

## 4.2 Bases de datos orientadas a documentos

---

4.2.1 Características

4.2.2 Arquitecturas

4.2.3 Análisis de  
ventajas y  
desventajas

4.2.4 Aplicaciones

# Bases de datos de documentos

---

- Una base de datos de documentos es un tipo de base de datos no relacional que ha sido diseñada para almacenar y consultar datos como documentos de tipo XML o JSON.
- Las bases de datos de documentos facilitan a los desarrolladores el almacenamiento y la consulta de datos en una base de datos mediante el mismo formato de modelo de documentos que emplean en el código de aplicación.
- La naturaleza flexible, semiestructurada y jerárquica de los documentos y las bases de datos de documentos permite que evolucionen según las necesidades de las aplicaciones.
- El modelo de documentos funciona bien con casos de uso como catálogos, perfiles de usuario y sistemas de administración de contenido en los que cada documento es único y evoluciona con el tiempo.
- Las bases de datos de documentos permiten una indexación fácil, potentes consultas ad hoc y análisis de colecciones de documentos.

# Documento

---

- El concepto central de una base de datos orientada a documentos es el concepto mismo de *Documento*. Mientras cada implementación de base de datos orientada a documentos difiere en los detalles, en general todas ellas comparten el principio de que los documentos encapsulan y codifican datos o información siguiendo algún formato estándar. Entre las codificaciones usadas en la actualidad se encuentran XML, YAML y JSON, así como formatos binarios como BSON.
- Los documentos dentro de una base de datos orientada a documentos son similares, de algún modo, a registros, tuplas o filas en una base de datos relacional pero menos rígidos. No se les requiere ajustarse a un esquema estándar ni tener todos las mismas secciones, atributos o claves.
- Las bases de datos de documentos generalmente proporcionan metadatos adicionales para asociarlos y almacenarlos junto con el contenido del documento. Esos metadatos pueden estar relacionados con las instalaciones que proporciona el almacén de datos para organizar documentos, brindar seguridad u otras características específicas de implementación.

# Ejemplos de documentos

## Ejemplo en XML:

```
<artista>
  <nombre_artistico>Iron Maiden</nombre_artistico>
  <albums>
    <album>
      <nombre>The Book of Souls</nombre>
      <lanzamiento>2015</lanzamiento>
      <genero>Hard Rock</genero>
    </album>
    <album>
      <nombre>Killers</nombre>
      <lanzamiento>1981</lanzamiento>
      <genero>Hard Rock</genero>
    </album>
    <album>
      <nombre>Powerslave</nombre>
      <lanzamiento>1984</lanzamiento>
      <genero>Hard Rock</genero>
    </album>
    <album>
      <nombre>Somewhere in Time</nombre>
      <lanzamiento>1986</lanzamiento>
      <genero>Hard Rock</genero>
    </album>
  </albums>
</artista>
```

## Ejemplo en JSON:

```
{
  "nombre_artistico": "IronMaiden",
  "albums": [
    {
      "nombre": "TheBookofSouls",
      "lanzamiento": 2015,
      "genero": "HardRock"
    },
    {
      "nombre": "Killers",
      "lanzamiento": 1981,
      "genero": "HardRock"
    },
    {
      "nombre": "Powerslave",
      "lanzamiento": 1984,
      "genero": "HardRock"
    },
    {
      "nombre": "SomewhereinTime",
      "lanzamiento": 1986,
      "genero": "HardRock"
    }
  ]
}
```

- 
- **Cada documento puede tener la misma o diferente estructura.**
    - Para agregar tipos adicionales de datos a una base de datos de documentos, no hay necesidad de modificar el esquema completo de la base de datos como con una base de datos relacional.
    - Los documentos se agrupan en "colecciones", que tienen un propósito similar a una tabla relacional. Una base de datos de documentos proporciona un mecanismo de consulta para buscar colecciones de documentos con atributos particulares.

# Organización

---

- Las implementaciones de bases de datos de documentos ofrecen una variedad de formas de organizar documentos, incluidas las nociones de
  - Colecciones: grupos de documentos, en los que, según la implementación, se puede obligar a un documento a vivir dentro de una colección, o se le puede permitir vivir en varias colecciones.
  - Etiquetas y metadatos no visibles: datos adicionales fuera del contenido del documento
  - Jerarquías de directorios: grupos de documentos organizados en una estructura similar a un árbol, generalmente basada en una ruta o URI
- A veces, estas nociones organizativas varían en cuanto a representaciones lógicas o físicas (por ejemplo, en disco o en memoria).

# Búsqueda y recuperación

---

- Los documentos se buscan mediante una *clave* única que identifica al documento. Generalmente esta clave se compone de una simple cadena. En algunos casos puede tratarse de una URI o una ruta, que sirve para rescatar el documento de la base de datos. Generalmente la base de datos mantiene un índice de dichas claves, por lo que la recuperación es rápida.
- Otra de las características que definen una base de datos orientada a documentos es que, más allá de la sencilla correspondencia clave-documento (o clave-valor) usada para recuperar un documento, la base de datos ofrece un API o un lenguaje de consulta para recuperar documentos según su contenido. El conjunto de características del API o del lenguaje de consulta, así como lo que se obtiene, varía significativamente entre distintas implementaciones.

# Ventajas

---

- **Modelado flexible de datos:** a medida que las aplicaciones web, móviles, sociales e IoT cambian la naturaleza de los modelos de datos de aplicaciones, las bases de datos de documentos eliminan la necesidad de forzar modelos de datos relacionales para admitir nuevos tipos de modelos de datos de aplicaciones.
- **Rendimiento de escritura rápido:** a diferencia de las bases de datos relacionales tradicionales, algunas bases de datos de documentos priorizan la disponibilidad de escritura sobre la estricta consistencia de los datos. Esto garantiza que las escrituras siempre serán rápidas, incluso si una falla en una parte del hardware o de la red da como resultado un pequeño retraso en la replicación de datos y la coherencia en todo el entorno.
- **Rendimiento rápido de consultas:** muchas bases de datos de documentos tienen potentes motores de búsqueda y funciones de indexación que proporcionan capacidades de consulta rápidas y eficientes.

# Casos de uso

---

## Administración de contenido

- Una base de datos de documentos es una excelente opción para aplicaciones de administración de contenido, como blogs y plataformas de video.
- La base de datos de documentos es más intuitiva para que un desarrollador actualice una aplicación a medida que evolucionan los requisitos.
- Además, si el modelo de datos necesita cambiar, solo se deben actualizar los documentos afectados.
- No se requiere actualización del esquema y no es necesario tiempo de inactividad de la base de datos para realizar los cambios.

## Catálogos

- Las bases de datos de documentos son eficientes y efectivas para almacenar información de catálogo.
  - Por ejemplo, en una aplicación de e-commerce, los diferentes productos generalmente tienen diferentes números de atributos. La administración de miles de atributos en bases de datos relacionales no es eficiente y afecta al rendimiento de lectura.
  - Al utilizar una base de datos de documentos, los atributos de cada producto se pueden describir en un solo documento para que la administración sea fácil y la velocidad de lectura sea más rápida. Cambiar los atributos de un producto no afectará a otros.

# eXtensible Markup Language XML

XML es un metalenguaje universal para definir etiquetas

Proporciona un marco de trabajo uniforme para intercambio de datos y metadatos entre aplicaciones

No obstante, el XML no proporciona ningún medio para hablar de la semántica de los datos, esto es, no hay significados asociados con el anidamiento de las etiquetas

- Cada aplicación debe de interpretar el anidamiento apropiadamente

# ¿Porque usar XML?

- Es importante porque:
  - Se puede trasladar cualquier dato a XML
  - Se puede enviar XML sobre la Web (HTTP, SOAP)
  - Se puede insertar XML en cualquier aplicación, lo que significa intercambio de datos en la Web

# Estructura de un documento XML

XML busca dar solución al problema de expresar información estructurada de la manera más abstracta y reutilizable posible, esto es, con partes bien definidas, y que esas partes se compongan a su vez de otras partes

Una etiqueta consiste en una marca hecha en el documento, que señala una porción de este como un elemento

Un documento XML está formado por el prólogo y por el cuerpo del documento así como texto de etiquetas que refiere el documento

# Prólogo

- El prólogo de un documento XML contiene:
  - Una declaración XML. Es la sentencia que declara al documento como un documento XML.
  - Una declaración de tipo de documento. Enlaza el documento con su DTD (definición de tipo de documento), o el DTD puede estar incluido en la propia declaración o ambas cosas al mismo tiempo.
  - Uno o más comentarios e instrucciones de procesamiento.

```
<?xml version="1.0" encoding="UTF-8 | ISO-  
8859-1 | windows-1252" standalone="no |  
yes"?>
```

