DISEÑO DE BASES DE DATOS RELACIONALES:

OTROS SISTEMAS DE INFORMACIÓN

Tema 5

Manuel Ramos Cabrer

Índice

- Bases de datos basadas en objetos
- Sistemas de información semiestructurada (XML)
- Sistemas de recuperación de información

BASES DE DATOS BASADAS EN OBJETOS

Modelos de datos Objeto-Relacionales

- Extienden el modelo de datos relacional incluyendo orientación a objetos y construcciones para tipos de datos añadidos.
- Permiten que los atributos de las tuplas tengan tipos complejos, incluyendo valores no atómicos como pueden ser relaciones anidadas.
- Preservan los fundamentos relacionales, en particular el acceso declarativo a los datos, al mismo tiempo que extienden la capacidad de modelado.
- Compatibilidad hacia arriba con los lenguajes relacionales existentes.

Tipos de datos complejos

- Motivación:
  - Permitir dominios no atómicos (atómico ≡ indivisible)
    - Ejemplo de dominio no atómico: conjunto de enteros, o conjunto de tuplas.
  - Permitir un modelado más intuitivo para aplicaciones con datos complejos.
- Definición intuitiva:
  - Permite relaciones donde permitimos valores no atómicos (escalares) — relaciones dentro de relaciones.
  - Mantiene los fundamentos matemáticos del modelo relacional.
  - Viola la primera forma normal.

Ejemplo de una relación anidada

- Ejemplo: sistema de información de una biblioteca
- Cada libro tiene:
  - título,
  - una lista (array) de autores,
  - editorial, con subcampos *nombre* y *lugar*, y
  - un conjunto de palabras clave
- Relación *libros* no 1FN

titulo	*array_autores*	*editorial*	*conjunto_palabras_clave*
		(nombre, lugar)	
Compiladores	[Sánchez, Juárez]	(McGraw-Hill, Nueva York)	{intérpretes, análisis}
Redes	[Juárez, Fernández]	(Oxford, Londres)	{internet, web}

Tipos de datos complejos y SQL

- En SQL:1999 se introdujeron extensiones para soportar tipos de datos complejos:
  - Colecciones (*Collection*) y tipo de datos grandes
    - Las relaciones anteriores son un ejemplo de tipos colección.
  - Tipos estructurados
    - Estructuras de registros anidados como atributos compuestos
  - Herencia
  - Orientación a objetos
    - Incluyendo identificadores de objetos y referencias
- No está totalmente implementado en ningún SGBD actual
  - Aunque se pueden encontrar algunas características en los principales SGBD comerciales

Tipos estructurados y herencia en SQL

- En SQL se pueden declarar **tipos estructurados** (también denominados **tipos definidos por el usuario**)

```
create type Nombre as
        (nombre          varchar(20),
        papellido        varchar(20),
        sapellido        varchar(20) )
         final

        create type Direccion as
        (calle          varchar(20),
        ciudad          varchar(20),
        cpostal         varchar(5) )
         not final
```

  Nota: **final** y **not final** indican si se pueden crear subtipos.

- Los tipos estructurados se pueden utilizar para crear tablas con atributos compuestos

```
create table persona (
    nombre      Nombre,
    direccion   Direccion,
    fechaDeNacimiento date)
```

- Se utiliza notación con puntos para referenciar los componentes: *nombre.papellido*

Tipos estructurados

- **Tipos de columna definidos por el usuario**

  create type *TipoPersona* **as** (
  nombre Nombre,
  direccion Direccion,
  fechaDeNacimiento **date**)
  not final

- Ahora podemos crear una tabla cuyas columnas son un tipo definido por el usuario

  create table *cliente* **of** *TipoPersona*

- De forma alternativa, podemos utilizar **tipos de columna sin nombre**.

```
create table cliente_r (
        nombre      row(nombre  varchar(20),
                        papellido  varchar(20),
                        sapellido  varchar(20)),
        direccion   row(calle      varchar(20),
                        ciudad     varchar(20),
                        cpostal    varchar(5)),
        fechaDeNacimiento date)
```

Métodos

- Se pueden añadir declaraciones de métodos a los tipos estructurados.

  method *edadEn Fecha* (e*nFecha* **date**)

  returns interval year

- El cuerpo del método se define aparte.

  create instance method *edadEnFecha* (e*nFecha* **date**)

  returns interval year

  for *TipoPersona*

  begin

  return *enFecha* - **self**.*fechaDeNacimiento*;

  end

- Ahora podemos encontrar la fecha de cada cliente:

  select *nombre.papellido, edadEnFecha* (**current_date**)

  from *customer*

Funciones constructor

- Las **funciones constructor** se utilizan para crear valores de tipos estructurados
- P.e.

```
create function Nombre(nombre varchar(20), papellido varchar(20) , sapellido
varchar(20))
returns Nombre
begin
    set self.nombre = nombre;
    set self.papellido = papellido;
    set self.sapellido = sapellido;
end
```

- Para crear un valor de tipo *Nombre,* utilizamos

```
new Nombre('Juan', 'Sánchez', 'Pérez')
```

- Normalmente se utiliza en sentencias insert

```
insert into Clientes values
    (new Nombre('Juan', 'Sánchez', 'Pérez'),
     new Direccion('Mayor, 20', 'Madrid', '28001'),
     date '22/08/1960');
```

Herencia de tipos

- Supongamos que tenemos la siguiente declaración de tipo para persona:

```
create type Persona
    (nombre varchar(20),
    direccion varchar(20))
```

- Podemos utilizar herencia para definir los tipos alumno y docente

```
create type Alumno
  under Persona
  (titulacion      varchar(20),
   departamento    varchar(20))
create type Docente
  under Persona
  (salario         integer,
   departamento    varchar(20))
```

- Los subtipos pueden redefinir métodos utilizando **overriding method** en vez de **method** en la declaración del método.

Herencia múltiple

- SQL:1999 y SQL:2003 no soportan herencia múltiple
- Si nuestro sistema tiene extensiones que soporten herencia múltiple, podremos hacer:

  create type *Becario*
  under *Alumno, Docente*

- Para evitar conflictos entre las dos definiciones de *departamento*, podemos renombrarlo

  create type *Becario*
  under
  Alumno **with** (*departamento* **as** *alumno_dep*),
  Docente **with** (*departamento* **as** *docente_dep*)

- Cada valor debe tener una correspondencia a un **tipo más específico**

Herencia de tablas

- Las tablas creadas a partir de subtipos se pueden utilizar a su vez como **subtablas**
- P.e. **create table** *personas* **of** *Persona;*
  create table *alumnos* **of** *Alumno* **under** *personas;*
  create table *docentes* **of** *Docente* **under** *personas;*
- Las tuplas que se añadan a una subtabla son visibles automáticamente en las consultas sobre la supertabla
  - P.e. una consulta sobre *personas* también ve *alumnos* y *docentes*.
  - Igualmente, las actualizaciones/eliminaciones sobre *personas* también se traducen en actualizaciones/eliminaciones sobre las subtablas.
  - Para modificar este comportamiento, podemos utilizar "**only** *personas"* en la consulta.
- Conceptualmente, la herencia múltiple es posible con tablas
  - p.e. *becarios* bajo *alumnos* y *docentes*
  - *Pero actualmente SQL no lo soporta*
    - No podemos crear una persona (tupla en *personas*) que sea tanto alumno como docente.

