Sean los eventos:

Aumentan_ganancias: los márgenes de ganancia de la empresa aumentan en más de un 3%.

Nuevos_departamentos: se crean nuevos sectores que dan origen a movimientos de empleados entre departamentos dentro de la empresa.

Sea la consulta: "*listar todos los nombres de los empleados tales que pertenecen a los departamentos existentes entre el 10/01/1999 y el 10/03/2001 en el caso de que se hayan creado nuevos departamentos suponiendo que aumentaron las ganancias de la empresa*" cuyo equivalente en BBTSQL2 es:

```
SELECT nombre FROM empleados WHERE dept_id IN (SELECT dept_id
            FROM departamentos
            WHERE VALID(departamentos) CONTAINS
            PERIOD[DATE '10/01/1999', DATE '10/03/2001']
            BRANCHING nuevos_departamentos )
BRANCHING aumentan_ganancias;
```

Como mencionamos en el apartado 5.5, el *parser* agrega la concatenación de las expresiones de eventos de cada fila a la lista de campos seleccionados por el usuario, ya que estos deben ser visibles por el estrato para su evaluación posterior (ver puntos 5.5 y 5.7). Para la evaluación correcta de esta sentencia, debería poder accederse al resultado intermedio que genera la consulta no correlacionada para poder filtrar aquellas filas que cumplen con el evento `nuevos_departamentos`. Esto no es posible debido a que el estrato solo puede obtener el resultado final que produce el DBMS una vez que la consulta es convertida a SQL. Esto implica que los atributos pertenecientes a la relación departamentos no son visibles desde la consulta externa y por lo tanto no se puede evaluar la expresión de eventos interna. Este es un problema de alcance que no se limita solo a este ejemplo, pero ocurre cuando se tienen consultas con

subconsultas y los atributos de esta última deben ser visibles por el estrato.

Para hacer posible la implementación de tales queries, se requiere el desarrollo de los operadores de pertenencia de conjuntos (`IN`, `=`, `EXISTS`, etc.) procesando cada query por separado de manera tal que el motor pueda tener el control de los resultados intermedios, lo cual escapa al alcance de este trabajo y por otra parte sería opuesto al criterio de no modificación del DBMS subyacente. Otra alternativa sería la de convertir esta clase de consultas en juntas, pero garantizar la equivalencia de la consulta resultante una vez hecha la conversión no es un problema trivial.

Por estas razones, nuestra implementación no permite la inclusión de cláusulas *branching* en subconsultas.

5.9.2. CONSTRAINTS TEMPORALES

La implementación presentada en este trabajo no incluye la definición de *constraints*. De todas formas, estas deberían ser una extensión de los *constraints* existentes en los DBMS.

Mencionamos anteriormente el problema que resulta de la existencia de más de una fila con valores equivalentes en sus atributos visibles, pero en donde sus marcas temporales no se solapan. Claramente, esta situación impide la existencia de una clave única (*Primary Key*) que no incluya al tiempo. Nuestra implementación permite la creación de índices sobre columnas seleccionadas por el usuario.

Otro caso de *constraints* de uso común en las Bases de Datos relaciones son las claves foráneas. La semántica de estas en un modelo temporal se ve afectado, debido a que las dependencias deben ser también temporales, es decir, incluir en estas el tiempo de validez. Para mostrar la clase de problemas que podrían surgir al no cumplir con el requisito de ser temporales presentamos el siguiente ejemplo:

Consideremos la relación `EMPLEADOS` y `LOCALES`. Indicamos que un empleado se encuentra trabajando en un local dado incluyendo un atributo `emplocid`, en la relación `EMPLEADOS`. Los valores posibles de estos atributos están gobernados por la existencia de un valor equivalente en

la relación LOCALES. Esta condición es suficiente cuando se trata de una Base de Datos relacional, pero no cuando es temporal. Debido a que los períodos de validez (existencia en el mundo real) pueden no ser los mismos, debe incluirse la validación de solapamiento temporal de la validez de las tuplas. Esto incluye predicados de desigualdad, dado que los períodos de validez no deben ser necesariamente iguales (un local es, generalmente, anterior a la existencia de empleados que trabajen en el). El manejo de esta clase de *constraints* añadiría una gran complejidad al estrato y por esta razón nuestra implementación no contempla la existencia de tales *constraints* temporales.

5.9.3. AGREGACIÓN

Como mencionamos anteriormente, la evaluación de consultas a ramas alternativas provoca que el *parser* incorpore las diferentes expresiones de eventos provenientes de las tuplas en cuestión a la proyección especificada por el usuario. Este agregado permite que no necesitemos intervenir en la ejecución de la consulta por parte del DBMS de acuerdo con los criterios definidos en 4.1. Una consecuencia de esto es que las cláusulas de agregación o grupos (GROUP BY, COUNT, SUM, etc.) se ven afectadas, ya que el agregado de expresiones requeridas por el estrato medio en el conjunto de atributos seleccionados por el usuario impide el agrupamiento original (aquel que se indicó en la consulta original). **Por esta razón, una restricción que impone la implementación es la de no poder realizar consultas de tipo *branching* con agregación requiriendo que el programador que desee esta posibilidad escriba código específico para hacerlo**. Aquí se debe tener en cuenta que la fusión temporal impone un orden en la sencuencia en que las filas son entregadas al usuario y este orden puede facilitar la tarea del programador, debido a que las funciones de agrupamiento pueden sacar provecho del orden impuesto por el estrato.

5.9.4. INDEXACIÓN TEMPORAL

No existe en nuestra implementación la capacidad de crear índices temporales o indexación temporal. Esto es, la posibilidad de crear un acceso veloz a través del tiempo válido o transaccional de una tabla. En principio, esta restricción se debe a que hemos basado la implementación de las columnas que dan soporte temporal a las tablas en el tipo DATE. Esto obliga al estrato a aplicar funciones de conversión a dichas columnas para obtener la

granularidad necesaria. Debido a que ORACLE no hace uso de índices cuando se aplican funciones a las columnas indexadas[12], no tiene sentido generar índices sobre dichos atributos. Existe la posibilidad de aumentar el *overhead* de atributos por tabla manteniendo cuatro columnas adicionales con la conversión a caracteres de las fechas en la granularidad correspondiente y luego indexar estas nuevas columnas.

5.10. SUMARIO

Hemos dado en este capítulo una descripción de la implementación de los componentes construidos de acuerdo a la arquitectura presentada en el capítulo 4. También analizamos las restricciones que presenta la implementación de servidores de consultas utilizado estratos medios. La principal de las desventajas es que estos no pueden acceder a los resultados intermedios que produce el DBMS y solo pueden hacerlo al resultado final, debiendo realizar procesos posteriores para filtrar el resultado que debe ser enviado al usuario. En algunos casos, como la agregación, fuerzan al estrato medio a agregar información adicional a la consulta original para poder llevar a cabo tareas como la evaluación de expresiones.

Definimos la funcion *Eval* encargada de realizar la evaluación de expresiones dependiendo de sección de orígen de las tuplas en una relación temporal *branching*: corriente o alternativa futura.

Presentamos la política utilizada para realizar una de las operaciones que provocan los mayores trastornos en las relaciones *branching* cuando se realizan modificaciones en el presente que afectan a datos alternativos: la consistencia de las ramas alternativas. Sin esta operación, la base de datos corre serio riesgo de perder la congruencia cuando datos alternativos son dependientes de información corriente.

Analizamos las ventajas y desventajas de las decisiones tomadas en cuanto a la implementación de ciertos componentes y la utilización de ciertas características de la base de

[12] Esto si ocurre en las versiones posteriores a ORACLE 8.1.5. De todas formas, esta no es una caracteristica presente en todos los DBMS del mercado.

datos para la realización de ciertas tareas como una solución de compromiso entre performance y flexibilidad del sistema.

CAPITULO 6

UN CASO DE ESTUDIO

Presentamos en este capítulo un caso de estudio que permita mostrar con ejemplos las principales aplicaciones reales que tiene el lenguaje BBTSQL2. Incluimos también un análisis comparativo de performance de aquellas operaciones que la diferencian de un DBMS relacional.

6.1. CASO DE ESTUDIO

6.1.1. DESCRIPCIÓN DEL EJEMPLO

Como mencionamos en el Capítulo 1, los ejemplos se encuentran basados en un caso bastante común que refleja la información de una empresa que cuenta con locales distribuidos en regiones. Se registran también los datos de los empleados y en que local trabaja cada uno de ellos. La empresa comercializa una serie de productos y lo que deseamos en general es obtener datos consolidados acerca de ventas a nivel local o regional como así también realizar proyecciones futuras acerca del desarrollo de la empresa.

El esquema se crea con las siguientes sentencias:

```
create table Empleados ( EmpId  varchar2(3) , EmpNombre varchar2(10), LocId
varchar2(3), Turno interval hour to second )
As Branching Time Year To Day

create table Locales ( LocId  varchar2(3) , LocNombre varchar2(10), RegId
varchar2(3) )
As Branching Time Year To Day

create table Regiones (RegId  varchar2(3) , RegNombre varchar2(10) )

create table Productos (ProdId varchar2(3) , ProdNombre varchar2(20) )

create table Precios (ProdId  varchar2(3) , Precio number(5,2) )
As Branching Time Year To Second

Create table Ventas (ProdId  varchar2(3), LocId  varchar2(3) , EmpId
varchar2(3) , Cantidad number(6))
As Valid Year to Second
```

6.1.2. CONSULTAS

6.1.2.1. TIEMPO VÁLIDO Y TRANSACCIONAL

1. Una consulta en la que se utiliza el tiempo transaccional es: "*listar las fechas en que fue modificado el precio del Café Franca Blanca*"

```
SELECT Precio, transaction(Precios)
FROM Precios, Productos
WHERE Precios.ProdId=Productos.ProdId and ProdNombre= "Cafe Franja Blanca";
```

Se obtiene el siguiente resultado:

13	[27/05/2000 15:58:07-Forever]
23	[27/05/2000 15:58:07-Forever]

Indicando que el precio fue alterado en ambos casos el 25 de mayo de 2000 y no fue modificado posteriormente.

