**Álgebra de Tablas**

Versión simplificada de apunte del 2020 creado por Francisco Ziliani con aportes de Matías Bordone.

**Introducción**

La forma clásica de enseñar fundamentos teóricos de consultas de bases de datos es el Algebra Relacional (AR). En esencia, el AR es un conjunto de operaciones que describen el resultado de realizar una consulta en una base de datos relacional. Usualmente se la denomina de tipo operacional o procedimental, a diferencia del Cálculo Relacional (CR) que es de tipo declarativo. Sin embargo, a la hora de modelar fehacientemente el tipo de consultas que se realizan en una base de datos, tanto AR como CR tienen limitaciones.

Por ejemplo, el AR tiene como base la teoría de conjuntos, lo que no permite distinguir múltiples ocurrencias de un registro en una tabla, ni sobre el orden que tienen los registros, elementos que están muy presentes en las bases de datos reales. Respecto de la imposibilidad de tener registros duplicados, por ejemplo, los operadores de agrupamiento que permiten calcular valores en una tabla (sumatorias, promedios, etc.) son incorporados en el AR “por la ventana”, puesto que necesariamente necesitan operar sobre conjuntos con duplicados.

Por este motivo, aunque menos difundido en la literatura, existe también la versión de AR con multiconjuntos, es decir, conjuntos con duplicados. Pero esta definición también tiene una limitante: al no mantener el ordenamiento de los datos, no es posible aplicar optimizaciones que se beneficien de este conocimiento. Por ello, desde la cátedra de Bases de Datos nos proponemos redefinir los operadores del AR a través de operaciones sobre listas de tuplas en vez de conjuntos de tuplas, en una versión levemente inspirada por el trabajo [1].

Para esto vamos a hacer uso de los conocimientos de programación funcional sobre listas que se han utilizado en otras materias. Utilizaremos a lo largo de este documento la notación de Haskell. Aquí un breve repaso.

**Listas y sus operaciones**

Las listas se escriben como [a, b, c] o (a : b : c : []) para una lista con elementos a, b, y c (usamos la notación x: [] para indicar “x seguido de []”). En nuestro caso, los elementos serán tuplas o valores (Int, String, Float, etc.).

Funciones sobre listas pueden ser definidas por recursión.

**Ejemplo 1**: (suma de enteros de una lista)

1. suma :: [Int] -> Int
2. suma [] = 0
3. suma x : xs = x + suma xs

Generalizando esta recursión, podemos pensar en las siguientes ecuaciones para definir una función única f para cada constante c :: A y función h :: A -> A:

f [] = c

f (x : xs) = h x (f xs)

El esquema anterior se llama *definición por recursión estructural* sobre listas.

Las dos ecuaciones definidas anteriormente pueden ser capturadas por una función simple llamada *foldr* que toma la constante *c* y la función *h* como argumentos; o sea: f = foldr(h, c). La operación foldr se dice de alto orden porque el primer argumento es una función. La idea de foldr(h, c) es que reemplaza [] por *c*, y : por *h* y luego evalúa el resultado:

**Definición 1** (foldr):

1. foldr :: (a ->b ->b) -> b -> [a] -> b
2. foldr h, c [] = c
3. foldr h c (x : xs) = h x (foldr h c xs))

En la primera línea de la definición ponemos el tipo de la función. foldr toma tres argumentos. El primero es una función cuyo tipo es a -> b -> b, es decir, toma algo de un tipo polimórfico (que se puede instanciar con cualquier tipo) a. Este tipo corresponde al tipo de los elementos de la lista. Luego, toma algo de otro tipo polimórfico b, y retorna un elemento de este último tipo. Como segundo argumento, foldr toma un elemento de tipo b, que será la constante, y como tercer argumento toma la lista.



Figure 1: Despliegue de la función foldr

Podemos ver un ejemplo general de foldr f z [1, 2, 3, 4, 5] en la figura 1.