Requisitos de consistencia para tablas

- Cada tupla de la supertabla (p.e. *personas)* se puede corresponder con, como mucho, una tupla en cada una de las subtablas (p.e. *alumnos* y *docentes)*
- Restricción adicional en SQL:1999:

  Todas las tuplas que se correspondan entre sí (es decir, con el mismo valor para los atributos heredados) se deben derivar de una tupla (introducida en una tabla).
  - Es decir, cada entidad debe tener un tipo más específico
  - No podemos tener una tupla en *personas* que se corresponda con tuplas tanto en *alumnos* como en *docentes*.

Arrays y conjuntos en SQL

- Ejemplo de declaración de array y conjunto:

```
create type Editorial as
    (nombre          varchar(20),
     lugar           varchar(20));
create type Libro as
    (titulo          varchar(20),
     array_autores   varchar(20) array [10],
     fecha_pub       date,
     editorial       Editorial,
     conjunto_palabras_clave varchar(20) multiset);
create table libros of Libro;
```

Creación de colecciones de valores

- Construcción de arrays:

  array [‘Suárez’,’Gómez’,’Lago’]

- Multiconjuntos

  multiset [‘ordenadores’, ‘bases de datos’, ‘SQL’]

- Para crear una tupla del tipo definido para la relación de libros:

  (‘Compiladores’, **array**[‘Suárez’,’Juárez’],
  new *Editorial* (‘McGraw-Hill’,’Nueva York’),
  multiset [‘intérpretes’,’análisis’])

- Para introducir esta tupla en la relación de libros:

  insert into *books*
  values
  (‘Compiladores’, **array**[‘Suárez’,’Juárez’],
  new *Editorial* (‘McGraw-Hill’,’Nueva York’),
  multiset [‘intérpretes’,’análisis’]);

Consulta de atributos con colecciones

- Para encontrar todos los libros que tengan la palabra “análisis” como palabra clave:

  select *titulo*
  from *libros*
  where ‘análisis’ **in** (**unnest**(*conjunto-palabras-clave*))

- Podemos acceder a elementos individuales de un array utilizando índices
  - P.e.: Si sabemos que un determinado libro tiene tres autores, podemos hacer:

    select *array_autores*[1], *array_autores*[2], *array_autores*[3]
    from *libros*
    where *titulo* = `Sistemas de Bases de Datos’

- Para obtener una relación conteniendo pares de la forma “titulo, nombre_autor” para cada libro y cada autor del libro:

  select *B.titulo, A.autor*
  from *libros* **as** *B*, **unnest** (*B.array_autores*) **as** *A* (*autor*)

- Para mantener el orden utilizamos la cláusula **with ordinality**

  select *B.titulo, A.autor, A.puesto*
  from *libros* **as** *B*, **unnest** (*B.array_autores*) **with ordinality as**
  A (*autor, puesto*)

Desanidado

- La transformación de una relación anidada en una forma con menos (o ningún) atributo de tipo relación se denomina **desanidado**.
- P.e.

  select *titulo*, *A* **as** *autor*, *editorial.nombre* **as** *edit_nombre*,
  editor.lugar **as** *edit_lugar*, *K.palabra_clave*
  from *libros* **as** *B*, **unnest**(*B.array_autores*) **as** *A* (*autor*),
  unnest (*B.conjunto_palabras_clave*) **as** *K* (*palabra_clave*)

- Relación resultante

titulo	*autor*	*edit_nombre*	*edit_lugar*	*palabra_clave*
Compiladores	Sánchez	McGraw-Hill	Nueva York	intérpretes
Compiladores	Juárez	McGraw-Hill	Nueva York	intérpretes
Compiladores	Sánchez	McGraw-Hill	Nueva York	análisis
Compiladores	Juárez	McGraw-Hill	Nueva York	análisis
Redes	Juárez	Oxford	Londres	internet
Redes	Fernández	Oxford	Londres	internet
Redes	Juárez	Oxford	Londres	web
Redes	Fernández	Oxford	Londres	web

Anidado

- Operación opuesta, creando atributos de tipo colección.
- El anidado se puede hacer de forma similar a la agregación, pero utilizando la función **colect**() en vez de la operación de agregación, para crear un multiconjunto.
- Para anidar la relación del ejemplo anterior (*libros_simple*) sobre el atributo *palabra_clave*:

  select *titulo*, *autor*, *Editorial* (*edit_nombre, edit_lugar*) **as** *editorial*,
  collect (*palabra_clave)* **as** *conjunto_palabras_clave*
  from *libros_simple*
  group by *titulo, autor, editorial*

- Para anidar tanto por autores como por palabras clave:

  select *titulo*, **collect** (*autor*) **as** *conjunto_autores*,
  Editorial (edit_*nombre, edit_lugar*) **as** *editorial*,
  collect (*palabra_clave*) **as** *conjunto_palabras_clave*
  from *libros_simple*
  group by *titulo*, *editorial*

Tipos referencia e identidad de objeto

- Si definimos un tipo *Departamento* con un campo *nombre* y un campo *director* que sea una referencia al tipo *Persona,* con la tabla *personas* como ámbito:

  create type *Departamento* (
  nombre **varchar** (20),
  director **ref** (*Persona*) **scope** *personas*)

  podemos crear la tabla *departamentos* de la siguiente forma:

  create table *departamentos* **of** *Departamento*

- Podemos omitir la declaración de **scope** de la declaración de tipo si la añadimos a la sentencia **create table**:

  create table *departamentos* **of** *Departamento*
  (*director* **with options scope** *personas*)

- La tabla referenciada debe tener un atributo que almacene el identificador, que se denomina **atributo autoreferencial**

  create table *personas* **of** *Persona*
  ref is *Id_persona* **system generated;**

Inicializando valores de tipo referencia

- Para crear una tupla con un valor referencia, podemos crear primero la tupla con una referencia nula y después actualizar la referencia:

```
insert into departamentos
        values ('IT', null)
update departamentos
    set director= (select p.Id_persona
                   from personas as p
                   where nombre = 'Juan')
    where nombre = 'IT'
```

Identificadores generados por el usuario

- El tipo de los identificadores de objeto se debe especificar como parte de la definición de tipo de la tabla referenciada, y
- La definición de la tabla debe especificar que la referencia es generada por el usuario

```
create type Persona
    (nombre varchar(20)
     direccion varchar(20))
   ref using varchar(20)
create table personas of Persona
  ref is Id_persona user generated
```

- Cuando se crea una tupla, debemos proporcionar un valor único para el identificador:

```
insert into personas (Id_persona, nombre, direccion) values
  ('01284567', 'Juan', 'Gran Vía, 23')
```

- Podemos utilizar el valor del identificador cuando introducimos una tupla en *departamentos*
  - Evita tener que hacer una consulta para obtener el identificador:

```
insert into departamentos
values('IT', '02184567')
```

Identificadores generados por el usuario

- Podemos utilizar una clave primaria ya existente como identificador:

```
create type Persona
        (nombre varchar (20) primary key,
         direccion varchar(20))
    ref from (nombre)
create table personas of Persona
    ref is Id_persona derived
```

- Cuando introducimos una tupla para *departamentos*, podemos hacer:

```
insert into departamentos
    values('IT','Juan')
```

Expresiones Path

- Encontrar los nombres y direcciones de los directores de todos los departamentos:

  select *director –>nombre*, *director –>direccion*
  from *departamentos*

- Las expresiones de la forma “director–>nombre” se denominan **expresiones path**.
- Las expresiones path ayudan a evitar joins explícitos
  - Si director de departamento no fuera una referencia, se necesitaría hacer un join de *departamentos* con *personas* para obtener la dirección.
  - Hace que sea mucho más sencillo expresar las consultas.