- A diferencia del prólogo, el cuerpo no es opcional en un documento XML
- El cuerpo debe contener solo un elemento raíz, característica indispensable también para que el documento esté bien formado.

```
<Elemento_raiz>
  <SubElemento>
    ( . . . )
  </SubElemento>
</Elemento_raiz>
```

Cuerpo

# Elementos y etiquetas en XML

## Etiquetas

- Las etiquetas XML son sensivas a mayúsculas y minúsculas

Las etiquetas están normalmente en pares de inicio/fin

- <libro> ... </libro>

Todo el contenido debe estar entre etiquetas

- <libro>
  - <titulo>Fundamentos de Bases de Datos </titulo>
  - </libro>

Se pueden anidar arbitrariamente los elementos

Un elemento vacío se abrevia: <libro/>

# Atributos XML

- Son pares de nombre-valor que ocurren dentro de las etiquetas y después del nombre del elemento

```
<libro precio="250" moneda="Pesos">
<titulo>Fundamentos de Bases de datos
</titulo>
<autor>Abiteboul</autor>
...
</libro>
```

- Todos los valores de los atributos deben estar encerrados entre comillas dobles
- Un elemento puede tener varios atributos, pero su nombre solo puede ocurrir una vez

- Sintaxis:

```
<! [CDATA[ ... texto ... ]]>
```

- Se emplea para indicar al analizador (parser) que no se tome en cuenta el texto de la sección CDATA en el análisis

<ejemplo>

```
    <! [CDATA[ algo de texto </nada>
  ]]>
```

</ejemplo>

Secciones  
CDATA

# Referencias de entidad

- Sintaxis:

&nombre\_entidad;

- Se emplea para sustituir el texto indicado por la referencia de entidad

```
<ejemplo> 250 &lt; 200  
</ejemplo>
```

- Están definidas las siguientes referencias de entidad:

&lt;	<
&gt;	>
&amp;	&
&apos;	'
&quot;	"
&#38;	Caracter Unicode

# Comentarios

- Sintaxis:

<!-- ... texto de comentario... -->

- Las etiquetas de comentarios no son analizadas.

# Instrucciones de procesamiento

- Proporciona información a una aplicación externa o al procesador de XML
- <?receptor instrucción?>**
- *Receptor* identifica a la instrucción de procesamiento de la aplicación
  - *Instrucción* es un parámetro opcional pasado a la aplicación
- <?xmlstylesheet type="text/xml" href="contactos.xsl"?>**

- Es un mecanismo para nombrar etiquetas globalmente de forma única:

```
<h:html
    xmlns:xdc="http://www.xml.com/books"
        xmlns:h="http://www.w3.org/HTML/1998/html4"
    >
    <h:head><h:title>Book Review</h:title></h:head>
    ...
    <xdc:bookreview>
        <xdc:title>XML: A Primer</xdc:title>
        ...
    </h:html>
```

- Permite la mezcla de etiquetas de diferentes vocabularios sin confusión
- Los namespaces sólo identifican el vocabulario; es necesario tener mecanismos adicionales para la estructura y el significado de las etiquetas

## Namespaces en XML

# Documento XML bien formado

---

El documento inicia con la declaración de XML

---

```
<?xml version="1.0" standalone="yes"?>
```

---

standalone -> que no tienen una DTD asociada

---

El documento tiene un elemento principal que contiene a los demás elementos del documento

---

Todas las etiquetas deben de estar anidadas correctamente

# Documento válido

Es un documento que se ajusta a la DTD (Document Type Definition, Definición de Tipo Documento) declarada en él

Solo los documentos XML con una DTD se dice que son válidos

Un documento XML sin una DTD puede ser válido o no, pero debe ser bien formado

# Reglas de la DTD

- Una DTD dice que es permitido dentro de la estructura del documento
  - Qué elementos pueden ocurrir
  - Qué atributos pueden o deben estar en un elemento
  - Qué subelementos pueden o deben ocurrir dentro de cada elemento y cuantas veces
- Aplica a los elementos en el contexto
- Sin embargo, no establece restricciones a los datos
- Sintaxis:

```
<!ELEMENT elemento (subelementos)>
<!ATTLIST elemento (atributos)>
```
- Los subelementos pueden ser especificados como
  - Nombres de elementos, o
  - #PCDATA (parsed character data, esto es, cadenas de texto), o
  - EMPTY (sin subelementos)

# Cardinalidad

- Se puede especificar la cardinalidad de ocurrencia de elementos mediante los siguientes indicadores:
  - ? cero o una ocurrencia
  - + una o más ocurrencias
  - \* cero o más ocurrencias
- Para indicar un orden en la posición de los subelementos, se debe especificar mediante coma ',' para una secuencia (respeta posición y es obligatorio), | para choice (especifica que solo una de las opciones puede aparecer en un momento determinado) , o paréntesis () para especificar agrupamientos de elementos.
- Los atributos pueden ser alguno de los siguientes:
  - REQUIRED obligatorio
  - IMPLIED opcional
  - FIXED valor constante

# DTD asociada a un documento XML

```
<?xml version="1.0" ?>
<!DOCTYPE bibliografia [
<!ENTITY www "World Wide Web" >
<!ELEMENT autor (#PCDATA)>
<!ATTLIST autor autorRef CDATA #REQUIRED edad CDATA #REQUIRED>
<!ELEMENT autores (autor+)>
<!ELEMENT contenido EMPTY>
<!ATTLIST contenido enlace CDATA #REQUIRED>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT relacionado EMPTY>
<!ATTLIST relacionado articulos CDATA #REQUIRED>
<!ELEMENT articulo (autores, contenido?, titulo, relacionado*)>
<!ATTLIST articulo pubid CDATA #REQUIRED rol CDATA #REQUIRED>
<!ELEMENT bibliografia (articulo+)>
]>
<bibliografia>
<articulo pubid="wsa" rol="publication">
  <autores>
    <autor autorRef="joyce" edad="45">J. R. Collins </autor>
  </autores>
  <contenido enlace="http://mysite.com/confusion"/>
  <titulo>Object Confusion in a Deviator System in &www; </titulo>
  <relacionado articulos="Systems, Analysis"/>
</articulo>
</bibliografia>
```

# Tipos ID, IDREF y IDREFS

- Es posible especificar que un atributo sea un atributo de tipo ID, para que los atributos especificados como IDREF o IDREFS dentro del documento pueden usarse para hacer referencia a los atributos de tipo ID, lo que habilita los vínculos entre documentos.
  - Los atributos de tipo ID, IDREF e IDREFS corresponden a relaciones PK/FK (clave principal/clave externa) de una base de datos relacional, con algunas diferencias.
- Para que los atributos ID, IDREFS e IDREF sean válidos:
  - El valor de ID debe ser único dentro del documento XML.
  - Para cada atributo IDREF e IDREFS, los valores de ID a los que se haga referencia deben estar en el documento XML.
  - El valor de un atributo ID, IDREF e IDREFS debe ser un token con nombre. (Por ejemplo, el valor entero 101 no puede ser un valor ID)