2. En cambio con el tiempo válido podemos realizar la siguiente consulta: *"listar la historia de precios de todos los productos"*

```
SELECT ProdNombre, Precio, valid(Precios)
FROM Productos, Precios
WHERE Precios.ProdId=Productos.ProdId;
```

Siendo el resultado:

Café Franja Blanca	12	[01/05/1991 00:00:00-20/12/1999 23:59:59]
Café Franja Blanca	13	[21/12/1999 00:00:00-31/12/2001 23:59:59]
Café Superior	22	[01/03/1991 00:00:00-20/12/1999 23:59:59]
Café Superior	23	[21/12/1999 00:00:00-31/12/2001 23:59:59]
...		

El resultado muestra los precios como así también los períodos de validez de cada precio. Por ejemplo, el Café Franja Blanca costó $12 desde el 1 de mayo de 1991 hasta el 20 de diciembre de 1999, aumentando posteriormente a $13 a partir del 21 de diciembre del mismo año.

6.1.2.2. COALESCE

Los ejemplos siguientes ilustran el funcionamiento de la fusión temporal automática que

realiza el estrato. Supongamos que se quiere extender el contrato de uno de los empleados cuyo vencimiento estaba previsto originalmente para el 31 de diciembre del año 2010 hasta el año 2015. Para esto, basta con insertar una nueva tupla con el período de contratación restante. Es decir:

```
Insert into Empleados ( EmpId , EmpNombre, LocId, Turno)
values ("A03", "Pedro","SUC" , interval '8' hour)
Valid Period [date '1/1/2011' year to day, date '31/12/2015' year to day]
```

3. Sea ahora la consulta: *"Período de validez de contrato del empleado Pedro"*. Para esto se ejecuta la siguiente consulta:

```
Select EmpNombre, valid(Empleados) from Empleados where EmpNombre="Pedro";
```

El resultado obtenido es:

Pedro	[01/01/1990-31/12/2015]

La fusión temporal previa se realizó durante la inserción de la extensión del contrato, pero como se mencionó en el Capítulo 2, existen casos en donde esto no es suficiente. Mostramos a continuación un caso en donde se hace explícita la fusión temporal durante la operación de selección.

Supongamos que el empleado Pablo recibe una extensión de contrato igual a la que recibió Pedro, pero adicionalmente es asignado a otra sucursal. En ese caso insertamos el nuevo período de contratación:

```
Insert into Empleados ( EmpId , EmpNombre, LocId, Turno)
values ("A05", "Pablo","SUC" , interval '7' hour)
Valid Period [date '2011' year, date '2015' year];
```

4. Luego, podemos realizar la siguiente consulta: *"Nombre y localidad donde trabaja el empleado Pablo"*

```
Select EmpNombre, LocId from Empleados where EmpNombre="Pablo" and
period [date '1990' year, date '2020' year] contains valid(Empleados);
```

Pablo	MPU	[1990/01/01 2010/12/31]
Pablo	SUC	[2011/01/01 2015/12/31]

Un resultado diferente se obtiene si no se consulta la sucursal a la que pertenece Pablo, en cuyo caso la consulta se expresa como:

```
Select EmpNombre from Empleados where EmpNombre="Pablo" and
```

```
period [date '1990' year, date '2020' year] contains valid(Empleados);
```

Pablo	[1990/01/01-2015/12/31]

La diferencia se debe a que la segunda consulta es equivalente a preguntar durante qué período Pablo es empleado de la empresa, sin importar a que sucursal pertenece. Por lo tanto ambos períodos de validez son colapsados en uno solo que cubre a ambos.

Hemos extendido la sintaxis de BBTSQL2 para permitir conjuntos no fusionados temporalmente. Esto resulta especialmente útil cuando el usuario sabe que la operación no es necesaria y evitando así el alto costo del *coalesce*. La consulta anterior podría alterarse agregando la cláusula NOCOALESCE:

```
Select EmpNombre from Empleados where EmpNombre="Pablo" and
period [date '1990' year, date '2020' year] contains valid(Empleados)
NOCOALESCE;
```

El resultado de esta nueva consulta es:

Pablo	[1990/01/01-2010/12/31]
Pablo	[2011/01/01-2015/12/31]

6.1.2.3. JOINS

Las juntas temporales, como mencionamos en capítulos anteriores, tienen un semántica diferente a la de los DBMSs relacionales que no tienen soporte temporal. Aquí las juntas deben garantizar que el resultado es contemporáneo, es decir, que las filas provenientes de las tablas que forman parte de la junta no solo deben cumplir con el predicado del *join* sino que también deben solaparse en el tiempo.

Para esto, insertamos un empleado que trabajará en la sucursal Sucre del 2011 al 2015. Debido a que la sucursal Sucre estará abierta hasta el año 2010, ninguna tupla debería formar parte del resultado cuando se relacione al nuevo empleado con esta sucursal. Las sentencias necesarias son las siguientes:

```
Insert into Empleados ( EmpId , EmpNombre, LocId, Turno)
values ("A08", "Gustavo","SUC",  interval '8' hour)
Valid Period [date '2011' year, date '2015' year];
```

5. Sea ahora la consulta: *"Empleados y locales con código de empleado A08"*

```
Select EmpNombre, LocNombre, valid(Empleados), valid(Locales) from Empleados,
Locales where Empleados.LocId=Locales.LocId and EmpId="A08";
```

Obteniendo un resultado vacío.

Supongamos ahora que el ingreso del nuevo empleado es adelantado un año. Entonces:

```
Insert into Empleados ( EmpId , EmpNombre, LocId, Turno)
values ("A08", "Gustavo","SUC" , interval '8' hour)
Valid Period [date '2009' year, date '2010' year];

Select EmpNombre, LocNombre, valid(Empleados), valid(Locales) from Empleados,
Locales where Empleados.LocId=Locales.LocId and EmpId="A08";
```

Obtendríamos el siguiente resultado:

Gustavo	Sucre	[01/01/2009-31/12/2015]	[01/01/1990-31/12/2010]

6.1.2.4. BRANCHING TIME

Si aumentasen las ventas en Belgrano se inauguraría una nueva sucursal en el año 2001 con un empleado. Para esto creamos un evento y las acciones correspondientes:

```
Create event e1 AumentanVentasBel;

Create action for E1 is
Insert into Locales ( LocId  , LocNombre , RegId  )
values ("LPA", "La Pampa",  "BEL")
Valid Period [date '2001' year, date '2010' year];

Create action for E1 is
Insert into Empleados ( EmpId , EmpNombre, LocId, Turno)
values ("A10", "Patricia","LPA",  interval '6' hour )
Valid Period [date '2001' year, date '2010' year];
```

Ahora que contamos con el evento y las acciones asociadas, podemos realizar consultas sobre las consecuencias posibles de la ocurrencia o no del evento E1.

6. Una primer consulta simple sería: "*listar los empleados que pertenecerán a la nueva sucursal inaugurada ante la ocurrencia de AumentanVentasBel*". Entonces escribimos la consulta:

```
Select Empnombre, LocNombre, valid(Empleados), valid(Locales)
from Empleados, Locales
Where Empleados.LocId=Locales.LocId and Locales.LocId="LPA"
Branching E1;
```

Patricia	La Pampa	[01/01/2001-31/12/2010]	[01/01/2001-31/12/2010]

Un *signal* del evento E1 se utiliza para indicar a la Base de Datos la ocurrencia del mismo.

```
Signal E1=T;

Select Empnombre, LocNombre, valid(Empleados), valid(Locales)
from Empleados, Locales
Where Empleados.LocId=Locales.LocId and Locales.LocId="LPA";
```

Obtenemos un resultado idéntico al anterior, pero la diferencia se encuentra en que la consulta carece de la cláusula BRANCHING. Por lo tanto, que Patricia se convierta en empleada ha dejado de ser una posibilidad para convertirse en un hecho.

6.1.2.5. OPERACIONES TEMPORALES

Una de las grandes ventajas que poseen las bases temporales son las facilidades aritméticas que ofrecen para operar con datos de tipos fecha, período o intervalo. Para ilustrar dichas facilidades, mostramos una serie de ejemplos que hacen uso de ellas.

7. Supongamos la consulta: "*averiguar la fecha de vencimiento del contrato de alquiler del local Cabildo, si extendemos dicho contrato por 2 años 3 meses y 5 días*"

```
Select Locnombre, end(valid(Locales)) + interval '5/3/2' year to day from
Locales where Locid="CAB";
```

Obteniendo así el resultado:

Cabildo	04/04/2013

8. Si ahora la consulta pide que se adelante su vencimiento dos meses

```
Select Locnombre, end(valid(Locales)) - interval '2' month
from Locales where Locid="CAB";
```

Cabildo	01/11/2010

9. O pide que se postergue seis meses y medio la fecha de contrato de alquiler del local

```
Select Locnombre,  valid(Locales)  + interval '15/6' month to day from Locales
where Locid="CAB";
```

Cabildo	[15/07/2010-14/07/2011]

Es importante notar la diferencia entre la consulta 9 y las consultas 7 y 8. En este caso, el intervalo se está sumando a un período, por lo tanto ambos límites son retrasados seis meses y medio.