Implementando características O-R

- Similar a como traducimos las características E-A a esquemas de relación.
- Implementación de subtablas
  - Cada tabla almacena claves primarias y los atributos definidos en esa tabla

  o,

  - Cada tabla almacena tanto sus atributos locales como los atributos heredados.

Lenguajes de programación con persisitencia

- Lenguajes extendidos con construcciones para gestionar datos con persistencia.
- El programador puede manipular directamente los datos persistentes.
  - no es necesario trasladarlos a memoria y volver a guardarlos en disco (al contrario de SQL embebido)
- Objetos persistentes:
  - **Persistencia por clase** – declaración explícita de persistencia.
  - **Persistencia por creación** – sintaxis especial para crear objetos persistentes.
  - **Persistencia por marcado** – los objetos se hacen persistentes después de su creación.
  - **Persistencia por ámbito** – un objeto es persistente si se declara así explícitamente o puede ser utilizado por un objeto persistente.

Identidad de objetos y punteros

- Grados de permanencia de identidad de objetos
  - **Intraprocedimiento**: sólo durante la ejecución de un procedimiento individual.
  - **Intraprograma**: sólo durante la ejecución de un programa o consulta individual.
  - **Interprograma**: entre ejecuciones de programas, pero no si cambia el formato de almacenamiento en disco.
  - **Persistente**: interprograma, y además persistente a reorganizaciones de almacenamiento de datos.
- Existen diferentes versiones persistentes de C++ y Java:
  - C++
    - ODMG C++
    - ObjectStore
  - Java
    - Java Database Objects (JDO)

Sistemas C++ persistentes

- Extensiones del lenguaje C++ para soportar almacenamiento persistente de objetos.
- Varias propuestas, estándar ODMG propuesto, pero no demasiada actividad posterior
  - **punteros persistentes**: p.e. d_Ref<T>
  - **creación de objetos persistentes**: p.e. new (db) T()
  - **Extensiones de clase**: acceso a todos los objetos persistentes de una clase determinada.
  - **Relaciones:** Representadas por punteros almacenados en los objetos relacionados
    - Problema: consistencia de punteros
    - Solución: extensión del sistema de tipos para mantener referencias inversas automáticamente.
  - **Interface Iterator**
  - **Traducciones**
  - **Actualizaciones:** función *mark_modified()* para indicar al sistema que un objeto persistente que está en memoria ha sido modificado.
  - **Lenguaje de consulta**

Sistemas Java persistentes

- Estándar para añadir persistencia a Java : **Java Database Objects (JDO)**
  - Persistencia por ámbito.
  - Mejora del Bytecode
    - Las clases de declaran como persistentes de forma separada.
    - El programa modificador del bytecode modifica el bytecode de la clase para añadir persistencia.
      - P.e. Recupera los objetos bajo demanda.
      - Maca los objetos modificados para volver a escribirlos en la base de datos
  - Asociación de bases de datos
    - Permite almacenar objetos en una base de datos relacional.
  - Extensiones de clase
  - Tipo único de referencia
    - no hay diferencia entre un puntero a un objeto en memoria y un objeto persistente.
    - Técnica de implementación basada en **objetos hueco** (también denominado **trucaje de punteros**)

Mapping Objeto-Relacional

- **Object-Relational Mapping (ORM):** permiten escribir código sobre un modelo de datos orientado a objetos, pero almacena los datos en bases de datos relacionales tradicionales.
  - alternativa a las bases de datos orientadas a objetos y objeto-relacionales, que no han tenido éxito comercial.
- El implementador proporciona una correspondencia (*mapping*) entre objetos y relaciones.
  - Los objetos son totalmente transitorios, no hay identidad de objetos permanente.
  - p.e. La clase Java *Alumno* se corresponde con la relación *alumno*, con una determinada correspondencia de atributos.
  - Un objeto se puede corresponder con múltiples tuplas en múltiples relaciones.

Mapping Objeto-Relacional

- Los objetos se pueden recuperar de la base de datos
  - El sistema utiliza la correspondencia para recuperar los datos relevantes de las relaciones y constructores de objetos.
  - Las objetos modificados se vuelven a almacenar en la base de datos generando las sentencias update/insert/delete adecuadas.
  - Por ejemplo, los objetos se crean/almacenan en la base de datos con *session.save(object)*
    - se utiliza la correspondencia definida para crear las tuplas apropiadas en al base de datos.
- Limitaciones: sobrecarga, sobre todo para actualizaciones masivas.

Hibernate

- Se utiliza mucho el sistema ORM **Hibernate**
  - Es un sistema de software libre con versiones para muchos SGBD.
  - Proporciona una API para comenzar/terminar transacciones, recuperar objetos, etc.
  - Proporciona un lenguaje de consulta para realizar operaciones directamente sobre el modelo de objetos
    - Permite expresar consultas complejas, incluyendo joins.
    - Las consultas se traducen a SQL.
  - Permite modelar asociaciones mediante conjuntos asociados a los objetos
    - p.e. las materias cursadas por un alumno pueden ser un conjunto dentro del objeto Alumno.

Comparación de bases de datos O-O y O-R

- **Sistemas relacionales**
  - tipos de datos simples, lenguajes de consulta potentes, alta protección.
- **Sistemas objeto-relacionales**
  - tipos de datos complejos, lenguajes de consulta potentes, alta protección.
- **Basadas en lenguajes de programación persistentes**
  - tipos de datos complejos, integración con lenguajes de programación, altas prestaciones.
- **Sistemas de mapping objeto-relacional**
  - tipos de datos complejos integrados con lenguajes de programación, pero construidos como una capa sobre un sistema de bases de datos relacional.
- Nota: Muchos sistemas reales difuminan esta clasificación
  - P.e. lenguaje de programación persistente construido como un wrapper sobre una base de datos ofrece los dos primeros beneficios, pero tiene unas prestaciones bajas.

XML

Tratamiento de información semiestructurada

XML: Motivación

- Necesidad de intercambio de datos: necesidad crítica en el mundo interconectado actual.
  - Ejemplos:
    - Banca: transferencia de fondos
    - Procesamiento de pedidos (especialmente pedidos inter-compañía)
    - Datos científicos
      - Química: ChemML, …
      - Genética: BSML (Bio-Sequence Markup Language), …
  - El flujo de información en papel entre organizaciones se está reemplazando por flujo electrónico de información.
- Cada área de aplicación tiene su propio conjunto de estándares para representar la información.
- XML se ha convertido en la base para toda una nueva generación de formatos de intercambio de datos.

XML: Motivación

- La anterior generación de formatos estaban basados en texto plano con cabeceras de línea indicando el significado de cada campo
  - Similar conceptualmente a cabeceras de e-mail.
  - No permiten estructuras anidadas, no existe un lenguaje estándar de “tipos”.
  - Demasiado ligado a la estructura de bajo nivel del documentos (líneas, espacios, etc.)
- Cada estándar basado en XML define qué elementos son válidos utilizando
  - lenguajes de especificación de tipos XML para definir la sintaxis
    - DTD (Document Type Descriptors)
    - XML Schema
  - Descripciones textuales de la semántica.
- XML permite definir nuevas etiquetas según se requiera
  - No obstante, se puede restringir por DTDs.
- Existen muchas herramientas para analizar, navegar y consultar documentos/datos XML.

Comparación con datos relacionales

- Ineficiente: las etiquetas, que representan el esquema de la información, se repiten.
- Mejor que tuplas relacionales como formato de intercambio de datos
  - Al contrario que las tuplas relacionales, XML es autocontenido debido a la presencia de etiquetas.
  - Formato no rígido: se pueden añadir nuevas etiquetas
  - Permite estructuras anidadas
  - Gran aceptación, no sólo en sistemas de bases de datos, también en navegadores, herramientas y aplicaciones.