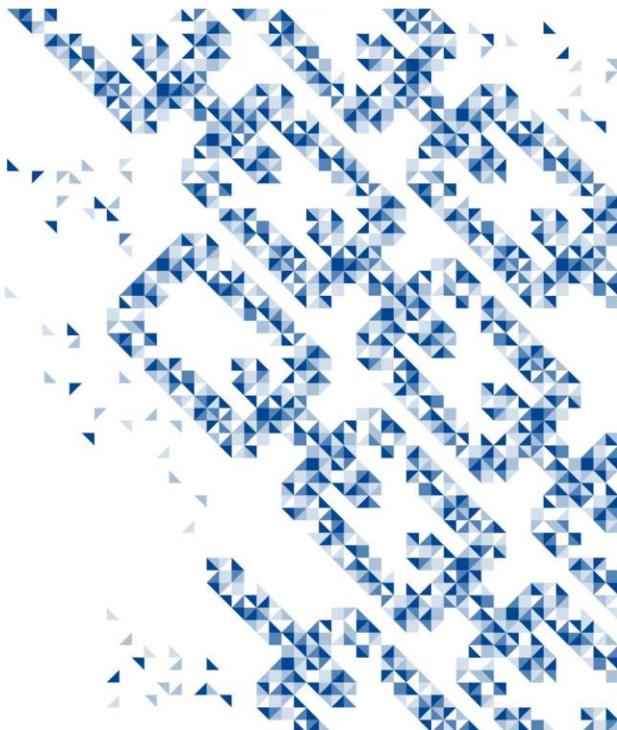
# Validación XML mediante una DTD

- La validación de una DTD asociada a un documento XML debe realizarse por un paser validador.
- En caso de que el contenido de la DTD esté en un archivo externo (no contenido en el documento XML), se debe incluir el tipo SYSTEM y el nombre del archivo DTD, que deberá estar en la misma ruta del archivo XML para que sea localizado.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE raiz SYSTEM "archivo.dtd">
<raiz> ...
```

- También es posible indicar otro directorio de residencia del archivo utilizando una ruta completa o una URL, cuidando de que el archivo esté disponible y sea localizable.
- <!DOCTYPE raiz SYSTEM  
"http://misitio.com./dtds/archivo.dtd">

# XML Schema



- El principal motivo de la definición de XML Schema es satisfacer las carencias de las DTD con respecto a la definición de la estructura de un documento XML.
- Los requerimientos establecidos por la W3C fueron principalmente:
  - Proporcionar un conjunto más rico de datos que las DTD.
  - Proporcionar un sistema de tipo de datos que permita la definición de datos construidos por el usuario, con sus restricciones.
  - Distinguir la representación léxica de la información contenida.

# Ventajas de XML Schema

---

Descripción y restricciones de documentos usando la sintaxis de XML.

---

Soporte para tipos de datos.

---

Descripción del modelo de contenido y el reuso de elementos vía la herencia.

---

Extensibilidad.

---

Habilidad de escribir esquemas dinámicos.

---

Capacidades de autodocumentación cuando una hoja de estilo es aplicada.

# XML Schema

- La especificación de XML Schema consiste de tres partes:
  - *XML Schema Parte 0*: Primer.- Explica qué son los esquemas, como difieren de las DTD y cómo construir un esquema.
  - *XML Schema Parte 1: Estructuras*.- Propone métodos para describir y la estructura y contenido de los documentos XML, y define las reglas para gobernar la validación de documentos.
  - *XML Schema Parte 2: Tipos de datos*.- Define un conjunto simple de tipos de datos que son asociados con tipos de elementos y atributos XML.

# Definiciones de esquemas

- Un esquema es un documento XML.
- Debido a que es un documento autodescriptivo, se definen las etiquetas:  
`<element name="nombre_elemento">`  
`<attribute name="nombre_atributo">`
- para establecer las características del documento XML.
- También se hace distinción de cuando un elemento contiene otros elementos o no, mediante:
  - Tipos complejos.- son elementos que contienen otros elementos, o que tienen atributos.
  - Tipos simples.- son elementos que no tienen hijos ni tampoco atributos. Los atributos también son tipos simples.

# Ejemplo

**Orderdata.xml**

```
<Ordenes>
    <Cliente nombre="Juan" apellido="López" correoe="jlopez@mail.com"/>
    <Pedido/>
</Ordenes>
```

**Orderdata.xsd**

```
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:element name="Ordenes">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Cliente" >
                    <xs:complexType>
                        <xs:attribute name="nombre" type="xs:string" use="required" />
                        <xs:attribute name="apellido" type="xs:string" use="required" />
                        <xs:attribute name="correoe" type="xs:string" />
                    </xs:complexType>
                </xs:element>
                <xs:element name="Pedido" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

- El modelo de contenido es la forma en que se definen las estructuras válidas de los elementos y otros componentes. De aquí, hay dos tipos de etiquetado:
  - Definiciones.- para crear nuevos tipos (simples o complejos).
  - Declaraciones.- describe el modelo de contenido de elementos y atributos. A su vez, posee dos tipos de declaraciones:
    - Declaraciones simples.- crean tipos simples.
    - Declaraciones complejas.- crean tipos complejos.

## Tipos de datos y estructuras

- Un documento XML Schema consiste en un preámbulo, seguido de dos o más declaraciones.
- Está definido por la etiqueta `<schema>`, la cual tiene los siguientes atributos:

```
<xs:schema version="1.0"  
xmlns:xs="http://www.w3.org/2001/XMLSchema"  
elementFormDefault="qualified">
```

- en donde el namespace `xs:` se usa para identificar los tipos de datos y los elementos del esquema.

## Preámbulo

# Declaración de elementos

- La forma de declarar un elemento es usando la etiqueta <element>

```
<xs:element name="nombre_elem" />
```

en donde el atributo ***name*** es obligatorio.

- También es posible establecer restricciones al elemento, como:
  - El tipo primitivo o construido, mediante el atributo ***type***.
  - Valor por omisión, con el atributo ***default***.
  - Valores constantes con el atributo ***fixed***.

# Declaraciones de atributos

- La forma de declarar un atributo es usando la etiqueta <attribute>

```
<xs:attribute name="nombre_atrib" />
```

en donde el atributo ***name*** es obligatorio.

- También es posible establecer restricciones al atributo, como:
  - Valor por omisión, con el atributo ***default***.
  - El tipo primitivo o construido, mediante el atributo ***type***.
  - (opcional) La restricción de existencia, determinada por el atributo ***use*** (“required”).
  - Un valor constante mediante el atributo ***fixed***.

RESTRICCIÓN	DESCRIPCIÓN
<b>enumeration</b>	Define una lista de valores aceptables
<b>fractionDigits</b>	Especifica el máximo número de decimales permitido ( $\geq 0$ )
<b>length</b>	Especifica el número exacto de caracteres o lista de ítems permitidos ( $\geq 0$ )
<b>maxExclusive</b>	Especifica el límite superior para valores numéricos
<b>maxInclusive</b>	Especifica el límite superior para valores numéricos (igual inclusive)
<b>maxLength</b>	Especifica el máximo número de caracteres o lista de ítems permitidos ( $\geq 0$ )
<b>minExclusive</b>	Especifica el límite inferior para valores numéricos
<b>minInclusive</b>	Especifica el límite inferior para valores numéricos (igual inclusive)
<b>minLength</b>	Especifica el mínimo número de caracteres o lista de ítems permitidos ( $\geq 0$ )
<b>pattern</b>	Define la secuencia exacta de caracteres que son aceptables
<b>totalDigits</b>	Especifica el número exacto de dígitos permitidos ( $\geq 0$ )
<b>whiteSpace</b>	Especifica cómo son manejados los espacios en blanco (line feeds, tabs, spaces y carriage returns) value="preserve   replace   collapse"

## Restricciones

- Es posible limitar los valores que se establecen en un elemento o atributo mediante una restricción.
- Las restricciones aplicables son las siguientes:

- El elemento <password> puede contener letras mayúsculas, minúsculas y dígitos del 0-9, hasta máximo 8 caracteres.

```
<xss:element name="password">
  <xss:simpleType>
    <xss:restriction base="xss:string">
      <xss:pattern value="[a-zA-Z0-9]{8}" />
    </xss:restriction>
  </xss:simpleType>
</xss:element>
```

- El elemento <password> contiene únicamente entre 5 y 8 caracteres.

```
<xss:element name="password">
  <xss:simpleType>
    <xss:restriction base="xss:string">
      <xss:minLength value="5" />
      <xss:maxLength value="8" />
    </xss:restriction>
  </xss:simpleType>
</xss:element>
```

# Ejemplos

# Ejemplos

- El elemento `<direccion>` no tendrá espacios en blanco sobrantes
  - ```
<xs:element name="direccion">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:whiteSpace value="collapse"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```
- El elemento `<coche>` solo podrá contener la lista de valores posibles.
  - ```
<xs:element name="coche" type="cocheT"/>
<xs:simpleType name="cocheT">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Audi"/>
        <xs:enumeration value="Golf"/>
        <xs:enumeration value="BMW"/>
    </xs:restriction>
</xs:simpleType>
```

# Definiciones de tipos complejos

- Para definir elementos que contienen más elementos y/o atributos, se emplea la etiqueta <**complexType**>. Usualmente contiene un conjunto de declaraciones de elementos y atributos o de referencias a elementos.
- Posteriormente se puede hacer referencia a él mediante el atributo **type**.
- Si se especifica el atributo **mixed**=**"true"**, entonces el elemento puede contener texto y otros subelementos.
- Se pueden crear "tipos" con nombre, lo que permite que puedan ser reutilizados posteriormente en otras definiciones.

```
<xs:complexType name="ClienteT">
    <xs:attribute name="nombre" type="xs:string" />
    <xs:attribute name="apellido" type="xs:string" />
    <xs:attribute name="correo" type="xs:string" />
</xs:complexType>
<xs:element name="Customer" type="ClienteT">
```

- Para especificar el orden en que deben de estar los elementos o atributos, se emplean las etiquetas:
  - **<sequence>**, para indicar que los elementos deben de estar en el orden en que fueron declarados,
  - **<choice>**, para indicar que solo uno de los elementos declarados debe estar presente,
  - **<all>**, para que los elementos puedan aparecer en cualquier orden.

## Secuencias

# Ejemplo

- El siguiente ejemplo es de un elemento *Cliente*, el cual permite que los elementos definidos en él aparezcan en cualquier orden, pero sólo una vez .