La función foldr es interesante porque permite definir de forma compacta muchas operaciones sobre listas. Como ejemplo, la suma de enteros de una lista anterior se puede definir en una sola línea (se desprende trivialmente de la recursión estructural anterior que *c* = 0 y *h* = +):

suma xs = foldr (+) 0 xs

Al ejecutar paso a paso con la lista [1,2,3], obtenemos la siguiente sucesión de pasos. Cada paso tiene antepuesto la ecuación utilizada:

Texto

Descripción generada automáticamente

Una función útil sobre listas es aquella que aplica una función *f* sobre todos los elementos de la lista. Tradicionalmente esta operación se llama map *f*.

Diagrama

Descripción generada automáticamente

**Definición 2** (ecuaciones de map):

map :: (a -> b) -> list a -> list b

map f [] = []

map f (x : xs) = f x : (map f xs)

Mirando estas ecuaciones como recursión estructural queda claro que:

c = []

h = (\x xs -> fx : xs)

Luego tenemos los elementos para definir map de manera compacta como foldr.

**Definición 3** (map):

map :: (a -> b) -> list a -> list b

map f = foldr (\x xs -> fx : xs) []

Definimos el operador de pertenencia, notándolo - ∊- para indicar que lo utilizaremos

Infijo. Tenemos las ecuaciones:

1. -∊- :: a -> [a] -> Bool
2. x ∊[] = False
3. x ∊ y : xs = x == y || x ∊ xs

Mirando estas ecuaciones como recursión estructural queda claro que:

*c* = False

*h* = (\a b -> a == x || b)

Luego tenemos los elementos para definir - ∊- de manera compacta como foldr.

**Definición 4** (Pertenece):

1. -∊- :: a -> [a] -> Bool
2. *x* ∊ *l* = foldr (\a b -> a == *x* || b) False *l*

Definimos la operación concatenación de listas:

1. ++ : [a] -> [a] -> [a]
2. [] ++ *l*‘ = *l’*
3. (y : *l’*) ++ *l* = y : (*l’* ++ *l*)

Mirando estas ecuaciones como recursión estructural queda claro que:

*c* = *l’*

*h* = :

Usando foldr:

*l* ++ *l’* = foldr (:) *l’* *l*

Una forma de testearla es evaluar esta definición para un caso particular.

[1,2] ++ [3,4]   
= {def ++} foldr : [3,4] 1:2:[]

= {def foldr 2} 1: foldr [3,4] 2:[]

= {def foldr 2} 1: (2: foldr [3,4] [])

= {def foldr 1} 1:(2: [3,4])

[1,2,3,4]

Para comprobar propiedades más interesantes vamos a necesitar el principio de inducción sobre listas.

**Definición 5** (Principio de inducción sobre listas). Sea *P* una propiedad sobre listas (notaremos

P(l) para indicar que *P* se cumple para la lista l). El principio de inducción sobre listas se define como:



Dicho en castellano, esta definición nos dice que para probar que toda lista l cumple la propiedad *P*, se debe probar que vale para la lista vacía (caso base), y también que asumiendo que vale para una lista *l’* (hipótesis inductiva) - vale también para la lista con un elemento *x* extra.

**Ejemplo 2**: Demostración que la concatenación de listas es asociativa.

*Prueba*:

Caso base: l = []: ([] ++ l) ++ l’ = (por ++ 2) l ++ l’ = (por ++ 2) [] ++ (l ++ l’).

La hipótesis inductiva es:

HI: (xs ++ l) ++ l’ = xs ++ (l ++ l’)

El paso inductivo es:

((x:xs) ++ l)++ l’ = (por ++ 3) (x : (xs ++ l)) ++ l’

= (por ++ 3) x : ((xs ++ l) ++ l’)

= (por HI) x: (xs ++ (l ++ l’))

= (por ++ 3 hacia atrás) (x: xs) ++ (l ++ l’)

Con este breve repaso de listas concluido, procedemos ya a hablar de cómo modelar las tablas de una base de datos.