Motivación para anidamiento

- El anidamiento de datos es útil en la transferencia de datos
  - Ejemplo: elementos representando *item* anidados en un elemento *itemlist*
- El anidamiento no está soportado, o está desaconsejado, en bases de datos relacionales
  - Con varios pedidos, el nombre y la dirección del cliente se guardan de forma redundante.
  - La normalización sustituye las estructuras anidadas en cada pedido por una clave foránea a la relación que almacena el nombre y la dirección del cliente.
  - El anidamiento se soporta en bases de datos objeto-relacionales.
- Pero el anidamiento es apropiado cuando se transfieren datos
  - Una aplicación externa no tiene acceso directo a la información referenciada por una clave foránea.

Estructura de datos XML

- La mezcla de texto con subelementos es legal en XML.
  - Ejemplo:

```
<materia>
   Esta materia se ofrece desde 2012.
   <id_materia> BIO-399 </id_materia>
   <nombre>Biología Computacional </nombre>
   <nombre_depto>Biología </nombre_depto>
   <creditos> 3 </creditos>
</materia>
```

  - Útil para el marcado de documentos, pero desaconsejable para representación de datos.

Atributos vs. Subelementos

- Distinción entre subelemento y atributo
  - En el contexto de documentos, los atributos son parte del marcado, mientras que los subelementos son parte de los contenidos básicos del documento.
  - En el contexto de la representación de datos, la diferencia no está clara y puede ser confusa
    - La misma información se puede representar de las dos maneras
      - <materia id_materia= “IT-101”> … </materia>
      - <materia>
        <id_materia>IT-101</id_materia> …
        </materia>
  - Sugerencia: utilizar atributos para identificadores de elementos y utilizar subelementos para contenidos.

Consultando y Transformando datos XML

- Consulta de datos XML
- Traducción de información de un esquema XML a otro
- Estas dos operaciones están muy relacionadas y se gestionan con las mismas herramientas
- Lenguajes estándar de consulta/traducción de XML
  - XPath
    - Lenguaje simple consistente en expresiones de path.
  - XSLT
    - Lenguaje simple diseñado para traducir de XML a XML y de XML a HTML
  - XQuery
    - Un lenguaje de consulta XML con un conjunto amplio de características soportadas.

Modelo en árbol de datos XML

- Las consultas y transformaciones se basan en el **modelo en árbol** de los datos XML.
- Un documento XML se modela como un árbol, con **nodos** representando los elementos y atributos.
  - Los nodos elemento tienen nodos hijo, que pueden ser atributos o subelementos
  - El texto es un elemento modelado como un nodo de texto hijo de un elemento.
  - Los hijos de un nodo se ordenan en base a su orden en el documento XML.
  - Los nodos elemento y atributo (excepto para el nodo raíz) tienen un solo padre, que es un nodo elemento.
  - El nodo raíz tiene un solo hijo, que es el elemento raíz del documento.

XPath

- XPath se utiliza para direccionar (seleccionar) partes de documentos utilizando **expresiones path**.
- Una expresión path es una secuencia de pasos separados por "/"
  - Similar a nombres de fichero en una jerarquía de directorios.
- Resultado de una expresión path: conjunto de valores que junto con los atributos/elementos que los contienen cumplen el path especificado.
- P.e. /universidad-3/docente/nombre

  devolverá los docente de la universidad-3, por ejemplo:

  <nombre>Santiago</nombre>
  <nombre>Belén</nombre>
- P.e.. /universidad-3/docente/nombre/text( )

  devolverá los mismos nombres, pero sin las etiquetas.

Xpath

- La “/” inicial denota la raíz del documento.
- Las expresiones path se evalúan de izquierda a derecha
  - Cada paso opera sobre el conjunto de instancias resultado del paso anterior.
- Cualquier paso puede tener predicados de selección, expresados entre [ ]
  - P.e. /universidad-3/materia[creditos >= 4]
    - devuelve materias con 4 o más créditos.
    - /universidad3/materia[creditos] devuelve materias que tengan un subelemento créditos.
- A los atributos se accede con “@”
  - P.e. /universidad-3/materia[creditos >= 4]/@id_materia
    - devuelve los identificadores de materias con 4 o más créditos.

Funciones en XPath

- XPath proporciona diversas funciones
  - La función count() al final de un path cuenta el número de elementos en el conjunto generado por el path
    - P.e. /universidad-2/docente[count(./imparte/materia)> 2]
      - Devuelve los docentes impartiendo más de 2 materias.
  - También una función para comprobar la posición de un nodo respecto a sus hermanos.
- En los predicados se pueden utilizar las conectivas booleanas and y or y la función not().

Otras características de XPath

- El operador "|" implementa la operación de unión
  - P.e. /universidad-3/materia[@nombre_depto="Ing. telem."] | /universidad-3/materia[@nombre_depto="Biología"]
    - Devuelve la unión de las materias de Ingeniería Telemática y Biología.
    - "|" no se puede anidar dentro de otros operadores.
- "//" se puede utilizar para "saltar" varios niveles de nodos
  - P.e. /universidad-3//nombre
    - Encuentra cualquier elemento nombre en cualquier lugar bajo el elemento /universidad-3, independientemente del elemento que lo contenga.
- Un paso en el path puede avanzar al padre, hermanos, ascendientes y descendientes de los nodos generados en el paso anterior, no sólo a los hijos
  - "//" significa "todos los descendientes"
  - ".." especifica el padre.
- doc(nombre) devuelve la raíz del documento indicado.

XQuery

- XQuery es un lenguaje de consulta de propósito general para datos XML.
- Está estandarizado por el World Wide Web Consortium (W3C)
- XQuery se deriva del lenguaje de consulta Quilt, que a su vez se basa en SQL, XQL y XML-QL.
- XQuery utiliza clausulas

  for … let … where … order by …result …

  sintaxis:

  for ⇔ SQL **from**

  where ⇔ SQL **where**

  order by ⇔ SQL **order by**

  result ⇔ SQL **select**

  let permite variables temporales, y no tiene equivalente en SQL.

Sintaxis FLWOR en XQuery

- La cláusula For utiliza expresiones XPath, y las variables en la cláusula For toman valores del conjunto devuelto por XPath.
- Expresión FLWOR sencilla en XQuery
  - encontrar todas las materias con creditos > 3, y rodear cada resultado con las etiquetas <id_materia> .. </id_materia>

```
for  $x in /universidad-3/materia
let   $Idmateria := $x/@id_materia
where $x/creditos > 3
return <id_materia> { $Idmateria } </id_materia>
```

  - Los elementos en la cláusula **return** son texto XML a no ser que se encierren en {}, en cuyo caso se evalúan.
- La cláusula Let no es realmente necesaria en esta consulta, y la selección se puede hacer con XPath. La consulta se puede escribir como:

```
for $x in /universidad-3/materia[creditos > 3]
return <id_materia> { $x/@id_materia } </id_materia>
```

- Notación alternativa para construir los elementos:

```
return element id_materia { element  $x/@id_materia }
```

Joins

- Los joins se especifican de forma muy similar a SQL

```
for $c in /universidad/materia,
    $i in /universidad/docente,
    $t in /universidad/imparte
where $c/id_materia= $t/id_materia and $t/IID = $i/IID
return <docente_materia> { $c $i } </docente_materia>
```

- La misma consulta se puede realizar con selecciones expresadas en XPath:

```
for $c in /universidad/materia,
    $i in /universidad/docente,
    $t in /universidad/imparte[ $c/id_materia= $t/id_materia
                                and $t/IID = $i/IID]
return <docente_materia> { $c $i } </docente_materia>
```

Consultas anidadas

- La siguiente consulta convierte datos de una estructura plana a una estructura anidada:

```
  <universidad-1>
{    for $d in /universidad/departamento
      return <departamento>
                 { $d/* }
                 { for $c in /universidad/materia[nombre_depto = $d/nombre_depto]
                   return $c }
               </departamento>
}
{    for $i in /universidad/docente
     return  <docente>
                 { $i/* }
                 { for $c in /universidad/imparte[IID = $i/IID]
                   return $c/id_materia }
              </instructor>
}
</universidad-1>
```

- $c/* denota todos los hijos de un nodo al que está ligado $c, sin las etiquetas que los rodean.