```
<xs:element name="Cliente" >
  <xs:complexType>
    <xs:all>
      <xs:element name="nombre" minOccurs="1" />
      <xs:element name="apellido" minOccurs="1" />
      <xs:element name="correo" minOccurs="0"
maxOccurs="1" />
    </xs:all>
  </xs:complexType>
</xs:element>
```

- Una instancia de un documento XML con el esquema anterior sería:

```
<Cliente>
  <nombre>Juan</nombre>
  <apellido>López</apellido>
</Cliente>
```

- Así como también:

```
<Cliente>
  <correo>jlopez@mail.com</correo>
  <apellido>López</apellido>
  <nombre>Juan</nombre>
</Cliente>
```

# Tipos de datos primitivos

## Tipos de datos Primitivos

string	→ "Hello World"
boolean	→ {true, false, 1, 0}
decimal	→ 7.08
float	→ 12.56E3, 12, 12560, 0, -0, INF, -INF, NAN
double	→ 12.56E3, 12, 12560, 0, -0, INF, -INF, NAN
duration	→ P1Y2M3DT10H30M12.3S
dateTime	→ formato: CCYY-MM-DDThh:mm:ss
time	→ formato: hh:mm:ss.sss
date	→ formato: CCYY-MM-DD
gYearMonth	→ formato: CCYY-MM
gYear	→ formato: CCYY
gMonthDay	→ formato: --MM-DD

Nota: 'T' es el separador de fecha/tiempo

INF = infinito

NAN = not-a-number

# Tipos de datos primitivos

## Tipos de datos Primitivos

gDay	→ formato: ---DD (note los 3 guiones)
gMonth	→ formato: --MM--
hexBinary	→ una cadena hexadecimal
base64Binary	→ una cadena base 64
anyURI	→ <a href="http://www.xfront.com">http://www.xfront.com</a>
QName	→ un nombre de namespace calificado
NOTATION	→ Una NOTATION de la espec. XML

# Tipos de datos derivados

<b>Tipos derivados</b>	<b>Subtipo de tipo primitivo</b>
normalizedString	Cadena sin tabs, LF, o CR
token	Cadena sin tabs, LF, espacios antes o después, o consecutivos
language	Cualquier valor valido xml:lang value, ej., EN, FR, ..
IDREFS	Debe ser usado sólo con atributos
ENTITIES	Debe ser usado sólo con atributos
NMTOKEN	Debe ser usado sólo con atributos
NMTOKENS	Debe ser usado sólo con atributos
Name	
NCName	parte (sin calificador de namespace)
ID	Debe ser usado sólo con atributos
IDREF	Debe ser usado sólo con atributos
ENTITY	Debe ser usado sólo con atributos
integer	456
nonPositiveInteger	infinito negativo a 0

# Tipos de datos derivados

Tipos derivados	Subtipo de tipo primitivo
negativeInteger	infinito negativo a -1
long	-9223372036854775808 a 9223372036854775807
int	-2147483648 a 2147483647
short	-32768 a 32767
byte	-127 a 128
nonNegativeInteger	0 a infinito
unsignedLong	0 a 18446744073709551615
unsignedInt	0 a 4294967295
unsignedShort	0 a 65535
unsignedByte	0 a 255
positiveInteger	1 a infinito

# Validación con XML Schema

- Debe ser definido en el elemento raíz a través de un Namespace/atributo :

```
<raiz_XML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="archivoXMLSchema.xsd | URL">
```

...

- El namespace es utilizado para documentos que serán validados mediante XMLSchema; cabe mencionar que al ser procesado el documento no se visita la página de www.w3.org, sino que esta declaración es resuelta a una definición interna en el parser.
- El parámetro *xsi:noNamespaceSchemaLocation*, indica que el XMLSchema utilizado para validar el documento es el archivo llamado **archivoXMLSchema.xsd** el cual se encuentra en el mismo directorio del documento en cuestión, o en su caso, una URL en donde se encuentre ubicado el archivo XSD.
- De acuerdo a las reglas de namespaces, se permite que todo elemento anidado dentro de esta declaración pertenezca al "default namespace" por lo que no se requiere de ningún prefijo en los elementos del documento XML.

# DOM

El Document Object Model (DOM) es un modelo que especifica la forma de acceder los datos de un documento XML

Especifica un árbol jerárquico de nodos (elementos, comentarios, entidades, atributos, etc.) construido en memoria.

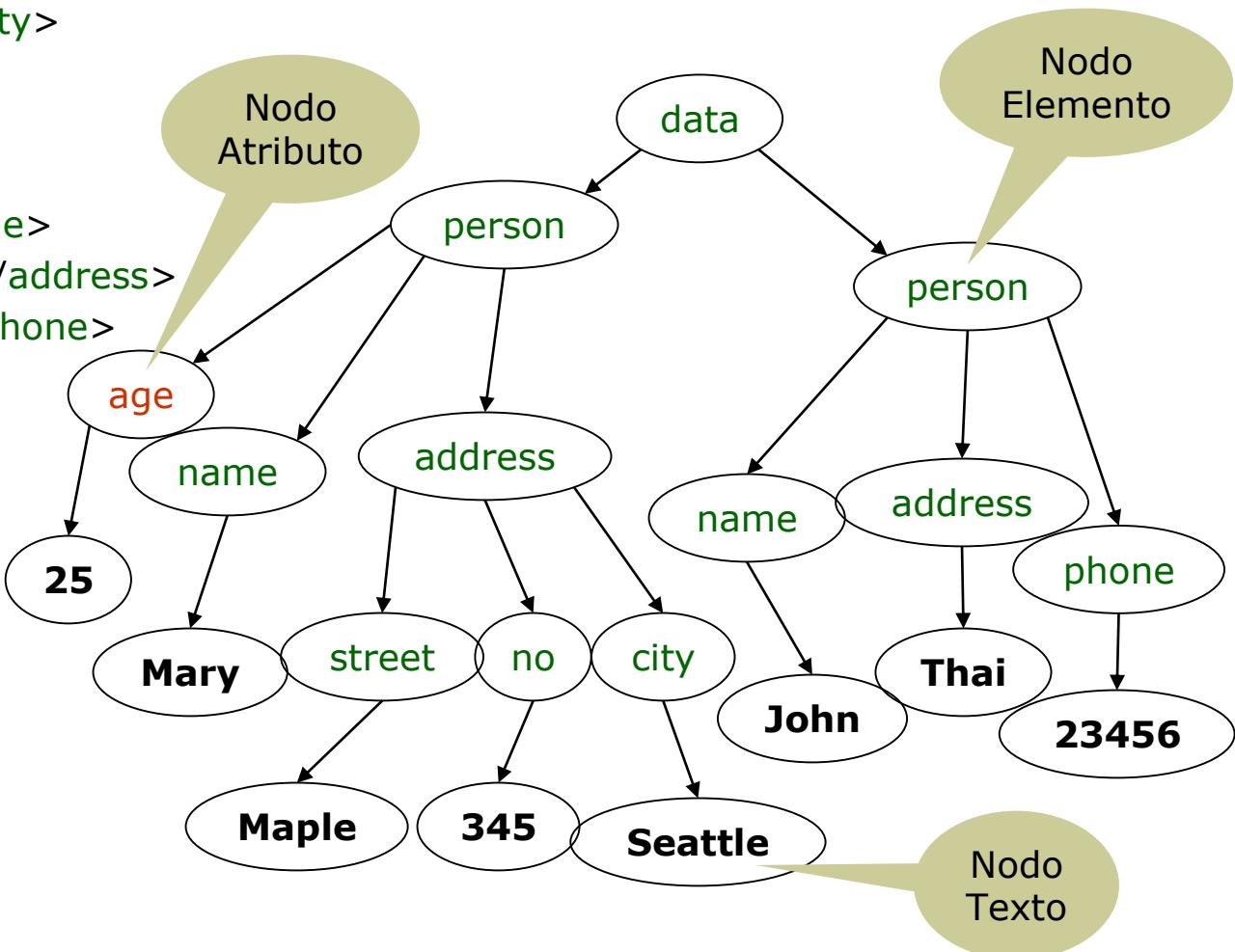
Además especifica un conjunto de funciones para desplazarse por el árbol

- <http://www.w3c.org/TR/REC-DOM-Level-1>

Incluye soporte para XML 1.0 y HTML, con cada elemento HTML representado como una interfaz.

# Semántica XML representada como árbol