**Tablas y sus esquemas:**

Nos interesa modelar el concepto de tabla, es decir, un elemento con la siguiente información:

Tabla

Descripción generada automáticamente

Donde los campos son nombres, y tienen un tipo τ que puede ser Int, String, DateTime, etc.

Para cada columna j, los valores *v*ij (con i ∊ {0, …, M}) deben tener el tipo τj . Cuando no sea importante, o se entiendan por contexto, omitiremos los tipos.

El encabezado de la tabla (exceptuando el nombre) es lo que llamaremos *esquema* y usualmente lo notaremos como:

**Definición 6** (Esquema de tabla):



Una tabla es, entonces, un esquema y una lista de tuplas que respetan ese esquema. Es decir, cada tupla de la tabla tiene tipo dom(R) = (τ1 x … x τN). Cuando el nombre de un campo no importe o sea anónimo, utilizaremos la notación \_:: τ. Cuando una tabla no tenga nombre (por ser producto de una operación, como veremos más adelante) vamos a omitir el nombre en el esquema. Otra forma que utilizaremos para notar que una tabla nombre tiene un esquema *R* es nombre :: *R*.

Luego, indicaremos como t ∊ *nombre* a una tupla de la tabla nombre. La tupla t en este caso representa un registro en la tabla. Vamos a notar las tuplas de la forma tradicional: (*v1, v2, … , vN*) es la tupla con *N* valores. Para obtener el valor de una tupla correspondiente a un campo en particular campoi utilizaremos la notación t.campoi .

**Ejemplo 3** (Tablas y tuplas): Las siguientes tablas determinan una base de datos de

clientes, donde cada cliente puede tener múltiples teléfonos.

Texto

Descripción generada automáticamente

**Definición 7** (Concatenación de tuplas). Sea *t* = (*a1, …, aN*) y *t’* = (*b1, …, bM*) dos tuplas de tablas con esquemas *R* y *S* respectivamente. Definimos la concatenación de las tuplas *t* y *t’* como:



El esquema de la concatenación de dos tuplas es la concatenación de los esquemas de las tuplas (*R* y *S*).

**Operadores**

En esta sección brindaremos los operadores del álgebra de tablas, que nos permitirán hacer consultas a una base de datos. Como ejemplo, vamos a estar utilizando las siguientes tablas, donde para facilitar a lectura le vamos a prefijar la primera letra del nombre de la tabla a los números que identifican registros:

Tabla

Descripción generada automáticamente

**Proyección generalizada**

La *proyección* nos permite obtener determinadas columnas de una tabla, opcionalmente realizando algún cálculo sencillo.

**Ejemplo 4** (Proyección). Supongamos que queremos obtener los legajos y el sueldo anual (incluyendo aguinaldo) del plantel docente:

Tabla

Descripción generada automáticamente

Como veremos, cuando se realiza un cálculo el nombre de la columna se “pierde”.

La proyección generalizada se puede definir usando ecuaciones.



Pero vemos que estas ecuaciones se corresponden con aquellas de map, luego usamos la siguiente definición:

Texto

Descripción generada automáticamente

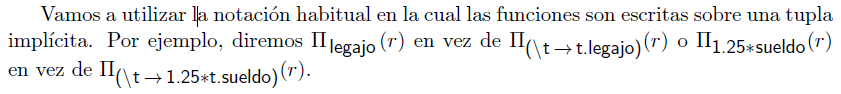
Es habitual que queramos extraer una cierta columna de una tabla, con lo que daremos un nombre especial a estas funciones:

**Definición 9** (proyector): Diremos que una función *f* :: dom(*R*) -> τi es un *proyector* si es de la forma (\t -> t.columna), donde columna :: τi es una de las columnas de R. Diremos en este caso que f *proyecta* a columna.

A partir de esta última definición establecemos que los nombres de los campos de la proyección serán:

Imagen que contiene Texto

Descripción generada automáticamente



**Selección**

La selección permite filtrar las tuplas de acuerdo a un criterio dado.