Agrupamiento y agregación

- Las consultas anidadas se utilizan para agrupar

```
for $d in /universidad/departamento
return
      <departamento-salario-total>
          <nombre_depto> { $d/nombre_depto } </nombre_depto>
           <salario_total> { fn:sum(
               for $i in /universidad/docente[nombre_depto = $d/nombre_depto]
                return $i/salario
              ) }
           </salario_total>
       </departamento-salario-total>
```

Ordenación en XQuery

- La cláusula **order by** se puede utilizar al final de cualquier expresión. P.e. para devolver los docentes ordenados por nombre:

```
 for $i in /universidad/docente
order by $i/nombre
return <docente> { $i/* } </docente>
```

- usaremos **order by** $i/nombre **descending** para ordenar de forma descendiente.
- Podemos ordenar a varios niveles de anidamiento (ordenar departamentos por nombre_depto, y por materias ordenadas por id_materia en cada departamento)

```
<universidad-1> {
for $d in /universidad/departamento
order by $d/nombre_depto
return
    <departamento>
       { $d/* }
       { for $c in /universidad/materia[nombre_depto = $d/nombre_depto]
         order by $c/id_materia
         return <materia> { $c/* } </materia> }
    </departamento>
} </universidad-1>
```

Funciones y otras características de XQuery

- Funciones definidas por el usuario con los tipos del sistema de XMLSchema

```
declare function local:materias_depto($iid as xs:string)
        as element(materia)*
{
    for $i in /universidad/docente[IID = $iid],
        $c in /universidad/materia[nombre_depto = $i/nombre_depto]
    return $c
}
```

- Los tipos son opcionales para los parámetros y los valores devueltos.
- El * inidca una secuencia de valores de un determinado tipo.
- Podemos utilizar los cuantificadores existencial y universal en los predicados de la cláusula Where
  - **some** $e **in** *path* **satisfies** *P*
  - **every** $e **in** *path* **satisfies** *P*
  - Añadimos **and fn:exists($e)** para evitar que el $e vacío satisfaga la cláusula **every**.
- XQuery también soporta cláusulas If-then-else.

Almacenamiento de datos XML

- Los datos XML se pueden almacenar en
  - Almacenes de datos no relacionales
    - Ficheros planos
      - Natural para almacenamiento de XML
      - Pero tienen todos los problemas de este tipo de ficheros (no concurrencia, no transacciones, …)
    - Sistemas de Bases de datos XML
      - Bases de datos construidas específicamente para almacenar datos XML, soportando el modelo DOM y consultas declarativas.
      - No existen aún sistemas a nivel comercial.
  - Bases de datos relacionales
    - Los datos se deben transformar a formato relacional.
    - Ventaja: sistemas de bases de datos maduros.
    - Desventajas: sobrecarga de transformar datos y consultas.

Almacenamiento de XML en bases de datos relacionales

- Alternativas:
  - Representación mediante Strings.
  - Representación en árbol.
  - Mapeado a relaciones

Representación mediante Strings

- Se almacena cada elemento de primer nivel como un campo de tipo String de una tupla en una base de datos relacional
  - Utilizar una sola relación para almacenar todos los elementos, o
  - Utilizar una relación separada para cada tipo de elemento de primer nivel
    - P.e. relaciones materia, departamento, alumno, docente, …
      - Cada una con un atributo String que almacena el elemento.
- Indexado:
  - Almacenar los valores de los subelementos/atributos a indexar como campos extra de la relación, y construir índices sobre esos campos
    - P.e nombre_materia o Id_alumno.
  - Algunos sistemas de bases de datos soportan **funciones índice,** que utilizan el resultado de una función como valor clave.
    - La función debe devolver el valor del subelemento/atributo requerido.

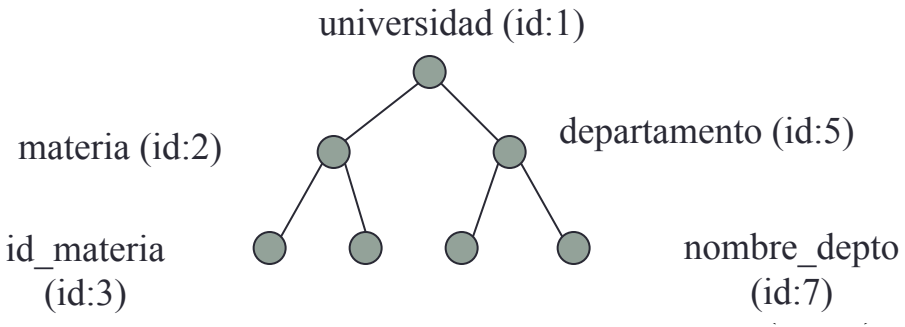
Representación mediante Strings

- Beneficios:
  - Puede almacenar cualquier dato XML, aún sin DTD.
  - Si hay muchos elementos de primer nivel en un documento, los Strings son pequeños comparados con el documento completo
    - Permite el acceso rápido a elementos individuales.
- Desventajas: Necesidad de analizar los Strings para acceder a valores dentro de los elementos
  - El análisis es lento.

Representación en árbol

- **Representación en árbol:** modelar los datos XML como un árbol y almacenarlo utilizando relaciones

  nodos(id, id_padre, tipo, etiqueta, valor)



- A cada elemento/atributo se le asigna un identificador único.
- El tipo indica elemento/atributo.
- La etiqueta indica el nombre de la etiqueta de elemento/atributo.
- El valor es el valor de texto del elemento/atributo.
- Se puede añadir un atributo extra *posicion* para almacenar el orden de los hijos.

Representación en árbol

- Beneficios: Se puede almacenar cualquier dato XML, incluso sin DTD
- Desventajas:
  - Los datos se dividen en demasiadas partes, incrementando la sobrecarga de espacio.
  - Aún las consultas más simples requieren muchas operaciones de join, lo que puede ser lento.

Mapeado de datos XML a relaciones

- Se crea una relación para cada tipo de elemento cuyo esquema se conoce:
  - Un atributo id para almacenar un identificados único para cada elemento.
  - Un atributo de relación correspondiente a cada atributo de elemento.
  - Un atributo id_padre para referenciar el elemento padre
    - Como en la representación en árbol.
    - También se puede almacenar información de posición de los hijos.
- Todos lo subelementos que aparecen una sola vez se pueden transformar en atributos de la relación
  - Para subelementos de texto, se puede guardar el texto como valor de un atributo.
  - Para subelementos complejos, se puede almacenar el *id* del subelemento.
- Los subelementos que pueden aparecer varias veces se representan en una relación separada
  - Similar a lo que se hace con atributos multivalorados cuando transformamos diagramas E-A a relaciones.