```
<data>
  <person age="25" >
    <name> Mary </name>
    <address>
      <street> Maple </street>
      <no> 345 </no>
      <city> Seattle </city>
    </address>
  </person>
  <person>
    <name> John </name>
    <address> Thailand </address>
    <phone> 23456 </phone>
  </person>
</data>
```



# SGBDR y XML

Un documento XML podría almacenarse en un SGBDR directamente en un campo LOB, varchar o descompuesto en tablas, pero esto no es eficiente y además es complejo de mantener

Por tanto, se requiere incorporar el tipo de dato nativo XML en los gestores

Esta es la alternativa que se está llevando a cabo por la mayoría de gestores (Oracle, SQL Server, DB2,...)

# Niveles de anidamiento en XML

- XML no está restringido a un sólo nivel de anidamiento para representar un registro de una base de datos

```
<persona>
  <nombre>
    <propio>Juan</propio>
    <apellido>López</apellido>
  </nombre>
  <tel>1234</tel>
</persona>
```

# Datos semiestructurados

```
<persona>
    <nOMBRE>María</nOMBRE>
    <tel>2345</tel>
    <tel>3456</tel>
</persona>
```

nombre	tel	
María	2345	3456

- XML es autodescriptivo
- Los elementos del esquema son parte de los datos, por lo tanto, es más flexible

## Datos XML

```
<persona>
    <nombre>Juan</nombre>
    <tel>1234</tel>
</persona>

<persona>
    <nombre>María</nombre>
</person>
```

nombre	tel
Juan	1234
María	-

# Mapeo DB- XML

- Un documento puede estructurarse para adoptar un esquema de una BD

**SQL**

```
CREATE TABLE Cliente  
(nombre varchar(50),  
apellido varchar(50),  
direccion varchar(50),  
ciudad varchar(60),  
estado varchar(2),  
CP varchar(10)  
)
```

**XML**

```
<!ELEMENT Cliente  
(nombre,apellido,direccion,ciudad,estado,CP)>  
<!ELEMENT nombre (#PCDATA)>  
<!ELEMENT apellido (#PCDATA)>  
<!ELEMENT direccion (#PCDATA)>  
<!ELEMENT ciudad (#PCDATA)>  
<!ELEMENT estado (#PCDATA)>  
<!ELEMENT CP (#PCDATA) >
```

# Mapeo DB- XML

- Y las instancias de los datos:

nombre	apellido	direccion	ciudad	estado	CP
Juan	López	Av. Granjas #243	Izcalli	EDOMEX	78675

```
<Cliente>
<nombre>Juan</nombre>
<apellido>López</apellido>
<direccion>Av. Granjas #243</direccion>
<ciudad>Izcalli</ciudad>
<estado>EDOMEX</estado>
<CP>78675</CP>
</Cliente>
```

# Distintas representaciones

- Cuando se representan datos en un documento XML, cualquier elemento que sea definido para contener sólo texto usando el tipo #PCDATA tendrá su correspondencia a una columna en una BD relacional.
- Otra forma para representar el documento XML sería la siguiente:

```
<!ELEMENT Cliente EMPTY>
<!ATTLIST Cliente
  nombre CDATA #REQUIRED
  apellido CDATA #REQUIRED
  direccion CDATA #REQUIRED
  ciudad CDATA #REQUIRED
  estado CDATA #REQUIRED
  CP CDATA #REQUIRED>
```

```
<Cliente nombre="Juan"
  apellido="López"
  direccion="Av. Granjas #243"
  ciudad="Izcalli" estado="EDOMEX"
  CP="78675" />
```

# Comparación

Las dos estrategias de mapeo son:

- Elementos, los cuales son anidados como hijos del elemento que representa el grupo de información.
- Atributos, los cuales son añadidos a los elementos que representan los grupos de información.

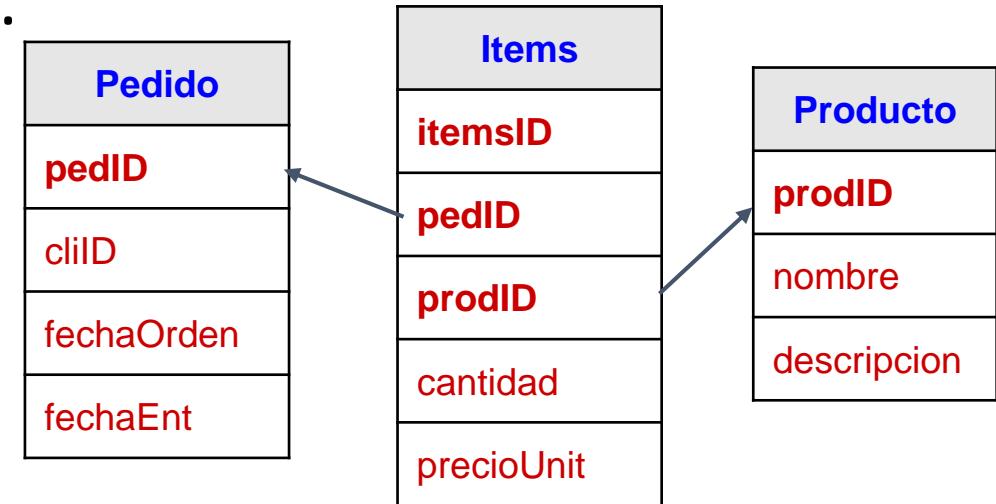
Las diferencias observables con respecto a las anteriores estrategias son:

- Legibilidad
- Compatibilidad con BD
- Fuerte tipado de datos
- Complejidad programática
- Tamaño del documento

# Relaciones

- Utilizando los atributos ID y IDREF es posible hacer el equivalente del modelo relacional.