6.1.2.6. AGREGACIÓN

10. Ahora queremos conocer la cantidad de horas hombre que se trabajarán en cada local durante el año 2001.

```
Select regnombre, sum(Turno)
from Regiones, Locales, Empleados
where Regiones.regid=Locales.regid and Locales.locid=Empleados.locid
and period [date '2001' year, date '2001' year] meets valid(Empleados)
group by regiones.regnombre;
```

Belgrano	29:30:0
Olivos	22:59:59

6.1.2.7. CONSISTENCIA DE RAMAS

Por último mostraremos por qué es necesario el uso de la función de consistencia de ramas (`TSQLBranchConsistency`) para evitar la inconsistencia mencionada en el capítulo 3.

Consideremos el caso en el que tenemos una acción que modifica todos los registros de una tabla que cumplan una determinada condición y agregamos un registro que cumple la condición de la acción después de haber creado la acción entonces la tabla quedará inconsistente.

```
Create event E2 AumentarPrecios;

Create action for E2 is update precios set precio=precio+1 valid period [date
'2001' year, date '2001' year];

Insert into Precios (ProdId , Precio ) values("NGT", 0.5)
Valid Period [date '1/11/2000' year to day, date '31/12/2001' year to day];
```

Si ejecutamos la consulta: *"Id Producto y Precio"* obtendremos un resultado incorrecto, debido a que el precio del producto insertado posteriormente a la creación de la acción no habrá sido actualizado.

```
Select ProdId, Precio, Valid(Precios)
from Precios where ProdId="NGT" branching E2;
```

NGT	0.5	[1/11/2000 00:00:00 - 31/12/2001 00:00:00]

Luego de invocar a la función TSQLBranchConsistency obtenemos un resultado diferente:

NGT	0.5	[01/11/2000 00:00:00 - 31/12/2000 00:00:00]
NGT	1.5	[01/01/2001 00:00:00 - 31/12/2001 00:00:00]

6.1.3. EJEMPLO DE APLICACIÓN EN UN PROGRAMA ESCRITO EN DELPHI Y VISUAL C++

Mostramos aquí la forma en que una aplicación escrita en los lenguajes de desarrollo *Delphi de Borland Intl* y *MS Visual C++*, invocan a funciones de BBTSQL2. Así el lenguaje presentado constituye una forma de BBTSQL2 "embebido".

Supongamos que nuestra empresa paga los sueldos en función de las horas trabajadas y por lo tanto queremos tener una aplicación que sea capaz de estimar la suma de dinero que se pagará durante el año 2001 en función de la cantidad de horas a trabajar y agrupadas por región. Entonces desarrollamos un form que permite ingresar el valor de la hora trabajada y en función de esta entrada del usuario, calcularemos los sueldos.

El form a crear contiene un botón que al ser pulsado, ejecuta el siguiente código:

```
procedure TEstimacionFrm.CalcularBtnClick(Sender: TObject);
var
   CantidadHoras, Region:  array [0..30] of Char;
   Sentencia:              String;
begin
   TSQLInitialize;
   Sentencia := 'Select regnombre, sum(Turno) from Regiones,Locales,Empleados '+
     ' where Regiones.regid=Locales.regid and '+
     ' Locales.locid=Empleados.locid '+
     ' and period [date ''2001'' year, date ''2001'' year] '+
     ' meets valid(Empleados) '
     ' group by regiones.regnombre;';

   TSQLExecuteStmt(PChar(Sentencia));
   TSQLBindValue(0, Region);
   TSQLBindValue(1, CantidadHoras);

   while ( TSQLFetchNextRow <> 1 ) do begin
       ListBox1.Items.Add(String(Region)+'-'+String(CantidadHoras)+
                   StrToInt(StrToInt(CantidadHoras)*StrToInt(ValorEdit.Text));
   end;
   TSQLDisconnect;
end;
```

El código mostrado realiza una conexión al servidor de consultas invocando a la función `TSQLInitialize()`. Una vez creada la sesión se envía la consulta a través de la función `TSQLExecuteStmt()`. Cuando la ejecución de la sentencia es terminada, el servidor de consultas posee un cursor con el cual es posible recorrer el resultado obtenido, accediendo mediante el uso de `TSQLFetchNextRow()`. Aquí el programa realiza un ciclo en el cual luego de ejecutar los

cálculos necesarios, presenta los resultados en un *listbox*[13]. La variable `ValorEdit.Text` corresponde a un componente visual en donde el usuario ingresó el valor horario utilizado para el cálculo. Finalmente, el programa se desconecta invocando a `TSQLDisconnect()`, liberando los recursos obtenidos por la sesión en el servidor y cerrando la comunicación a nivel TCP.

En forma similar se podría utilizar *MS Visual C++* como lenguaje anfitrión como se muestra a continuación:

```
void CTSQL2Queries::OnCalculate()
{
char     lpszCantHoras[30], lpszRegion[30], lpszValorHorario[30];
int      tsqlResult;
char     *Sentencia = "Select regnombre, sum(Turno) from regiones,\
         Locales,Empleados \
          where Regiones.regid=Locales.regid and \
          Locales.locid=Empleados.locid \
             and period [date ''2001'' year, date ''2001'' year] meets\
         valid(Empleados) group by regiones.regnombre;";

  TSQLInitialize();
  TSQLBindValue(0, lpszRegion);
  TSQLBindValue(1, lpszCantHoras);

  ValorEdit.GetText(1, lpszValorHorario);
  lValorHorario = atol(lpszValorHorario);

  while ( ( tsqlResult = TSQLFetchNextRow() ) != 1 ) {
     char lpszOutput[200];

     sprintf(lpszOutput, "%s-%s=%ld", lpszRegion, lpszCantHoras,
             atol(lpszCantHoras)*lValorHorario );
     m_QueryOutput.AddString( lpszOutputLine );
  }
  sprintf( lpszRowBuffer, "%ld fila(s).", iRowsReturned );
  m_QueryOutput.AddString( lpszRowBuffer );
}
```

El esquema del algoritmo es básicamente el mismo que el mostrado para su equivalente en Delphi. Los componentes utilizados son los mismos que en el ejemplo anterior: un form (`CTSQL2Queries`), un *EditBox* (`ValorEdit`), un *ListBox* (`m_QueryOutput`) y un botón (`Calculate`) para invocar al método del form encargado de realizar el cálculo[14].

---

[13] Un *listbox* es un componente visual standard de MS Windows 95, 98 y NT.

[14] Más información acerca de estos componentes standard de Windows y su manejo puede encontrarse en [TO96].

6.2. ANALISIS DE PERFORMANCE

6.2.1 OBJETIVOS Y SUPOSICIONES

Naturalmente, el agregado de las operaciones temporales a una Base de Datos tiene un costo adicional. Mostramos las mediciones realizadas con respecto a las operaciones que sufren el mayor cambio en comparación con la misma operación sin el soporte temporal.

El mayor de los costos en lo que hace al tiempo de ejecución de una de tales operaciones, está asociado con el "coalesce" de filas, requerimiento del modelo de datos de TSQL2. Debido a que es necesario el reemplazo de todas las filas que tienen los mismos valores en sus atributos explícitos y se solapan o encuentran en el tiempo por una nueva tupla cuyo tiempo de validez cubre a todas las anteriores, el tiempo de inserción es el más afectado. La razón, es que se deben recorrer todas las filas de la tabla para seleccionar aquellas que deben ser reemplazadas por la nueva tupla.

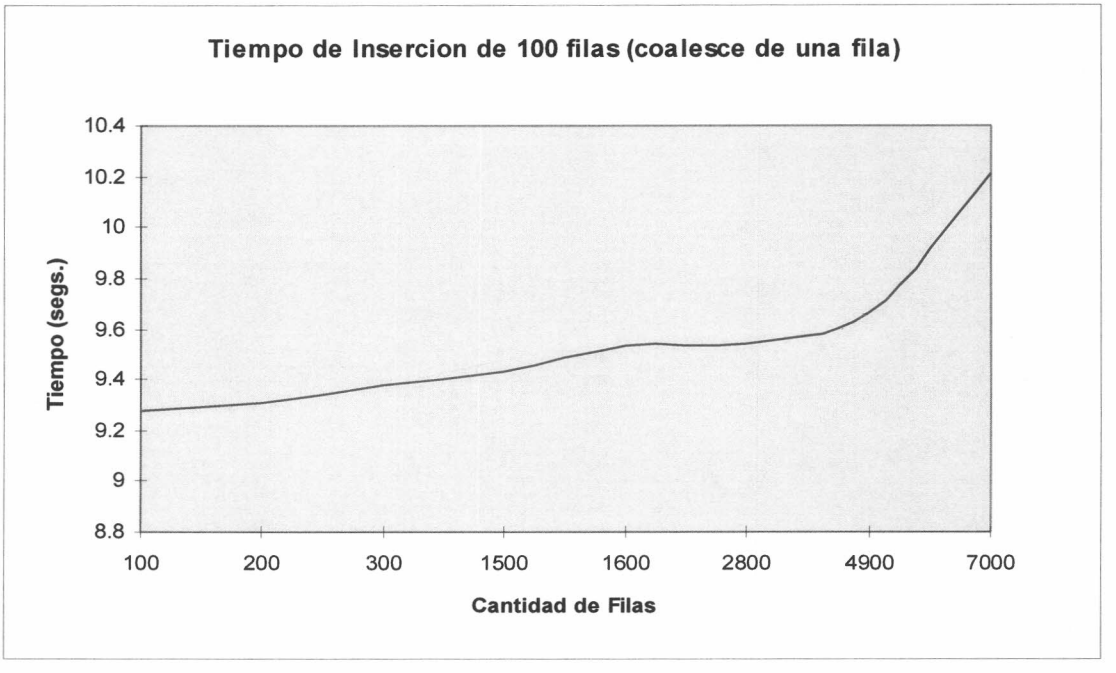
No se exige en nuestra implementación la definición de una clave primaria para ninguno de los tipos de tablas posibles. De todas formas, es recomendable hacerlo debido a que, como se mencionó antes, el recorrer toda la tabla es un paso necesario de la operación de COALESCE. Definiendo una clave primaria, se reduce el tiempo de búsqueda de las filas que deben ser reemplazadas, dado que disminuye la cantidad de filas que deben ser examinadas.