Almacenamiento de datos XML en base de datos relacionales

- Aplicando las ideas anteriores sobre un documento XML de una universidad con departamentos y materias anidadas, obtenemos:
  departamento(*id*, *nombre_depto*, edificio, *presupuesto*)
  materia(*id_padre*, *id_materia*, *nombre_depto*, *nombre*, *creditos*)
- **Publicación**: proceso de convertir datos relacionales a un formato XML.
- **Poda**: proceso de convertir un documento XML en un conjunto de tuplas a introducir en una o más relaciones.
- Los sistemas de bases de datos con soporte para XML soportan la publicación y poda automática.
- Muchos sistemas ofrecen *almacenamiento nativo* de datos XML utilizando del tipo de dato **xml**. Se utilizan índices y estructuras de datos internas especiales por motivos de eficiencia.

SQL/XML

- Nueva extensión estándar de SQL que permite la generación de XML anidado
  - Cada tupla se corresponde con un elemento fila (row) XML

```
<universidad>
 <departamento>
    <row>
       <nombre_depto>Ing. Telem.</nombre_depto>
       <edificio>Teleco</edificio>
       <presupuesto> 100000 </presupuesto>
    </row>
   …. más filas si hubiera más tuplas…
 </departamento>
 … otras relaciones ..
</universidad>
```

Extensiones SQL

- **xmlelement** crea elementos XML.
- **xmlattributes** crea atributos.

```
select xmlelement (name "materia",
    xmlattributes (id_materia as id_materia, nombre_depto
as nombre_depto),
    xmlelement (name "nombre", nombre),
    xmlelement (name "creditos", creditos))
from materia
```

- Xmlagg crea un bosque de elementos XML

```
select xmlelement (name "departamento",
        nombre_depto,
        xmlagg (xmlforest(id_materia)
                order by id_materia))
from materia
group by nombre_depto
```

SISTEMAS DE RECUPERACIÓN DE INFORMACIÓN

Tratamiento de información no estructurada

Sistemas de Recuperación de Información

- Los sistemas de **Recuperación de Información (RI)** utilizan modelos de datos más simples que los sistemas de bases de datos.
  - La información se organiza como un conjunto de documentos.
  - Los documentos no tienen estructura, no tienen esquema.
- Los sistemas de recuperación de información encuentran qué documentos son relevantes en base a indicaciones del usuario, tales como palabras clave o documentos de ejemplo
  - P.e., encontrar los documentos que contengan las palabras "bases de datos"
- Se pueden utilizar también sobre descripciones textuales (metadatos) de datos no textuales, como pueden ser imágenes.
- Los ejemplos más conocidos de sistemas de recuperación de información son los motores de búsqueda en la Web.

Sistemas de Recuperación de Información

- Diferencias con sistemas de bases de datos
  - Los sistemas RI no consideran actualizaciones transaccionales (incluyendo control de concurrencia y recuperación)
  - Los sistemas de bases de datos trabajan con datos estructurados, con esquemas que definen la organización de los datos.
  - Los sistemas RI consideran algunos aspectos relativos a consulta de los que normalmente no se ocupan los sistemas de bases de datos
    - Búsqueda aproximada por palabras clave.
    - Ordenación de las respuestas en base a grados estimados de relevancia.

Búsqueda de palabras clave

- En las búsquedas de **texto completo**, todas las palabras de cada documento se consideran palabras clave.
  - Utilizamos la palabra **término** para referirnos a las palabras de un documento.
- Los sistemas de recuperación de información normalmente permiten expresiones de consulta formadas por palabras clave y las conectivas lógicas *y, o,* y *no*.
  - *"Y"* se supone implícito, aun cuando no se especifique explícitamente.
- La ordenación de documentos en base a la relevancia estimada es un factor crítico.
  - La ordenación por relevancia se basa en factores tales como
    - Frecuencia de términos
      - Frecuencia de ocurrencia de las palabras clave de la consulta en el documento.
    - Frecuencia en documentos inversa
      - ¿En cuántos documentos aparecen las palabras clave de la consulta?
        - Menor ➔ dar más importancia a la palabra clave.
    - Hiperenlaces a documentos
      - Más enlaces a un documento ➔ el documento es más importante.

Ordenación por relevancia utilizando términos

- Ordenación **TF-IDF** (*Term frequency/Inverse Document frequency*) :
  - Dado $n(d)$ = número de términos en el documento d
  - $n(d, t)$ = número de ocurrencias del término t en el documento d.
  - Relevancia de un documento d para un término t

$$TF(d,t) = log\left(1 + \frac{n(d,t)}{n(d)}\right)$$

    - El factor log evita un excesivo peso para términos frecuentes
  - Relevancia de un documento para una consulta Q

$$r(d,Q) = \sum_{t \in Q} \frac{TF(d,t)}{n(t)}$$

Ordenación por relevancia utilizando términos

- La mayoría de los sistemas realizan mejoras sobre el modelo anterior
  - A las palabras en el título, lista de autores, cabeceras de sección, etc. se les da una mayor importancia.
  - A las palabras que aparecen más cerca del final del documento se les da menos importancia.
  - Se eliminan las palabras muy comunes, como “un”, “una”, “el”, “y” etc.
    - Se denominan **palabras de parada** (*stop words*)
  - **Proximidad**: si las palabras clave de la consulta aparecen cerca unas de otras en el documento, el documento tiene mayor importancia que si aparecen separadas
- Los documentos se devuelven en orden decreciente de valor de relevancia
  - Normalmente se devuelven los mejor valorados, no todos.

Recuperación basada en similitud

- Recuperar documentos similares a un documento dado
  - La similitud se puede definir en base a palabras comunes
    - P.e., encontrar los *k* términos en A con mayor $TF(d, t)/n(t)$ y utilizar esos términos para calcular la relevancia de otros documentos.
- **Realimentación de relevancia**: La similitud se puede utilizar para refinar la respuesta a una consulta por palabras clave
  - El usuario selecciona algunos documentos relevantes de los obtenidos mediante la consulta por palabras clave, y el sistema busca otros documentos similares a esos.
- Modelo de espacio vectorial: define un espacio *n*-dimensional, donde *n* es el número de palabras en el conjunto de documentos
  - El vector para el documento *d* va desde el origen a un punto cuya coordenada *i* es $TF(d,t)/n(t)$
  - El coseno del ángulo entre los vectores de dos documentos se utiliza como una medida de similitud.

Relevancia utilizando Hiperenlaces

- El número de documentos relevantes para una consulta puede ser enorme si sólo se tienen en cuenta frecuencias de términos.
- La utilización de frecuencias de términos hace que sea fácil el “spamming”
  - P.e., una agencia de viajes puede añadir muchas ocurrencias de la palabra “viaje” a su página para tener puestos altos en las ordenaciones
- La mayoría de las veces la gente busca páginas de sitios populares.
- Idea: utilizar la popularidad de un Web (p.e., cuántas personas lo visitan) para ordenar las páginas que contienen las palabras clave
- Problema: dificultad para saber la popularidad real de un Web

Relevancia utilizando Hiperenlaces

- Solución: utilizar el número de hiperenlaces a ese Web como una medida de su popularidad o **prestigio**.
  - Contar sólo un hiperenlace por cada sitio.
  - La medida de popularidad es por sitio Web, no por página individual
    - Aunque, la mayoría de hiperenlaces son a la página principal del sitio.
    - Concepto de “sitio Web” difícil de definir, ya que un prefijo URL como www.uvigo.es contiene muchas páginas no relacionadas y con diferentes popularidades
- Mejoras
  - Cuando se calcula el prestigio de un sitio en base a enlaces, dar más peso a enlaces que tienen a su vez un prestigio alto
    - La definición es circular
    - Plantear y resolver un sistema de ecuaciones lineales simultáneas.
  - Esta idea es la base del mecanismo de ordenación **PageRank** de Google.