```
CREATE TABLE Pedido (
    pedID NUMBER PRIMARY KEY,
    cliID NUMBER,
    fechaOrden DATE,
    fechaEntr DATE );
CREATE TABLE Producto (
    prodID NUMBER PRIMARY KEY,
    nombre VARCHAR(50),
    descripcion VARCHAR(255));
CREATE TABLE Items (
    itemsID NUMBER,
    pedID NUMBER,
    prodID NUMBER,
    cantidad NUMBER,
    precioUnit NUMBER(8,2),
    CONSTRAINT pk_item_ped_prod PRIMARY KEY (itemsID,pedID,prodID),
    CONSTRAINT fk_Items_Ped FOREIGN KEY (pedID) REFERENCES Pedido(pedID),
    CONSTRAINT fk_Items_Prod FOREIGN KEY (prodID) REFERENCES Producto(prodID));
```



# Ejemplo de documento XML (usando ID - IDREF)

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Ordenes (Pedido+, Producto+)>
<!ELEMENT Pedido (Items+)>
<!ATTLIST Pedido fechaOrden CDATA #REQUIRED fechaEntr CDATA #REQUIRED>
<!ELEMENT Items EMPTY>
<!ATTLIST Items productoREF IDREF #REQUIRED cantidad CDATA #REQUIRED precioUnit CDATA
#REQUIRED>
<!ELEMENT Producto EMPTY>
<!ATTLIST Producto prodID ID #REQUIRED nombre CDATA #REQUIRED descripcion CDATA #REQUIRED>

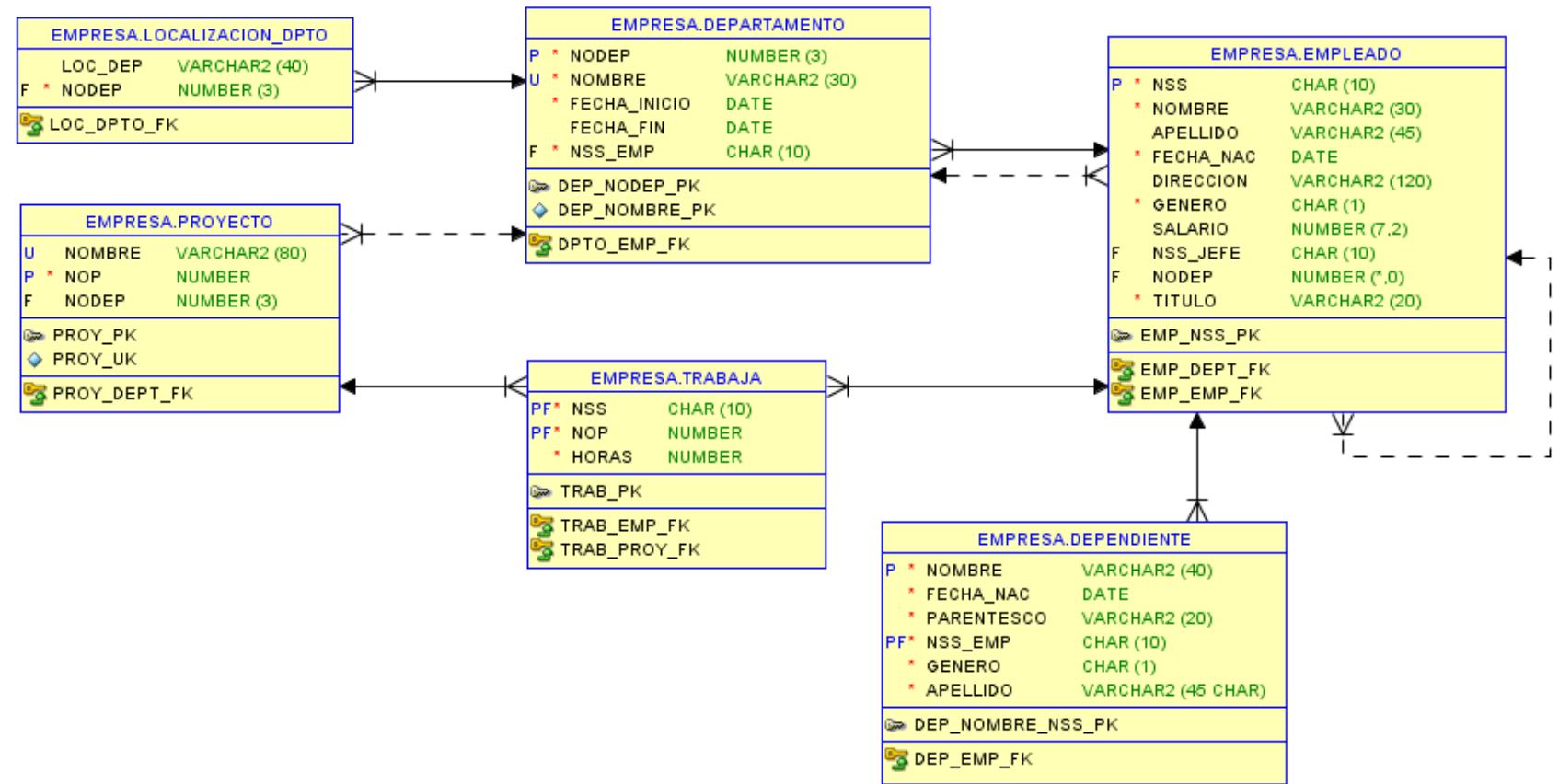
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Ordenes SYSTEM "Ordenes.dtd">
<Ordenes>
    <Pedido fechaOrden="7/23/2000" fechaEntr="7/28/2000">
        <Items productoREF="prod1" cantidad="17" precioUnit="0.10"/>
        <Items productoREF="prod2" cantidad="22" precioUnit="0.05"/>
    </Pedido>
    <Pedido fechaOrden="7/23/2000" fechaEntr="7/28/2000">
        <Items productoREF="prod2" cantidad="30" precioUnit="0.05"/>
        <Items productoREF="prod3" cantidad="19" precioUnit="0.15"/>
    </Pedido>
    <Producto prodID="prod1" nombre="tornillo" descripcion="acero inoxidable (3
pulg.)"/>
        <Producto prodID="prod2" nombre="pijas" descripcion="galvanizadas (media
pulg.)"/>
        <Producto prodID="prod3" nombre="tuercas" descripcion="acero inoxidable (½
pulg.)"/>
    </Ordenes>
```

# Evitando las referencias

```
<!ELEMENT Ordenes (Pedido+)>
<!ELEMENT Pedido (Items+)>
<!ATTLIST Pedido fechaOrden CDATA #REQUIRED fechaEntr CDATA #REQUIRED>
<!ELEMENT Items (Producto)>
<!ATTLIST Items cantidad CDATA #REQUIRED precioUnit CDATA #REQUIRED>
<!ELEMENT Producto EMPTY>
<!ATTLIST Producto nombre CDATA #REQUIRED descripcion CDATA #REQUIRED>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Ordenes SYSTEM "Ordenes.dtd">
<Ordenes>
    <Pedido fechaOrden="7/23/2000" fechaEntr="7/30/2000">
        <Items cantidad="17" precioUnit="0.10">
            <Producto nombre="tornillo" descripcion="acero inoxidable (3 pulg.)"/>
        </Items>
        <Items cantidad="22" precioUnit="0.05">
            <Producto nombre="pijas" descripcion="galvanizadas (media pulg.)"/>
        </Items>
    </Pedido>
    <Pedido fechaOrden="7/23/2000" fechaEntr="7/30/2000">
        <Items cantidad="30" precioUnit="0.05">
            <Producto nombre="pijas" descripcion="galvanizadas (media pulg.)"/>
        </Items>
        <Items cantidad="19" precioUnit="0.15">
            <Producto nombre="tuercas" descripcion="acero inoxidable (¼ pulg.)"/>
        </Items>
    </Pedido>
</Ordenes>
```

# Ejemplo de transformación



# ¿Porque bases de datos y XML?

## Con respecto a XML:

- Es una tecnología que habilita el intercambio eficiente de información entre aplicaciones Web.
- Es simple y extensible.
- Hay un fuerte apoyo de las empresas para su adopción.

## Con respecto a las bases de datos:

- Es una tecnología bastante madura, reflejada en los SADB actuales.
- El valor de las empresas está depositado en la información que almacenan y explotan.
- Sigue una constante evolución hacia otras nuevas tecnologías (bodegas de datos, multimedia, orientación a objetos, etc.)

# Sin embargo...

Debido al modelo semiestructurado de un documento XML, puede ser difícil hacer búsquedas dentro y entre documentos.

La administración y actualización de documentos XML puede llegar a ser una tarea compleja.

Los SABD no tienen un formato estándar para el intercambio de información.

Para habilitar una base de datos para la Web, son necesarias tecnologías adicionales.

Las empresas buscan realizar negocios a través de Internet (B2B y B2C)...

# Bases de Datos XML nativas

- Define un modelo (lógico) para un documento XML, y recupera y almacena documentos de acuerdo a ese modelo. Como mínimo, el modelo debe de incluir elementos, atributos y PCDATA (texto).
- Tiene como unidad fundamental de almacenamiento a un documento XML.
- No es necesario que tenga algún modelo particular de almacenamiento físico. Esto es, puede ser construida sobre una base de datos relacional, jerárquica u orientada a objetos, o usar un formato propietario de almacenamiento tal como archivos indexados comprimidos.

# Características

- Crean modelos lógicos en XML.
  - Mapean los modelos al mecanismo de almacenamiento correspondiente.
  - Las operaciones con los documentos se realizan en XML.
  - Dan un mayor nivel de abstracción al programador.
  - Dependencia del esquema.
    - Gestionan documentos como colecciones de datos.
    - No todas necesitan un esquema para almacenar documentos.
      - Problemas de integridad intra-documento.
    - Los esquemas se definen con DTD o con XML Schema (W3C).
  - No usan un mecanismo concreto de almacenamiento físico.
    - Depende de cada producto.
  - La unidad mínima de almacenamiento es un documento XML.
  - Existen retos pendientes para la integridad global de la BD
    - Integridad referencial inter-documento.
    - Restricciones semánticas inter-documento.

# Ventajas

Buenas para almacenar documentos XML

Pueden almacenar documentos XML así como formatos de estilos

Las aplicaciones son débilmente acopladas

El modelado de datos es simple y flexible

Complementan a los SABDR con soluciones de mapeo XML

El rendimiento puede ser muy bueno

# Algunos sistemas

## Almacenamiento nativo

- [eXist-db](#) (Open Source)
- [MarkLogic](#)
- [OpenLink Virtuoso](#)

## Construido sobre una base de datos de objetos

- [ozone](#) (Open Source)

## Construida sobre una base de datos relacional

- [pureXML overview -- Db2 as an XML database](#)
- [Getting into SQL/XML with Oracle Database 10g Release 2](#)
- [PostgreSQL](#)

# Inclusión de XML en SQL

Nueva parte del estándar para crear y manipular documentos XML.

ISO/IEC 9075-14: XML-Related Specifications (SQL/XML).

- Nuevo tipo predefinido.
- Nuevos operadores predefinidos para crear y manipular valores de tipo XML.
- Reglas para mapear tablas, esquemas y catálogos a documentos XML.