El objetivo de este análisis es principalmente mostrar el impacto de la operación *coalesce* en el tiempo de respuesta de las operaciones de inserción y selección, dado que el resto de las mediciones de rendimiento que hemos realizado sobre aquellas operaciones que solo involucran traducción de BBTSQL2 a SQL, exhiben una performance similar a la de sus sentencias equivalentes en el DBMS subyacente y por lo tanto carece de sentido presentarlas en este trabajo.

6.2.2. RESULTADOS OBTENIDOS

6.2.2.1. OPERACIONES DE ACTUALIZACIÓN

La medición tiene dos dimensiones importantes, que son mostradas en gráficos separados. El primero muestra el costo en tiempo que tiene la inserción dependiendo de la cantidad de filas que contiene la tabla.

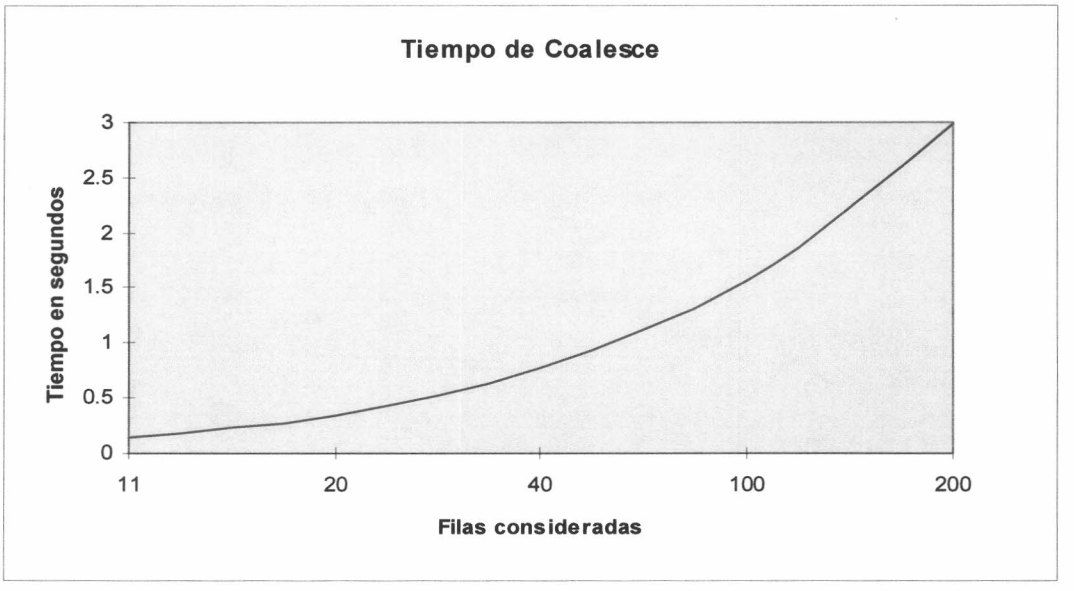


▪ **Figura 6.1**: Tiempos de inserción de una fila.

La medición se realizó haciendo inserciones de lotes de 100 filas en la tabla Empleados y tomando el tiempo total de la operación. Cada una de las nuevas filas insertadas, tenían una fila existente con la que debía ser fusionada en la tabla.

La segunda medición que se llevó a cabo, es la del costo de la operación *coalesce* en función de la cantidad de filas que se solapan en el tiempo, es decir, cuantas cumplen las condiciones necesarias para ser unidas durante la operación de *coalesce*.

En el siguiente gráfico se pueden ver los resultados obtenidos.



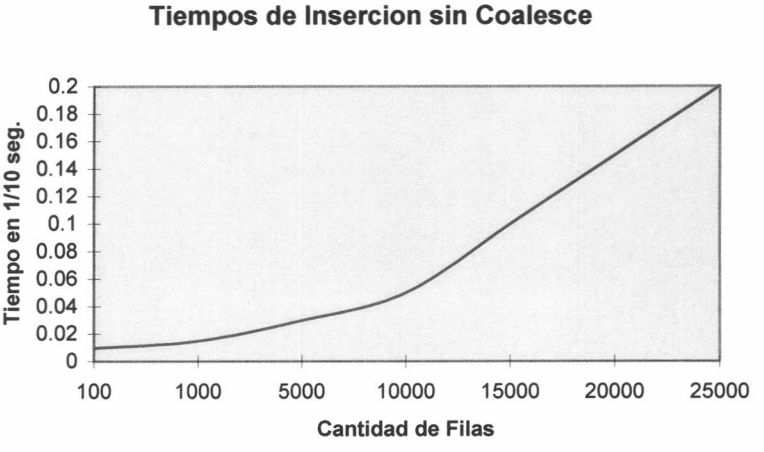
▪ **Figura 6.2**: Tiempos para la operación "coalesce".

Estos resultados se obtuvieron insertando en la tabla Empleados n filas que no se solapaban en el tiempo y finalmente una nueva tupla cuyo tiempo de validez contenía al de todas las insertadas previamente. Para todas las mediciones de *coalesce* graficadas la cantidad de registros de la tabla de prueba es de siete mil.

Tomando como ejemplo el caso de veinte filas consideradas, la figura 6.2 muestra que la inserción de una nueva tupla en la tabla que contiene temporalmente a las demás, implica un costo de inserción de aproximadamente medio segundo.

Como es de esperar, el tiempo de inserción de una nueva fila aumenta en función de la cantidad de filas que cumplen el criterio para hacer *coalesce*.

También es interesante destacar el tiempo de una operación de inserción en donde no interviene la operación de *coalesce*. El costo es notablemente menor cuando esto ocurre.



- **Figura 6.3**: Tiempos de inserción cuando no existe la necesidad de fusionar filas.

Este último testeo permite apreciar las diferencias de performance con respecto al caso en que la operación de inserción requiere de fusión de filas. El costo en tiempo de una inserción de una fila cuando se debe hacer fusión temporal de una fila, es aproximadamente diez veces superior al de una inserción que no lo requiere, siendo el primero de casi diez centésimas de segundo y 1 centésima en el segundo caso.

6.2.2.2. CONSULTAS DE SELECCIÓN

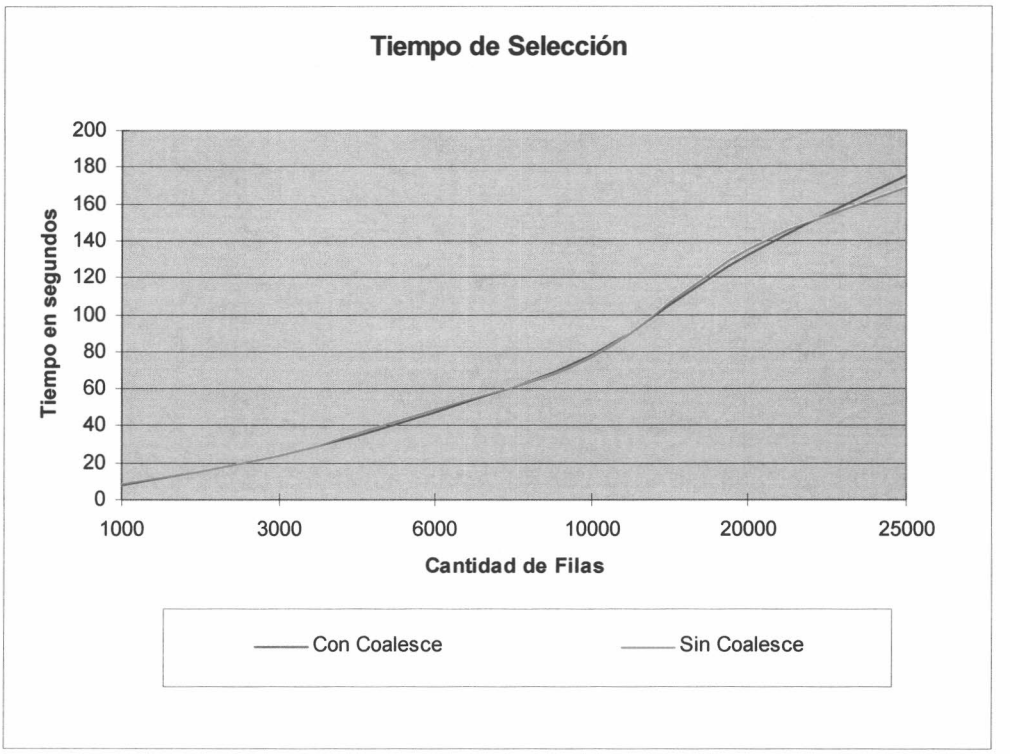
En este apartado comparamos la performance de una consulta de selección sobre una tabla que requiere el uso de la operación *coalesce* contra una consulta similar que no lo requiere.