Relevancia utilizando Hiperenlaces

- Relacionado con las teorías de **redes sociales** que calculan el grado de prestigio de personas
  - P.e., el presidente de EEUU tiene un prestigio alto, ya que mucha gente lo conoce.
  - Alguien conocido por mucha gente con prestigio tiene un prestigio alto.
- Ordenación basada en portales y autoridades
  - un **portal** es una página que almacena enlaces a muchas otras páginas (sobre un tema)
  - Una **autoridad** es una página que contiene la información real sobre un tema.
  - A cada página se le asigna un **prestigio de portal** en base al prestigio de los portales que la referencian.
  - A cada página se le asigna un **prestigio de autoridad** en base al prestigio de las autoridades que la referencian.
  - De nuevo, las definiciones de prestigio son cíclicas, y se obtienen resolviendo ecuaciones lineales.

Sinónimos y Homónimos

- **Sinónimos**
  - P.e., documento: “reparar motocicletas”, consulta: “arreglar motocicletas”
    - Necesitamos detectar que “arreglar” y “reparar” sin sinónimos.
  - El sistema puede extender la consulta como “motocicletas *and* (reparar *or* arreglar)”
- **Homónimos**
  - P.e., “vino” tiene distintos significados como nombre/verbo.
  - Podemos desambiguar significados (hasta cierto punto) a partir del contexto.
- Extender consultas de forma automática a partir de sinónimos puede ser problemático
  - Necesitamos entender el significado pretendido para buscar sinónimos
    - O bien verificar los sinónimos con el usuario.
  - Los sinónimos, a su vez, pueden tener otros significados.

Consultas basadas en concepto

- Idea
  - Para cada palabra, determinar el **concepto** que representa a partir del contexto
  - Utilizar un a o varios **ontologías**:
    - Estructuras representando las relaciones entre conceptos
    - Configuran una representación semántica de la información
- Esta aproximación se puede utilizar para estandarizar la terminología de un determinado campo.
- Las ontologías pueden enlazar varios idiomas.
- Base de la **Web Semántica**.

Indexado de Documentos

- Un índice invertido relaciona cada palabra clave K_i con un conjunto de documentos S_i que contienen la palabra clave
  - Documentos identificados mediante identificadores
- El índice invertido puede almacenar
  - Posiciones de la palabra clave en el documento para permitir ordenaciones basadas en proximidad.
  - Contadores del número de apariciones de la palabra clave para calcular TF
- operación **and**: Encontrar los documentos que contienen todos los K_1, K_2, ..., K_n.
  - Intersección $S_1 \cap S_2 \cap ..... \cap S_n$
- operación **or**: Encontrar los documentos que contienen al menos uno de K_1, K_2, …, K_n
  - Unión, $S_1 \cup S_2 \cup ..... \cup S_n$,.
- Cada S_i se almacena ordenado para poder hacer las operaciones de unión/intersección de forma eficiente.
  - "**not**" también se puede implementar de forma eficiente a partir de listas ordenadas.

Medidas de efectividad de la recuperación

- Los sistemas de recuperación de información ahorran espacio utilizando estructuras de índice que soportan sólo recuperación aproximada. Esto puede dar lugar a:
  - **Falsos negativos** – no se recuperan algunos documentos relevantes.
  - **Falsos positivos** – se recuperan algunos documentos irrelevantes.
  - Para muchas aplicaciones un buen índice no debería permitir ningún falso negativo, pero puede permitir unos pocos falsos positivos.
- Métricas relevantes de prestaciones:
  - **precisión** – qué porcentaje de los documentos recuperados son relevantes para la consulta.
  - **recall** - qué porcentaje de los documentos relevantes para la consulta se recuperaron.

Medidas de efectividad de la recuperación

- Balance recall vs. precisión:
  - Podemos incrementar el recall recuperando muchos documentos (bajando hasta un nivel de relevancia bajo), pero obtendremos muchso documentos irrelevantes, reduciendo la precisión.
- Medidas de efectividad de la recuperación:
  - Recall como una función del número de documentos obtenidos, o
  - **Precisión como una función de recall**
    - De manera equivalente, como una función del número de documentos obtenidos.
  - P.e., “precisión del 75% a un recall del 50%, y 60% a un recall del 75%”
- Problema: Qué documentos son realmente relevantes y cuáles no.

Motores de búsqueda Web

- Las **arañas Web** son programas que localizan y recolectan información en la Web
  - Siguen hiperenlaces de documentos conocidos de forma recursiva para encontrar otros documentos
    - Comenzando por un conjunto *semilla* de documentos.
  - Documentos encontrados
    - introducidos en un sistema de indexado.
    - Se pueden descartar después de indexarlos, o bien se pueden almacenar como una copia *cache*
- Recorrer toda la Web puede llevar mucho tiempo
  - Los motores de búsqueda normalmente cubren una parte de la Web, no toda
  - Realizar un recorrido simple lleva meses.

Arañas Web

- El recorrido se realiza con muchos procesos en muchas máquinas, ejecutándose en paralelo
  - Conjunto de enlaces a ser recorridos se almacena en una base de datos.
  - Los nuevos enlaces encontrados se añaden a este conjunto para recorrerlos más tarde.
- El proceso de indexado también se ejecuta en muchas máquinas
  - Se crea una nueva copia del índice en vez de modificar el viejo.
  - El índice viejo se utiliza para contestar consultas.
  - Después de que un recorrido se “completa”, el nuevo índice pasa a ser el índice viejo.
- Se utilizan muchas máquinas para contestar consultas
  - Los índices se pueden mantener en memoria.
  - Las consultas se pueden redirigir a distintas máquinas para balancear la carga.