**Ejemplo 5** (Selección). De la siguiente forma obtenemos los cursos que enseña Patricia Selinger (por su número de legajo):

Tabla

Descripción generada automáticamente

Las ecuaciones de la selección se pueden definir del siguiente modo:

**Definición 10** (ecuaciones de selección.)



Mirando estas ecuaciones como recursión estructural queda claro que:

c = []

h = (\t r’ -> if p t then x : r’ else r’)

Esto nos permite definir en forma compacta selección usando foldr:

Gráfico, Texto

Descripción generada automáticamente con confianza media

Notar que podemos utilizar en p cualquier operador booleano (comparación, and/or, etc.).

**Producto cartesiano**

El producto cartesiano funciona como su homónimo en matemática: dadas dos tablas, junta cada registro de la primera con todos los registros de la segunda. Esto permite hacer consultas interesantes que involucren más de una tabla.

**Ejemplo 6** (Producto cartesiano). Si queremos obtener los nombres de les profes que dictan “Análisis de Datos", podemos hacer lo siguiente:



Desmenucemos la consulta, de adentro para afuera. Primero tenemos profe x curso. Como dijimos, esto asocia cada tupla de profe con todas las tuplas de curso, obteniendo el siguiente resultado:

Tabla

Descripción generada automáticamente

Para desambiguar la columna legajo agregamos al comienzo el nombre de la tabla (acortado por la primera letra en la figura). Como podemos notar, esta tabla temporal no tiene nombre. Luego el filtro de selección en σ nos pide dos cosas: igualar los legajos, y quedarnos con el curso específico. Aunque el filtro se aplica en simultáneo, vamos a hacerlo por partes para que se entienda mejor. Primero, igualamos los números de legajo, obteniendo:

Tabla

Descripción generada automáticamente

Luego nos quedamos sólo con el curso “Análisis de datos”:

Tabla

Descripción generada automáticamente

Finalmente, obtenemos el nombre de la profesora:

Imagen que contiene Texto

Descripción generada automáticamente

Primero definimos el producto cartesiano por medio de ecuaciones:

**Definición 11** (ecuaciones de producto cartesiano)

[] ⤫ s = []

(t : r) ⤫ s = anexar s t (r ⤫ s)



Tabla

Descripción generada automáticamente

Donde anexar es una función que “pega" una tupla dada t a todas las tuplas de la tabla s, concatenando el resultado a la tabla q:

Claramente, la función anexar que sigue esta idea se puede expresar usando map y concatenación de listas:

anexar s x q = map (\t -> (x ; t) ) s ++ q

Analizando la recursión estructural del producto cartesiano:

c = []

h = anexar s

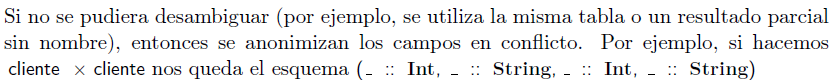
Luego, Podemos expresarlo con foldr:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente



**Reunión selectiva**

Cuando mostramos el operador ⤫ hicimos algo que es muy común: unir tablas que se referencian mediante el o los campos en común. En el ejemplo 6, unimos las tablas mediante el campo legajo. La *reunión selectiva* nos permite fácilmente hacer este tipo de operaciones:

**Ejemplo 7** (Reunión selectiva). Podemos escribir la consulta del ejemplo 6 del siguiente modo:



La reunión selectiva se describe por medio de los otros operadores. Es importante mencionar que de los campos que se igualan sólo quedan los de la tabla de la izquierda, evitando así tener redundancia.

Texto, Carta

Descripción generada automáticamente

**Reunión Natural**

La reunión natural nos permite hacer una reunión selectiva con todas las columnas que tengan igual nombre en ambas tablas, algo que es muy utilizado en la práctica. Por ejemplo, podemos simplificar aún más la consulta del ejemplo 6 haciendo:



**Definición 14** (reunión natural). Sean r(n1 :: τ1 ,…, nN :: τN) y s(m1 :: τ’1 ,…, mM :: τ’M), definimos la reunión natural de *r* y *s* como:



**Definiciones locales**

Cuando se hacen consultas complejas, vamos a utilizar el operador let, que funciona de forma similar a como funciona en Haskell.