Hemos usado para estas mediciones la tabla Empleados con 25000 registros. Los tiempos fueron tomados utilizando un programa creado basado en la API presentada en esta Tesis. La medición se corresponde a la finalización de la operación de obtención de filas provenientes del estrato medio. Por ejemplo, el tiempo de selección de la tabla de empleados corresponde a la recepción por parte del programa de la respuesta completa, es decir, las 25000 tuplas.

En primer término, mostramos la performance comparativa entre una selección completa de la tabla de Empleados con y sin fusión temporal, para esto utilizamos las siguientes consultas:

```
Select * from empleados;

Select * from empleados nocoalesce;
```



- **Figura 6.4**: Tiempos de selección de la tabla de Empleados

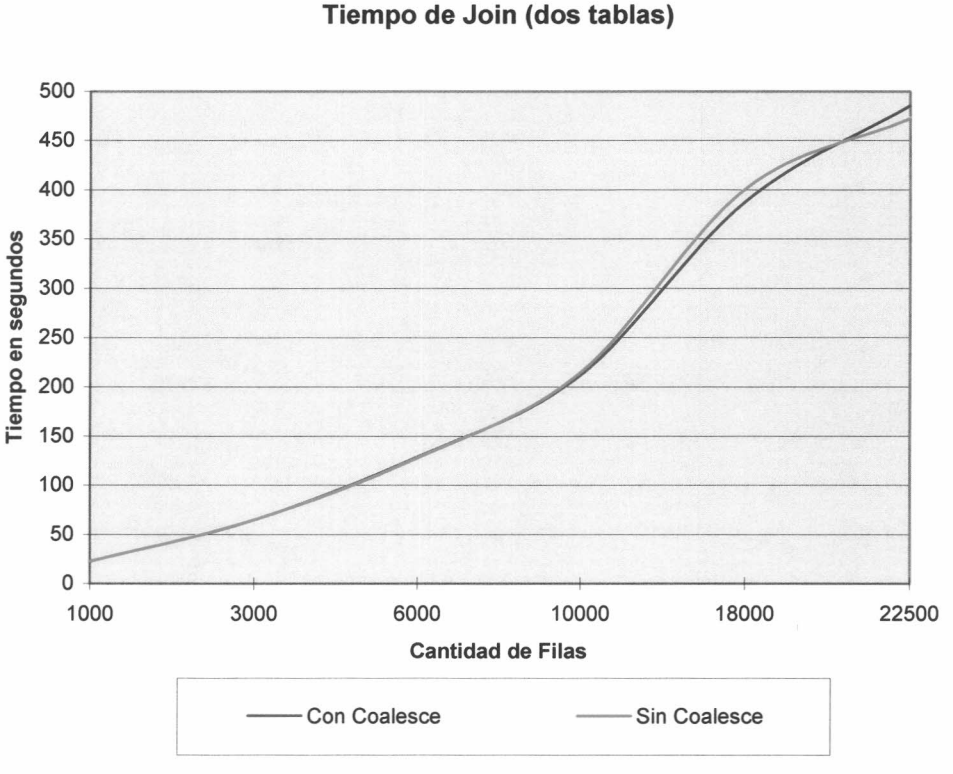
La figura 6.4 muestra el tiempo en segundos requerido para la obtención de las filas provenientes de la tabla de Empleados. Es visible que la performance de la operación no varía prácticamente cuando no se utiliza la opción de fusión temporal. A pesar de esto, en la práctica existe una diferencia importante: cuando el resultado requiere de *coalesce* el tiempo necesario para obtener la primer fila del resultado es mucho más alto que en el caso contrario. La razón de esto es que cuando se ejecuta la función `TSQLExecute` de la API, la sentencia traducida es procesada en la Base de Datos, pero debido a que la fusión temporal requiere de un ordenamiento completo del resultado, el tiempo necesario para realizar esta operación es aproximadamente el tiempo requerido para poder acceder a la primer fila. Por otra parte, esta aparente desventaja, se transforma en una reducción del tiempo de ejecución de la función `TSQLFetchNextRow` debido a que el resultado de las funciones agregadas por el *parser* fueron calculadas con anterioridad a la

operación de ordenamiento. A partir de aquí, se puede deducir que para programas *batch* es conveniente el tipo de selección que hace uso de fusión temporal y para programas que requieren intervención del usuario se obtiene mejor respuesta no utilizando el *coalesce*[15].

6.2.2.3. CONSULTAS DE JOIN

En esta sección comparamos el tiempo requerido para realizar la selección de n filas a partir de una consulta con junta. Las tablas utilizadas fueron Empleados y Locales con 25000 y 10 registros respectivamente. La consulta es

```
Select EmpNombre, Turno, LocNombre
From Empleados, Locales
Where Empleados.LocId=Locales.LocId
```



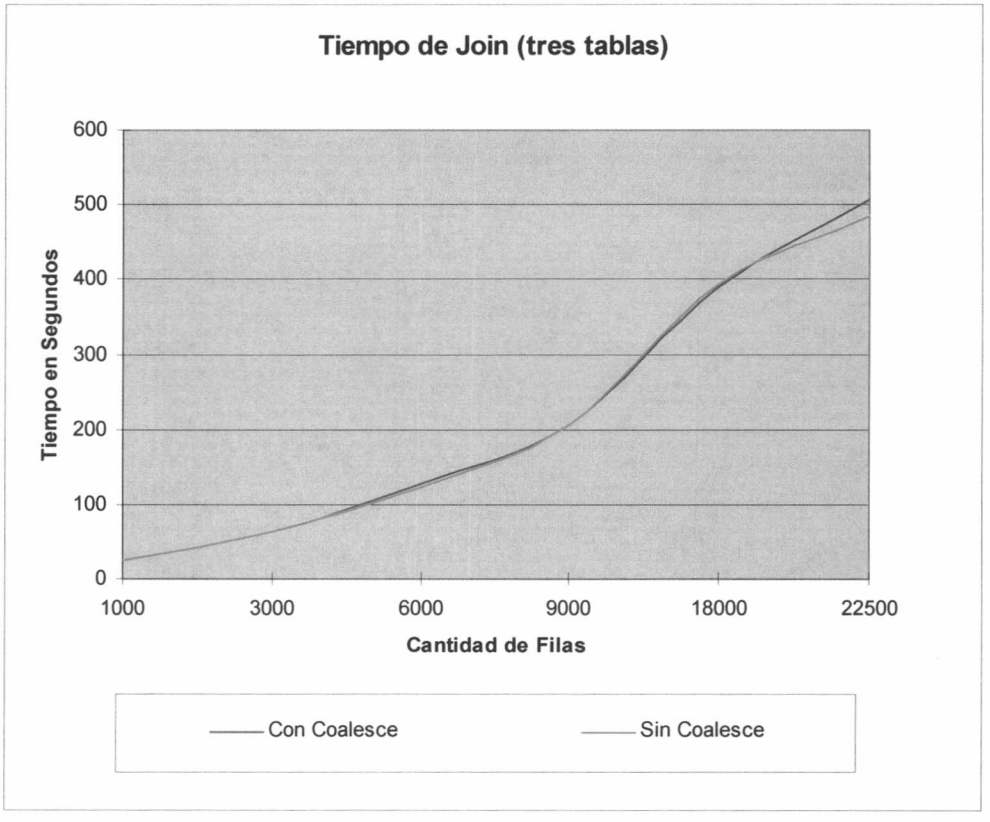
- **Figura 6.5**: Tiempo para la obtención de filas para una consulta con juntas.

[15] En algunos casos, esto no sera posible y normalmente los programas on-line realizan selecciones completas de tablas.

Como en el caso anterior, la cantidad de filas sobre el eje x se refiere a la cantidad de filas obtenidas como resultado de la consulta. Aquí también puede verse que los tiempos de ejecución son prácticamente iguales. El tiempo de ejecución de la obtención de una fila sin *coalesce* es ligeramente superior a su equivalente con *coalesce*, pero el tiempo requerido para poder acceder a la primer fila del conjunto resultante es mucho menor cuando se trata de la selección sin fusión temporal.

Finalmente evaluamos joins de tres tablas: Empleados, Localidades y Regiones, esta última con 10 registros. La consulta es:

```
Select EmpNombre, LocNombre, RegNombre
From Empleados, Locales, Regiones
Where Empleados.LocId=Locales.LocId and Loacles.RegId=Regiones.RegId
```



- **Figura 6.6**: Tiempos obtenidos para consultas con juntas de tres tablas.

Vemos en la figura 6.6 que los resultados para joins de tres tablas son similares a los

obtenidos para juntas de dos tablas.

Es importante notar que el plan de acceso utilizado por Oracle incluía la utilización de índices para obtener las filas de localidades y regiones tomando como base la tabla de `EMPLEADOS` utilizando el algoritmo de junta conocido como *Nested Loops* [UL88]. Por esta razón, un aumento significativo en la cantidad de filas de las tablas localidades y regiones no debería afectar en gran medida los resultados previos.

Podemos decir que el aumento del tiempo en la ejecución que se da a medida que la cantidad de filas pertenecientes al resultado aumenta, es lineal, aunque esta tendencia depende en gran medida de los métodos de optimización utilizados por el DBMS utilizado.

6.3. SUMARIO

En este capitulo hemos presentado ejemplos de consultas que hacen uso de las caracteristicas temporales y sus posibilidades. Mostramos ejemplos de codigo escrito en lenguajes de tercera generacion de uso frecuente como lo son el MS Visual C++ y el Borland Delphi embebiendo sentencias en BBTSQL2.