# SGBDR y XML

Un documento XML podría almacenarse en un SGBDR directamente en un campo LOB, VARCHAR o descompuesto en tablas, pero esto no es eficiente y además es complejo de mantener

Por tanto, se requiere incorporar el tipo de dato nativo XML en los gestores

Esta es la alternativa que usan la mayoría de gestores (Oracle, PostgreSQL, DB2,...)

# Tipo de dato XML

Permite almacenar datos XML de forma nativa en la BD

Puede ser optimizado (representación interna diferente a la cadena de caracteres)

Puede almacenar:

- Documentos XML bien formados (sólo un nodo raíz)
- Contenido XML (elementos, secuencia de elementos, texto,...)
- Se basa en XQuery. El valor de un tipo de dato XML es una instancia del modelo de datos XQuery.

- Soporte de XML en SGBD
  - IBM, Oracle implementan casi por completo el SQL/XML
  - Microsoft soporta similares características, pero con sintaxis propietaria
  - Todos soportan XQuery dentro del SQL
  - Existen diferencias en su implementación física (almacenamiento)
- Oracle 10g basado en CLOB o tablas OR
- Microsoft 2005 y 2008 almacenado como BLOB en formato interno propietario
- DB2 V9 basado en CLOB

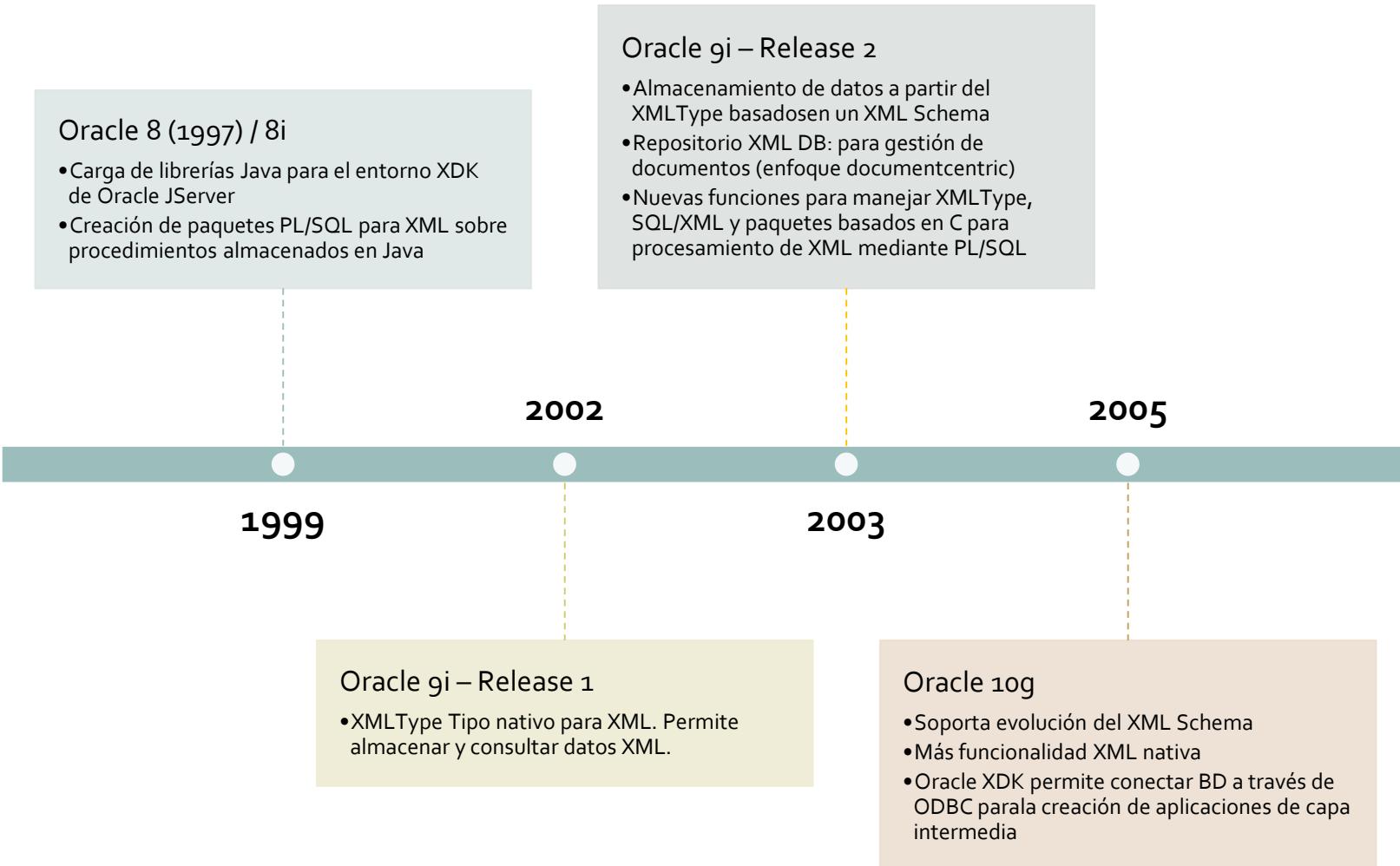
Productos comerciales

# ORACLE

- Capabilities

- Almacenamiento de documentos XML como columnas.
- Acceso a documentos XML en fuentes externas.
- Mapeo de elementos de documentos XML a tablas y columnas.
- En la versión 9i se incluye un tipo de dato (XMLType) para manejo nativo de XML.

# Oracle XML DB



# Almacenamiento

Dos opciones:

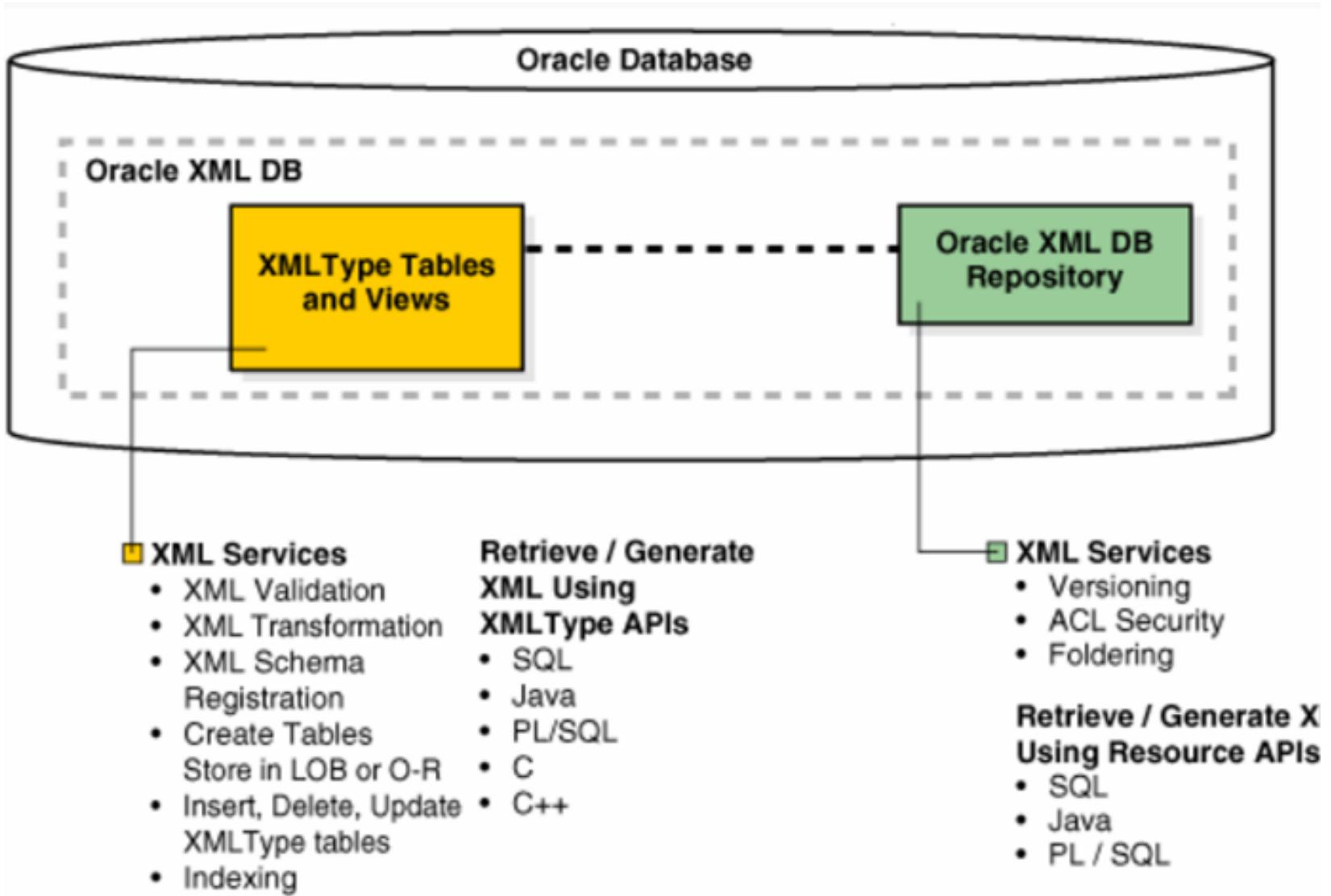
Repositorio de datos (Oracle XML DB Repository):

- Organizado jerárquicamente, consultable
  - Almacenamiento y visualización de contenido XML como un directorio jerárquico de carpetas
- Acceso a los documentos y representación de las relaciones entre documentos con:
  - XPath
  - URLs -> HTTP/FTP
  - SQL y PL/SQL

Tipo de dato nativo (XMLType)

- Permite definir tablas, columnas, parámetros, valores devueltos por funciones o variables en procedimientos PL/SQL
- Dispone de Funciones predefinidas para crear instancias XMLType, validar contenidos XML contra XML Schemas, aplicar hojas XSLT, etc.