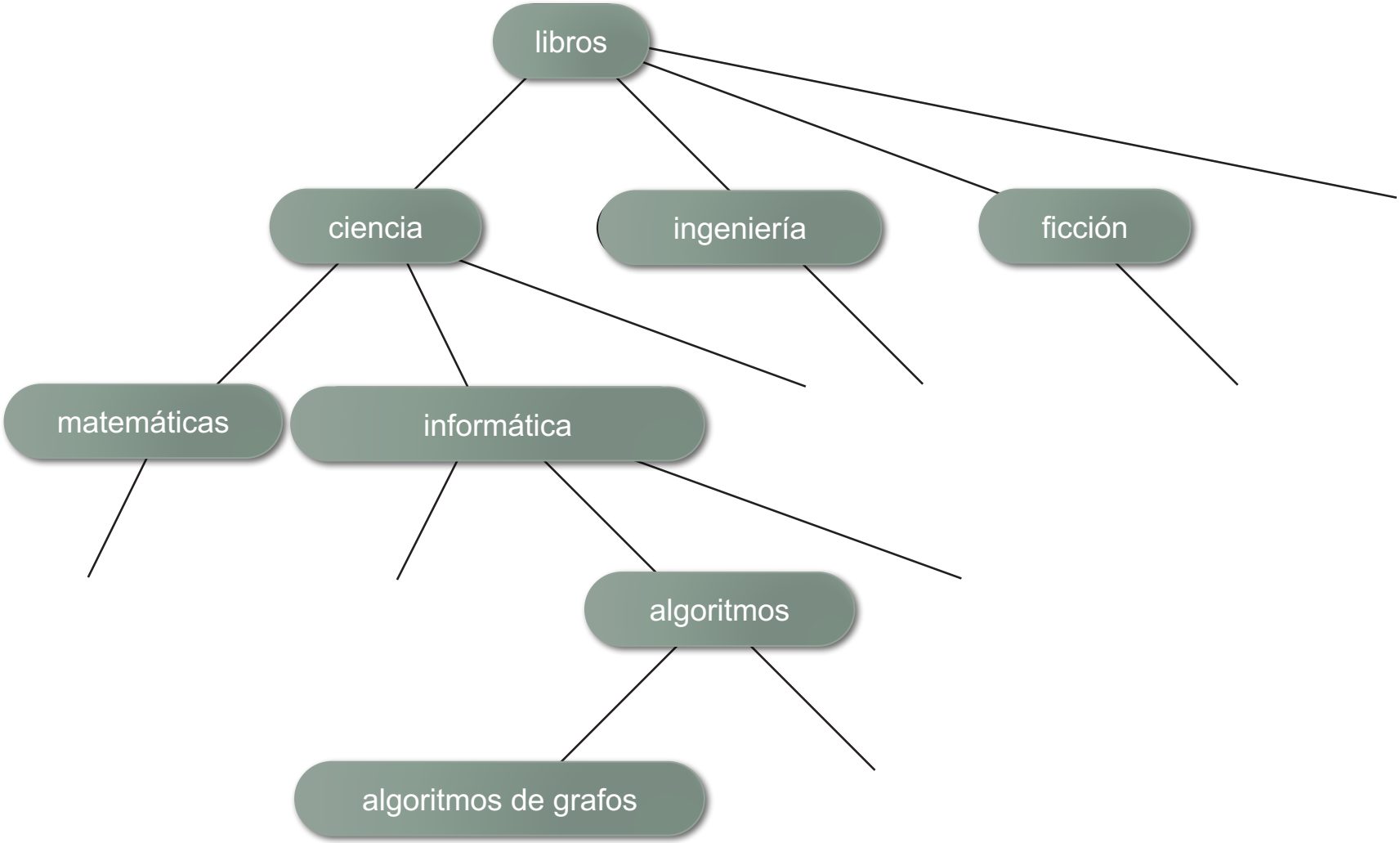
Recuperación de información y datos estructurados

- Los sistemas de recuperación de información originalmente trataban los documentos como colecciones de palabras.
- Los sistemas de **extracción de información** infieren una estructura a partir de los documentos, p.e.:
  - Extracción de atributos de la casa (tamaño, dirección, número de habitaciones, etc.) a partir de un anunció textual.
  - Extracción del tema y las personas citadas a partir de una noticia de prensa.
- Los datos extraídos se suelen guardan en relaciones o estructuras XML
  - El sistema busca conexiones entre los datos para responder consultas
  - **Sistemas de respuesta de consultas** (*question answering systems*)

Directorios

- Almacenan juntos documentos relacionados en una biblioteca, facilitando el acceso
  - Los usuarios pueden ver no sólo los documentos solicitados sino también los relacionados con ellos.
- El acceso se facilita mediante sistemas de clasificación que organizan juntos los documentos relacionados lógicamente.
- La organización es jerárquica: **jerarquía de clasificación**.

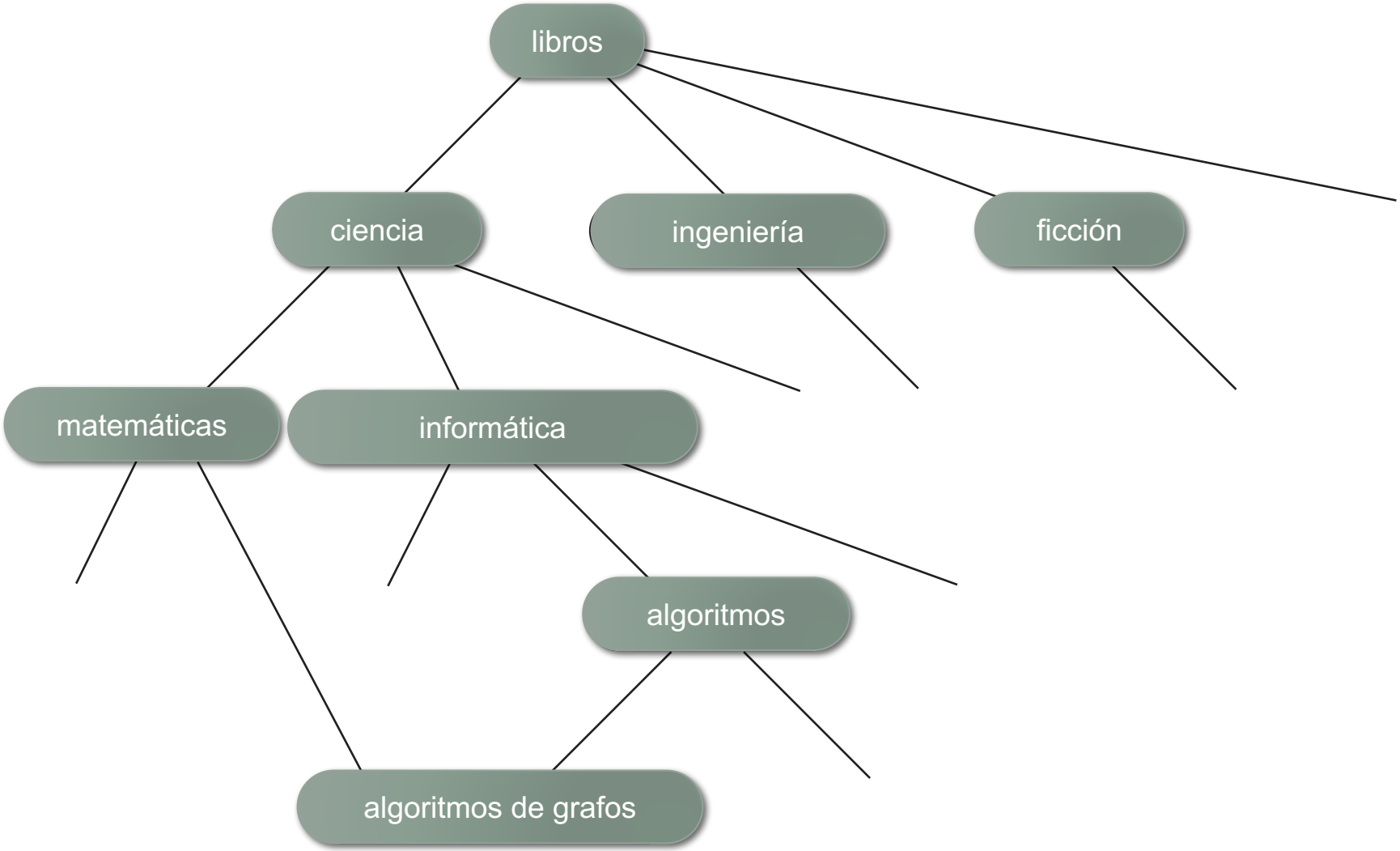
Una jerarquía de clasificación para un sistema de una biblioteca



Clasificación DAG

- Los documentos pueden estar en varios lugares de la jerarquía de un sistema de recuperación de información, ya que la posición física no es importante.
- La jerarquía de clasificación es, por tanto, un Grafo Directo Acíclico (DAG - *Directed Acyclic Graph*)

Una clasificación DAG para un sistema de recuperación de información de una biblioteca



Directorios Web

- Un **directorio Web** es un directorio de clasificación de páginas Web
  - P.e., Yahoo!, el proyecto Open Directory
  - Cuestiones:
    - ¿Cuál debería ser la jerarquía de directorio?
    - Dado un documento, ¿qué nodos del directorio son categorías relevantes para ese documento?
  - A menudo se realiza de forma manual
    - La clasificación de documentos en una jerarquía se puede hacer en base a similitud de términos.