Finalmente, mostramos que la operación de fusión temporal no representa un gran impacto en el tiempo total de ejecución de una consulta, pero si en el tiempo en el que se obtiene la primer fila del resultado. Esto permite decidir entre que tipo de cláusula utilizar dependiendo de la aplicación que se está construyendo: es natural pensar que en programas con intervención de usuarios la decisión se incline hacia un tiempo menor de respuesta en cuanto a la primer fila a obtener, mientras que en programas batch, dicho tiempo inicial no tiene tanta importancia y sí el tiempo total de procesamiento.

CAPÍTULO 7

CONCLUSIONES Y TRABAJO FUTURO

7.1. TRABAJO FUTURO

La implementación que hemos realizado posee algunas características básicas que pueden ser explotadas en trabajos posteriores. Una de ellas es la posibilidad de obtener la línea histórica de cambios realizados a una tupla en particular. El lenguaje actualmente no posee ninguna construcción que permita obtener tal resultado. La única limitación presente para realizar esta clase de operación es la que se da cuando dos o más filas son "colapsadas" en una sola a raíz de la fusión temporal, debiendo perder la identidad de alguna de los dos "padres" de la nueva fila fusionada. Creemos que deberían ser estudiadas formas posibles para implementar la línea de cambios históricos evitando esta limitación. Una consecuencia de esto sería la de incluir en el modelo operaciones de secuencia que permitirían expresar en forma sencilla y resolver en forma eficiente, consultas del tipo: *"listar los locales en que trabajó Pedro desde noviembre de 1995"*.

En segundo término, creemos que la performance puede ser mejorada a través del uso de la indexación temporal. Esto reduciría en gran medida la cantidad de filas a considerar en ciertas operaciones, descartando lo más pronto posible aquellas que se encuentran fuera de los rangos consultados.

Creemos que es necesaria también la profundización del estudio de una implementación más eficiente de la consistencia de ramas, ya que en este momento requiere de un alto costo computacional.

Finalmente, consideramos que la introducción de grados de certeza y probabilidades dentro del modelo en lo que hace a la ocurrencia de evento, podría ser una interesante perspectiva en lo que hace al uso de la información que se encuentra en las ramas alternativas, permitiendo una gama más amplia de consultas haciendo más útil aún la información futura para la planificación.

7.2. CONCLUSIONES

Hemos estudiado en el presente trabajo las posibilidades de la implementación de expandir el soporte de las Bases de Datos relacionales a un modelo temporal. Creemos que la arquitectura de estratos permite un desarrollo que es capaz de proveer la mayor parte de las facilidades temporales deseables en una Base de Datos con un bajo costo de desarrollo y buena performance. La contraparte de este modelo, la arquitectura integrada, debería tener un rendimiento no menor al exhibido por una implementación como la que forma parte de este trabajo de Tesis.

También hemos visto como la arquitectura estratificada impone serias restricciones a la implementación principalmente en lo que hace a las posibilidades de explotar la información de las ramas alternativas futuras.

El manejo de los tiempos de validez y transacción de la información almacenada son propiedades altamente deseables en un modelo de Base de Datos ya que disminuye en gran medida la posibilidad de errores introducidos al desarrollar aplicaciones que deben hacer uso intensivo de información que requiere de un constante manejo de fechas. Esta clase de información forma parte de la gran mayoría de las aplicaciones del mundo actual y ha merecido ser integrada en el standard SQL:1999.

La exploración de las posibilidades que ofrecen los modelos *Branching Time* también resulta de interés, ya que la planificación de estrategias es altamente utilizada en las empresas de hoy y derivar la complejidad que implican estas tareas a la Base de Datos garantiza una coherencia en el manejo de la información y su utilización.

A lo largo de la implementación, hemos visto el alto costo en términos computacionales y de recursos que provoca el uso de la fusión temporal. En algunos casos, la performance de tal operación puede tornarse prohibitiva sobre todo, en sistemas que manejan gran cantidad de datos como por ejemplo, los Sistemas de Soporte de Decisiones. (DSSs). Es por eso, que hemos

optado por extender el lenguaje para permitir la posibilidad de obtener relaciones no fusionadas como resultados a consultas realizadas en BBTSQL2.

Referencias

- [AH90] *Compiladores: Principios y Herramientas.* Aho, Sethi y Ullman. Addison Wesley.
- [B95] M. Böhlen, *Telmporal Database System Implementations.* SIGMOD Record, Vol 24 [4], Diciembre 1995.
- [BOO94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Bejamin/Cummings, Redwood City, CA, 1994. Segunda Edición.
- [CO96] *Redes Globales de Información con Internet y TCP/IP.* 3era edición. Douglas Comer. Prentice Hall.
- [COD70] E. F. Codd. "*A Relational Model for Large Shared Databanks*". Communications of the ACM, 13(6), 377-390, Junio 1970.
- [GA95] *Design Patterns. Gamma, Helm, Johnson, Vlissides.* Addison Wesley, 1995.
- [SA97] *Handling of Alternatives and Events in Temporal Databases.* N.L. Sarda.
- [SNO87] R. T. Snodgrass. *The Temporal Query Language TQuel.* ACM Transactions on Database Systems, 12(2): 247-298. Junio 1987.
- [SNO95] R. T. Snodgrass. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995
- [TO96] *Visual C++ 4 Unleashed.* Victor Toth, SAMS Publishing, 1996.
- [TR2] *Layered Implementation of Temporal DBMSs – Concepts and Technics.* Kristian

Torp, Christian S. Jensen y Michael Böhlen. A TimeCenter Technical Report.

- [TR5] *Stratum Approaches to Temporal DBMS Implementation.* Kristian Torp, Christian S. Jensen y Richard T. Snodgrass.. A TimeCenter Technical Report. Marzo 1997.

- [TR9] *Coalescing in Temporal Databases.* Michael H Böhlen, Richard T. Snodgrass y Michael D. Soo. 1997. A TimeCenter Technical Report. Abril 1997.

- [TR17] *Temporal Data Management.* Christian Jensen y Richard T. Snodgrass. Junio 1997.

- [TR19] *Efficient Conversion Between Temporal Granularities.* Hong Lin. A TimeCenter Technical Report. Julio 1997.

- [UL88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems.* Volumen I. Computer Science Press, 1988.

- [WIE92] Wiederhold, 6. *Mediator in the architecture of future information systems.* IEEE Computer, 25(3): 38-49, Marzo 1992.

- [WA93] Wang, X. S. Jajodia, y V. Subrahmanian. *Temporal Modules: An Approach Toward Temporal Databases.* Proceedings of the ACM SIGMOD on Management Data. 227-236. 1993.

Indice

LISTA DE FIGURAS

Sintaxis de BBTSQL2

Sentencias

Aquí se describen todas las construncciones válidas en el lenguaje definido. Más adelante se encuentran las distintas opciones y funciones built-in reconocidas por el servidor.

CREATE

La sentencias CREATE permite la creación de una tabla de cualquiera de los tres tipos existentes: snapshot, valid y branching time. Su estructura es la siguiente:

```
CREATE TABLE <table_name>
( column_name1 datatype [ ( size) ] [ NOT_NULL [ UNIQUE ] ]
[PRIMARY KEY] ,
  column_name2 datatype [ ( size) ] [ NOT_NULL [ UNIQUE ] ]... )
  [AS VALID | AS BRANCHING TIME ]
  <granularity>;
```

La granularidad especificada en la opción <granularity> es aplicable al tiempo válido. La granularidad del tiempo transaccional es siempre año a segundo (`YEAR TO SECOND`).

La cláusula opcional `PRIMARY KEY` debe especificarse para cada columna que compone la clave primaria de la tabla.

DROP

La sentencia DROP permite eliminar una tabla y todos sus objetos asociados. Su sintaxis es:

```
DROP TABLE <table_name>;
```

INSERT

```
INSERT INTO <table_name> (column1,column2...)
VALUES (value1,value2...)
[ VALID <period> ]
[ BRANCHING [NOT] <event_name1> , [NOT] <event_name2> ... ];
```

Las cláusulas temporales `VALID` y `BRANCHING` solo tienen sentido cuando esta sentencia es ejecutada sobre una tabla *valid* o *branching*.

Si bien `INSERT` permite el agregado de filas futuras alternativas, es conveniente realizar esta operación defieniendo eventos y sus acciones asociadas.

DELETE

Delete permite la eliminación lógica de filas de tablas temporales o la eliminación física de tablas *snapshot*.

```
DELETE FROM <table_name>
[ VALID <period> ]
[ BRANCHING [NOT] <event_name1> , [NOT] <event_name2> ... ]
[ WHERE condition ];
```

Cuando se trata de relaciones temporales, las filas pueden eliminarse por períodos de validez especificando dicho lapso mediante la cláusula VALID.

UPDATE

```
UPDATE <table_name>
 SET <column1> = <value1>, [ <column2> = <value2> ... ]
[ VALID <period> ]
[ BRANCHING [NOT] <event_name1> , [NOT] <event_name2> ... ]
[ WHERE condition ];
```

Al igual que en el caso de la eliminación de filas, se pueden realizar actualizaciones sobre períodos de validez. La cláusula VALID permite actualizar atributos sobre lapsos específicos.