Texto

Descripción generada automáticamente

**Definición 15** (let). Para facilitarnos un poco las cosas, definiremos el let utilizando la función anónima, y para distinguir fácilmente dónde termina la definición utilizaremos la terminación de línea (aunque en realidad no sea necesaria).

Imagen que contiene reloj

Descripción generada automáticamente

**Concatenación**

La concatenación de tablas se define como la concatenación de listas vista anteriormente. La única particularidad es que las tablas tienen que cumplir el siguiente requisito:

**Definición 17**. Se dice que dos esquemas son compatibles si tienen la misma cantidad de campos, con los mismos dominios. Vamos a decir que dos tablas son compatibles si sus esquemas lo son.

Texto

Descripción generada automáticamente

Notar que el resultado queda con el esquema de la tabla izquierda.

**Resta**

Ya en el ejercicio 9 mencionamos a la resta: \_esta permite quitar a una tabla las tuplas que existen en otra tabla. Las condiciones son iguales a las de la concatenación: los esquemas deben ser compatibles, y queda el esquema de la tabla izquierda. Su definición formal es la siguiente:

Texto, Word

Descripción generada automáticamente

Es decir, seleccionamos de r todas aquellas tuplas que no estén en s.

**Intersección**

La intersección permite dejar sólo las tuplas coincidentes entre las tablas.

Texto

Descripción generada automáticamente

**Renombramiento**

Como hemos visto, hay veces en que aplicar una operación nos hace perder los nombres de las columnas o las tablas. Por ello existe el operador de renombre, que sólo modifica el esquema de una tabla para darle un nuevo nombre, tanto a la tabla en sí como a los campos.

**Ejemplo 9** (Ejemplo de renombre). La siguiente consulta nos permite obtener los legajos de los profesores con mayor sueldo. Como es una consulta algo compleja, vamos a utilizar let:

Texto

Descripción generada automáticamente

Para entender lo que hace esta consulta, primero hacemos los renombres con el operador ρ, que en este caso cambia solo el nombre de la tabla por p1 y p2 (notar que en realidad con un solo renombre basta, pero para ayudar la lectura decidimos cambiar en ambos lados del ⤫). Luego de hacer el producto nos queda:

Tabla

Descripción generada automáticamente

Luego de seleccionar aquellos registros en donde p1.sueldo < p2.sueldo, nos queda:

Tabla

Descripción generada automáticamente

Es decir, queda Pierce tres veces por tener el menor sueldo, queda dos veces Codd por tener sueldo menor a Liskov y Selinger, y, lo más importante, no queda ningún registro con Selinger, puesto que es quien cobra más. Luego, al proyectar el legajo, nos quedarán solamente ocurrencias de p1, p3, y p4. Al quitar estas ocurrencias la primera columna de cliente (con el operador \), finalmente obtenemos el resultado:

Forma, Rectángulo

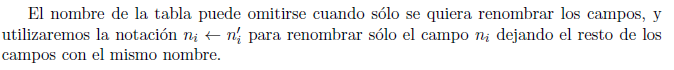
Descripción generada automáticamente

La definición de renombre es simple: sólo afecta el esquema de la tabla:

**Definición 16** (Renombre). Sea *r* tabla con *N* columnas de tipos τ1, …, τN; además, en el esquema de *r* una columna puede tener o no nombre. Se define el renombre de *r* así:

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza baja



**Remover duplicados**

La remoción de duplicados nos permite sacar aquellas tuplas que estén duplicadas en la tabla. Por ejemplo, si queremos obtener los legajos, sin duplicados, de les profes de los cursos, podemos hacer lo siguiente:



Las ecuaciones para remover duplicados son:

Luego tenemos que:

c = []

h = (\t s -> if t ∊ s then s else t : s)

Ahora estamos en condiciones de escribir la remoción de duplicados como foldr.