SELECT

La sentencia SELECT permite obtener información de cualquiera de los tipos de tablas existentes. Su estructura es:

```
SELECT [ [DISTINCT|ALL] [<table_name>.]column_name
[ ,[DISTINCT|ALL] [<table_name>.]column_name ... ] *]
FROM    <table_name1>[table_alias]
 [,<table_name2 [table_alias]>...]
WHERE condition
[GROUP BY <column_name> [,<column_name>...]
[HAVING condition] ]
[ORDER BY <column_name> [DESC|ASC]
   [,<column_name> [DESC|ASC] ...] |
[ BRANCHING [NOT] <event_name> ,
   [ [NOT] <event_name> ...]
[NOCOALESCE]
[UNION | INTERSECT | MINUS ...]
```

En donde <condition> en la cláusula `WHERE` puede ser:

```
WHERE [A] <operador de comparación> [B][<operador lógico> ...]
```

Donde A y B pueden ser algunos de los siguientes: una columna perteneciente a las tablas especificadas en la cláusula `FROM`, una constante o una expresión y cuyos tipos de dato se rigen según la tabla que se describe a continuación, según el operador de comparación.

Comparaciones válidas

Tipo de datos [A]	Operador de Comparación	Tipo de datos [B]
Sql	=, <>, >, >=, <, <=	Sql
Date / Time / Timestamp	=, <>, >, >=, <, <=	Date / Time / Timestamp
Interval	=, <>, >, >=, <, <=	Interval
Period	=, <>	Period
Date / Time / Timestamp	Precedes	Date / Time / Timestamp
Period	Precedes	Period
Period	Precedes	Date / Time / Timestamp

Date / Time / Timestamp	Precedes	Period
Date / Time / Timestamp	Contains	Date / Time / Timestamp
Period	Contains	Period
Period	Contains	Date / Time / Timestamp
Date / Time / Timestamp	Overlaps	Date / Time / Timestamp
Period	Overlaps	Period
Date / Time / Timestamp	Meets	Date / Time / Timestamp
Period	Meets	Period

En los operadores de comparación que tienen como operandos elementos temporales las granularidades de A y B deben ser iguales.

Por ejemplo, si en una cláusula se especifica ingreso meets DATE '10/05/1998', la granularidad de ingreso debe tener como mínimo *year*.

Otros operadores de comparación existentes en BBTSQL2 pero que son válidos solo para constantes, columnas y expresiones que no sean de tipo temporal son:

```
[NOT] BETWEEN <value> AND <value>
[NOT] LIKE <pattern>
<column_name> IS [NOT] _NULL
[NOT] IN    ( <subquery>  | <list>)
[NOT] ANY ( <subquery> | <list>)
[NOT] ALL   ( <subquery> | <list>)
```

En donde <list> es un conjunto de valores separado por comas.

CREATE EVENT

Para crear eventos en la base de datos debe utilizar la sentencia CREATE EVENT. Esta sentencia crea la definición en el diccionario de la base del evento en especificado. Para espeficificar las acciones que deben llevarse a cabo cuando se señala dicho evento, debe utilizarse el comando CREATE ACTION.

CREATE EVENT <event_name> <event_description>

CREATE ACTION

Permite asociar acciones a un evento creado previamente mediante el comando CREATE EVENT. Es importante notar que las acciones son llevadas a cabo en el orden de creación.

```
CREATE ACTION FOR <event_name > IS
        INSERT... |
        DELETE... |
        UPDATE...
```

SIGNAL

Permite señalar a la base de datos indicando la ocurrencia o no de un evento previamente creado mediante la sentencia CREATE ACTION. Se debe destacar que no se indicará error en caso de no existir el evento señalado. La sintáxis es:

```
SIGNAL <event_name> =  T | F
```

Tipos de datos

Los tipos de datos `INTEGER`, `NUMBER(n,n)`, `VARCHAR(n)` y `VARCHAR2(n)` son aceptados por el motor. Adicionalmente se cuenta con los siguientes tipos temporales:

```
DATE <granularity>
TIME  <granularity>
TIMESTAMP <granularity>
PERIOD  <granularity>
INTERVAL <granularity>
```

Granularidad

La granularidades permitidas son `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND` y sus combinaciones válidas.

Para el tipo de dato `DATE` las granularidades posibles son: `YEAR`, `MONTH`, `DAY`, `YEAR TO MONTH`, `YEAR TO DAY`, `MONTH TO DAY`, siendo `YEAR TO DAY` el default.

Para el tipo de dato `TIME` la granularidad aceptada es: `HOUR`, `MINUTE`, `SECOND`, `HOUR TO SECOND`, `HOUR TO MINUTE`, `MINUTE TO SECOND` y por default se considera `HOUR TO SECOND`.

Para los tipos de dato `TIMESTAMP`, `PERIOD` e `INTERVAL` es: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `YEAR TO MONTH`, `YEAR TO DAY`, `YEAR TO HOUR`, `YEAR TO MINUTE`, `YEAR TO SECOND`, `MONTH TO DAY`, `MONTH TO HOUR`, `MONTH TO MINUTE`, `MONTH TO SECOND`, `DAY TO HOUR`, `DAY TO MINUTE`, `DAY TO SECOND`, `HOUR TO MINUTE`, `HOUR TO SECOND`, `MINUTE TO SECOND`. Si no se especifica ninguna de las anteriores el default es `YEAR TO SECOND`.

Constantes Temporales

Las constantes temporales se construyen con las siguientes funciones:

```
DATE ' <date_format> '  <granularity>
DATE NOW

TIME ' <time_format> '  <granularity>
TIME NOW

TIMESTAMP ' <timestamp_format> '  <granularity>
TIMESTAMP NOW

PERIOD [<temporal_string> | <temporal_string_function> | BEGINNING |NOW,
            <temporal_string> | <temporal_string_function> | FOREVER   | NOW]

INTERVAL '<temporal_string>' <granularity>
```

Según la granularidad los strings temporales se deben escribir con el siguiente formato

Granularidad	Formato

Year	'yyyy'
Year to month	'mm/yyyy'
Year to day	'dd/mm/yyyy'
Year to hour	'dd/mm/yyyy hh'
Year to minute	'dd/mm/yyyy hh:mi'
Year to second	'dd/mm/yyyy hh:mi:ss'
Month	'mm'
Month to day	'dd/mm'
Month to hour	'dd/mm hh'
Month to minute	'dd/mm hh:mi'
Month to second	'dd/mm hh:mi:ss'
day	'dd'
Day to hour	'dd hh'
Day to minute	'dd hh:mi'
Day to second	'dd hh:mi:ss'
Hour	'hh'
Hour to minute	'hh:mi'
Hour to second	'hh:mi:ss'
Minute	'mi'
Minute to second	'mi:ss'
Second	'ss'

EJEMPLOS

```
DATE '15/1999' YEAR TO MONTH

PERIOD [DATE '1/2/1993',DATE '28/2/1993']

PERIOD [TIME '10:00' HOUR TO MINUTE,
        TIME '10:45' HOUR TO MINUTE]

INTERVAL '10/5' MONTH TO DAY

INTERVAL '10' MONTH
```

Funciones Temporales

BEGIN y END

Obtiene el límite inferior y superior del período indicado respectivamente. Ambas retornan un valor de tipo <date_time>.

Sintaxis:

```
BEGIN(<periodo>)

END(<periodo>)
```

Ejemplos:

```
select BEGIN(valid(empleados)) from empleados;
```

FIRST y LAST

```
FIRST( <date_time>, <date_time> )
LAST( <date_time>, <date_time> )
```

Ambos devuelven un valor de tipo <date_time>.

INTERSECT

Esta función devuelve un valor de tipo <period> que representa al período de solapamiento entre los periodos pasados como argumentos.

```
INTERSECT(<period>, <period>)
```

Ejemplo:

```
select intersect(valid(empleados),
           period[date '1990' year, date '2010' year])
           from empelados;
```

VALID

Devuelve el período de validez de cada upla con una granularidad igual a la de la tabla utilizada como argumento.

```
VALID (<table_name>)
```

Ejemplo:

```
select valid(empleados) from empleados;
```

TRANSACTION

Devuelve el tiempo transaccional de cada upla, la granularidad es YEAR TO SECOND.

```
TRANSACTION (<table_name>)
```

Ejemplo:

```
select transaction(empleados) from empleados;
```

ABSOLUTE

Devuelve un el valor absoluto de un intervalo

```
ABSOLUTE(<intervalo>)
```

Ejemplo:

```
select absolute(begin(valid(locales)-end(valid(locales)))
from locales;
```

CAST

Devuelve el <date_time> en la granularidad indicada en <granularity>.

```
CAST(<date_time>,<granularity>)
```

Ejemplo:

```
select cast(begin(locales), year to month) from locales;
```

ADD

Add permite sumar una dos argumentos de tipo temporal y sus variantes se encuentran especificadas a continuación:

```
<date_time> + <interval>
```

Devuelve <date_time>

```
<period> + <interval>
```

Devuelve <period>

```
<interval> + <interval>
```

Devuelve <interval>

SUB

SUB es la función que permite subtraer fechas y períodos. Al igual que ADD acepta distintos tipos de argumentos y sus variantes se encuentran debajo.

```
<date_time> - <interval>
```

Devuelve <date_time>

```
<period> - <interval>
```

Devuelve <period>

```
<interval> - <interval>
```

Devuelve <interval>

```
<date_time> - <date_time>
```

Devuelve <interval>

MUL

MUL permite multiplicar un intervalo por una constante.

```
<interval> * <number>
```

Devuelve <interval>

Ejemplo:

```
select empid, empnombre
      from empleados
      where end(valid(empleados))-begin(valid(empleados))
                  * 3 > interval '5' year;
```

DIV

DIV permite dividir a un intervalo por una constante numérica o otro intervalo.