Texto

Descripción generada automáticamente



**Agregación**

En el ejemplo 9 calculamos el máximo de forma bastante engorrosa. En este apartado vamos a mostrar el operador de agregación, que permite calcular máximos, mínimos, etc. de las columnas de una tabla. Por ejemplo, una forma alternativa de escribir el ejemplo 6 usando agregación es del siguiente modo:

Texto

Descripción generada automáticamente

Es decir, primero calculamos el salario máximo de la tabla de profes, almacenando el resultado en la tabla *máximo\_salario*, para luego unirla con la tabla profe. Esto tiene el efecto de filtrar los profes que no ganen el máximo (puede desenrollar las definiciones para convencerse). Para poder unir la tabla, necesitamos primero hacer el renombre de la columna que contiene el máximo, así tiene el nombre salario.

**Definición 22** (funciones de agregación). Funciones de agregación son las siguientes funciones sobre listas:

*count*:: [a] -> integer, para contar la cantidad de elementos en la lista.

*sum*: para sumar los valores que deben ser numéricos.

*avg*: para promediar los valores que deben ser numéricos

*min*: para obtener el mínimo valor en la lista.

*max*: para obtener el máximo valor en la lista.

Ejercicio: definir las funciones de agregación sobre listas primero recursivamente y luego con foldr.

**Definición** **23** (Agregación). Vamos a aplicar las funciones de agregación a columnas de tablas. Dada una tabla *r*, con columna *n*, para decir que se debe aplicar la funcion de agregación *f* a la columna *n* usamos la notación: *f*(*n*). Más precisamente, sea *r*(n1:: τ1, …, nN::τN), definimos la agregación como:

*ϒ*f1(a1), …, fm(am) (r) :: (\_:: τ’1 ,…, \_::τ’m)

*ϒ*f1(a1), …, fm(am) (r) = [ (f1 (map (\t-> t.a1)r), …, fm (map (\t-> t.am) r ) ]

Donde *fi* es función de agregación, o *f*i es de la forma: *f* ∘ *v* , para *f* función de agregación (o sea, removemos duplicados y luego aplicamos la función de agregación).

**Agrupación**

Vimos que con la agregación podemos calcular valores sobre una columna. Aunque esto tiene su utilidad, en general vamos a querer algo más preciso: conocer el valor para un determinado grupo de valores.

**Ejemplo 11** (Agrupación). Supongamos que queremos obtener la mayor cantidad de cursos que dictan los docentes. Para esto, necesitaremos calcular la cantidad de cursos dictados por docente, y luego aplicar el máximo:



Luego de la primera consulta, obtenemos la siguiente tabla:

Tabla

Descripción generada automáticamente

Al tomar el máximo de la segunda columna (renombrada como *cant*), obtenemos el valor final: 2.

**Definición 25** (Agrupación). Definimos la agrupación utilizando la misma letra griega *ϒ*), pero precediendo la lista de columnas para agrupar:



Parece bastante complejo, pero en realidad es bastante simple: primero proyectamos las columnas de agrupación de *r* y le sacamos los duplicados. De este modo, tenemos identificados los grupos sobre los cuales hacer los cálculos *fi*. Luego, a cada tupla de la agrupación le anexamos el (único) resultado de la agregación sobre los registros de *r* filtrados para ese grupo en particular.

**Ordenamiento**

El ordenamiento nos permite ordenar la información de acuerdo a una lista de columnas.

**Definición 26** (Ordenamiento). Vamos a especificar el ordenamiento como un simple insertion sort. La función *insert* inserta la tupla *t* en la tabla *r* en la posición que le corresponde: a izquierda las tuplas menores y a derecha las mayores. Luego la ordenación O se define como la inserción ordenada de cada elemento.



El ordenamiento descendente se define como la reversa del ordenamiento.

**Definición 27** (Ordenamiento descendiente).

Texto

Descripción generada automáticamente

Cuando queramos ordenar toda la tabla omitiremos la lista de columnas.

**Bibliografía**

1. P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. SIGMOD Rec., 23(1):87-96, Mar. 1994.