<interval> / <interval>

Devuelve <number>

<interval> / <number>

Devuelve <interval>

Lista de Palabras Reservadas

A continuación se encuentra una lista completa de las palabras reservadas en BBTSQL2.

```
Absolute        distinct        max             some
action          drop            meets           sum
all             end             min             table
and             event           minus           time
any             exists          minute          timestamp
as              first           month           to
asc             for             nocoalesce      transaction
avg             forever         now             valid
at              from            not             values
begin           granularity     null            varchar
beginning       group           number          varchar2
between         having          of              union
branching       hour            on              unique
by              in              or              update
cast            insert          order           where
char            integer         overlaps        with
contains        intersect       period          year
count           interval        precedes
create          into            primary
date            is              second
day             key             select
delete          last            set
desc            like            signal
```

Parámetros del Inicialización del Motor

El motor utiliza un archivo de parámetros cuyo nombre es `tsqldef`. En este, se especifican valores que gobernarán la respuesta y funcionamiento del motor de bases de datos temporales. A continuación, se explica cada uno de estos parámetros.

PORT

Este valor especifica el port de TCP que será utilizado para aceptar conexiones al servidor. El rango de valores posibles es de 1024 a 65535.

THREAD_TRACING

Indica si el servidor debe o no generar archivos de trace para cada thread que es iniciado. Por lo tanto se generará un archivo por cada nueva conexión que es aceptada por el servidor.

Los valores posibles son **YES** o **NO**.

SERVER_TRACING

Este parámetro es similar al anterior, pero se aplica al servidor directamente. Esto es, se le indica al servidor que genere un trace con los eventos ocurridos. Ejemplos de estos eventos pueden ser: nuevas conexiones, errores varios, creaciones de objetos, etc.

Los valores posibles son **YES** o **NO**.

SERVER_TRACE_OUTPUT

Indica si el trace generado por el servidor debe ser generado en un archivo o enviado a la salida standard.

Los valores posibles son el nombre de un archivo o la constante **SCREEN**.

THREAD_TRACE_OUTPUT

Idéntico al parámetro anterior, pero aplicado a los archivos de trace generados por las distintas conexiones.

HOST_NAME

Indica cual es el nombre del host sobre el cual corre el servidor de bases de datos temporales. Este debe coincidir con el nombre especificado en las propiedades para el protocolo TCP del servidor NT sobre el cual se encuentra instalado.

CONNECT_STRING

Este es el alias de SQL*Net o Net8 de Oracle utilizado para conectarse con la base de datos subyacente. Este alias puede encontrarse en el archivo de configuración de SQL*Net, bajo el directorio ORACLE_HOME/net80/admin y su nombre es tnsnames.ora.

USER
Nombre del esquema de la base de datos bajo el que se crearán los objetos solicitados por los clientes.

PASSWORD
Password del usuario indicado en el parámetro USER.

DICTIONARY_OWNER
Esquema bajo el cual fue creado el diccionario de la base de datos temporal. El diccionario de datos es manejado por el servidor, por lo tanto una conexión a la base de datos será hecha utilizando el usuario especificado en este parámetro.

DICTIONARY_OWNER_PASSWORD
Especifica la password del usuario indicado en el parámetro DICTIONARY_OWNER.

ERROR_MESSAGES
Indica cual es la ubicación del archivo de mensajes utilizado por el servidor. Este archivo debe ser encontrado durante la inicialización de lo contrario el servidor fallará durante esta etapa.

TEMP_DIR
El motor utiliza un directorio temporario para realizar operaciones con archivos que sirven para comunicación entre procesos y resultados temporarios. Este parámetro indica cual es el directorio que el servidor utilizará para tales operaciones.

CONSISTENCY_AGENT
Debido a que el modelo requiere de la reconstrucción de los "futuros posibles" ante posibles modificaciones en los valores que constituyen el presente de la información, el agente de consistencia es el encargado de mantener esta coherencia.

Este parámetro indica si el agente de consistencia debe o no ser inicializado. Es importante tener en cuenta que de no hacerlo, la coherencia de los datos no será mantenida provocando posibles resultados incorrectos. Esta información sería corregida la próxima que vez que el motor sea levantado con esta opción.

Los valores posibles son **YES** o **NO**.

CONSISTENCY_AT_STARTUP
Esta opción indica si el agente de consistencia debe o no ser lanzado durante la inicialización del motor. Si este parámetro es igual a **YES** el agente realizará una consistencia con la consiguiente demora en la inicialización.

CONSISTENCY_TIMEOUT
Especifica la frecuencia en unidades de tiempo en la que el agente debe ser ejecutado para mantener la coherencia de la información.

El valor debe ser indicado como un número entero.

CONSISTENCY_GRANULARITY
Indica cual es la unidad de tiempo que debe aplicarse al valor del parámetro CONSISTENCY_TIMEOUT.

Por ejemplo, si se especifica:

```
CONSISTENCY_TIMEOUT=10

CONSISTENCY_GRANULARITY=HOUR
```

En este caso, el agente será iniciado cada diez horas.

TRACE_LEVEL
Indica cual es el nivel de detalle que deben tener los archivos de trace. Los valores posibles para este parámetro son: 0, 1, 2 y 3.

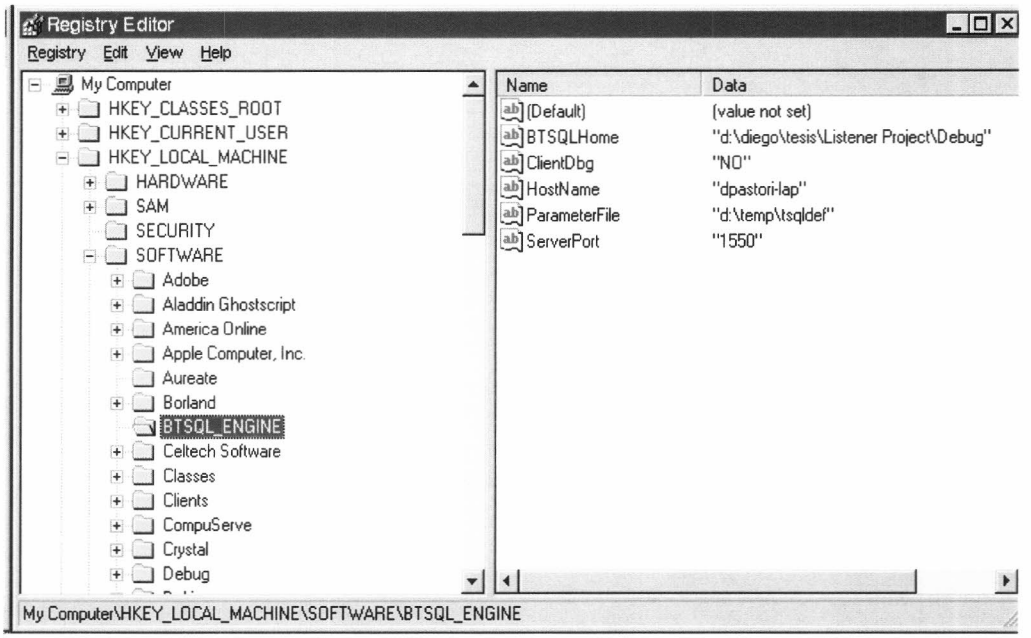
Si se especifica un valor 0, todos los mensajes serán escritos en el archivo de trace. Un valor 1 mostrará solo aquellos mensajes que son advertencias (posiblemente errores, pero que no son fatales para la ejecución del servidor). 2 mostrará solo aquellos mensajes que sean fatales y causen errores de ejecución en el servidor, provocando posibles fallas en su funcionamiento.

Parámetros del Registro de Windows NT/95/98/2000

El servidor de consultas temporales trabaja tanto sobre Windows NT 4.0 como Windows 95/98. Para el correcto funcionamiento deben configurarse una serie de parámetros en el registro (Registry). Estos permiten al servidor encontrar una serie de componentes necesarios para su inicialización.

El Registro

Las claves requeridas para la inicilización del servidor se encuentran a continuación.



Debajo se encuentra una explicación del significado de cada una de las claves como así también el valor necesario para la correcta configuración.

Localización de las claves

Como se ve en la figura, todas las claves correspondientes al servidor de consultas temporales como para los clientes se encuentran en HKEY_LOCAL_MACHINE\SOFTWARE\BTSQL_ENGINE.

BTSQLHome

Este String indica el directorio en donde se encuentra instalado el servidor de consultas temporales. No debe incluir la "/" final. Por ejemplo, un valor válido es:

```
d:\Program Files\Tesis\BBTSQL2
```

ClientDbg

Este valor indica si la librería dinámica debe o no generar información de debug. Este log se genera en un archivo de texto y cuyo nombre es client.dbg. Este archivo es creado en el directorio en el que se ejecuta la aplicación cliente. Los valores posibles son "YES" o "NO".

HostName

Indica el nombre por el cual se hace referencia al host en donde se ejecuta el servidor. Esta información puede encontrarse en el directorio C:\WINNT\system32\drivers\etc\hosts o a través de un DNS en caso de contar con uno en la red.

ParameterFile

Indica la ubicación del archivo de parámetros de inicialización del servidor de consultas temporales. Este valor no es necesario en las PCs en donde solo se utilizan aplicaciones clientes.

ServerPort

Indica el número de port de TCP que se utilizará para la comunicación inicial en donde el servidor de consultas acepta solicitudes de conexión. El rango de valores válidos es 1024-65535.

Si bien todos estos valores son creados y configurados por el programa de instalación, algunos deben ser modificados manualmente. Para esto, en Windows NT, pulsar el botón "Start" y luego en la ventana de "Run" ejecutar el comando "regedit".