Oracle XML DB  
Architecture:  
XMLType Storage and  
Repository



# Tabla con una columna XML

```
CREATE TABLE empleados (
    id CHAR(6),
    nombre VARCHAR(30),
    ...
    CV XMLType);
```

ID	LASTNAME	...	CV
123456	Pedro	.....	<?xml versión="1.0" ?><resume xmlns="http://www.cv.com/cv"> <nombre>...</nombre> <direccion>...</direccion> </resume>
876453	María	...	NULL
637596	Laura	....	<cv ref="http://www.banner.com/resume.html" />

# Columna XMLType

```
CREATE TABLE personas (id NUMBER PRIMARY KEY,  
nombre VARCHAR(20), direccion XMLTYPE);  
  
INSERT INTO personas(id,nombre,domicilio) VALUES(1,  
'Juan López', XMLTYPE('<direccion><calle>Justo  
Sierra#78</calle><Ciudad>Saltillo</Ciudad><Estado>C  
oahuila</Estado><CP>12345</CP></direccion>'));  
  
SELECT extract(domicilio, '/direccion/calle') FROM  
Personas;  
  
UPDATE personas SET domicilio = XMLTYPE('<direccion  
><calle>Benavente#254</calle><Ciudad>Pachuca</Ciuda  
d><Estado>Hidalgo</Estado><CP>22334</CP></direccion  
>') WHERE id = 1;
```

# XMLType: ¿Columna o Tabla?

- En Oracle, se pueden almacenar datos XML en

- Columnas **XMLType**

```
CREATE TABLE MiTabla (
    id NUMBER PRIMARY KEY,
    ...,
    xmlCol XMLTYPE);
```

- Tablas de objetos a partir del tipo **XMLType**

```
CREATE TABLE MiTablaXML OF XMLTYPE;
```

# Validación mediante XML Schema

- Para realizar la validación de un documento XML en Oracle Database Server, se debe primero registrar el documento XML Schema correspondiente. Para ello, es necesario crear un directorio en el SO y colocar allí los archivos XML y XMLSchema, y posteriormente “conectar” el servidor con esta carpeta (como usuario DBA), con el comando:

```
CREATE DIRECTORY DOCUMENTOS_XML AS '/ruta_archivosXML';
GRANT READ, WRITE ON DIRECTORY "DOCUMENTOS_XML" TO user;
```

- en donde **DOCUMENTOS\_XML** es el nombre asignado por el usuario al directorio, y *ruta\_archivosXML* es la ruta del SO en donde se localiza la carpeta de archivos. Ahora es posible ejecutar el procedimiento siguiente:

```
DBMS_XMLSHEMA.registerSchema (
SCHEMAURL => 'URL',
SCHEMADOC => BFILENAME ('DOCUMENTOS_XML', 'test.xsd'));
```

- en donde *URL* es una URL utilizada como identificador, y **test.xsd** es el archivo XMLSchema a registrar. Para revisar si se registró el esquema, se ejecuta:

```
SELECT SCHEMA_URL, LOCAL, XMLSERIALIZE (DOCUMENT SCHEMA AS VARCHAR(4000)) FROM
USER_XML_SCHEMAS;
```

- Es posible usar la dirección <http://localhost:8080/sys/schemas> para ver los esquemas registrados dentro de la base de datos.

# Creación de tablas XML

- Una vez que se tienen registrados los esquemas, se pueden crear tablas la cuales sean del tipo `XMLType` y realizar la validación de acuerdo a algún esquema previamente registrado. Suponga la creación de la tabla *empleados*:

```
CREATE TABLE empleados OF XMLType XMLSCHEMA "URL" ELEMENT "raiz_xml";
```

- en donde *URL* se refiere al identificador del esquema registrado previamente, y *raiz\_xml* es el nombre del elemento raíz del documento XML a validar.
- Si la tabla es relacional e incluye una columna de tipo XML, entonces se puede usar la siguiente construcción:

```
CREATE TABLE empleados (id NUMBER, documento_xml XMLType) XMLTYPE COLUMN  
documento_xml ELEMENT "URL#raiz_xml";
```

- en donde el carácter # separa la *url* del elemento raíz del documento XML.

# Creación de tablas XML

- Para verificar la creación de la tabla de tipo XML se puede consultar el diccionario de datos:

```
SELECT TABLE_NAME, XMLSCHEMA, SCHEMA_OWNER, ELEMENT_NAME FROM USER_XML_TABLES;
```

- Se cuentan con una serie de funciones informativas de los esquemas:

- `isSchemaBased()`, `isSchemaValid()`, `isSchemaValidated()` regresa 1 en caso de ser verdaderos, 0 en otro caso.
- `getSchemaURL()` obtiene la url del esquema asociado
- `schemaValidate()` realiza una validación de un documento XML

- Para verificar que un documento XML que se desea insertar sea válido con su XML Schema asociado:

```
▪ INSERT INTO tabla_xml VALUES (XMLType(BFILENAME ('DOCUMENTOS_XML', 'doc.xml'),  
nls_charset_id('AL32UTF8')));
```

- en donde `DOCUMENTOS_XML` es el directorio en donde residen los archivos XML, `doc.xml` es el nombre del archivo a insertar y `AL32UTF8` es el código de conjunto de caracteres local.

# Empleo de las funciones de validación

- Es posible hacer una validación XML Schema pre y post inserción del contenido XML en la base de datos.
- Como ejemplo de una prevalidación, se tiene la inclusión en un disparador, de la siguiente forma:

```
CREATE OR REPLACE TRIGGER validate_orders
    BEFORE INSERT ON orders FOR EACH ROW
BEGIN
    IF (:new.OBJECT_VALUE IS NOT NULL) THEN :new.OBJECT_VALUE.schemavalidate(); END IF;
END;
/
```

- También mediante una cláusula CHECK asociada a la tabla:

```
ALTER TABLE orders ADD CONSTRAINT chk_validate_orders CHECK (XMLIsValid(OBJECT_VALUE) = 1);
```

- La postvalidación verifica que el documento XML sea válido una vez que ha sido insertado en la base de datos:

```
SELECT o.xmlcol.isSchemaValid('http://www.example.com/schemas/ipo.xsd', 'Orders') FROM orders o;
```

- Funciones para manipulación de datos XML.  
Utilizan XPath para localizar ítems en una instancia XML.
  - EXTRACT()
  - EXTRACTVALUE()
  - EXISTSNODE()
  - XMLQUERY()
  - UPDATEXML()
  - DELETEXML()

SQL/XML:  
Funciones de  
manipulación  
de ORACLE

# EXTRACT()

- Selecciona un nodo individual y sus nodos hoja de una instancia XML.

**SELECT**

```
EXTRACT(domicilio, '/direccion').getstringval()
() AS salida FROM persona;
```

## SALIDA

```
<direccion><calle>Justo Sierra
#78</calle><Ciudad>Saltillo</Ciudad><Estado>
Coahuila</Estado><CP>12345</CP></direccion>
```

## EXTRACTVALUE()

- Extrae el valor de un nodo hoja. Devuelve un valor, no una instancia XML

```
SELECT extractValue(domicilio,  
'/direccion/calle') AS calle  
FROM persona;
```

**CALLE**

Justo Sierra #78

- Busca valores específicos en el nodo hoja, si existe devuelve true.

## EXISTSNODE()

```
SELECT count (*) FROM persona  
WHERE existsNode (domicilio,  
'/direccion [CP=12345]' ) = 1;
```

COUNT (\*)

1

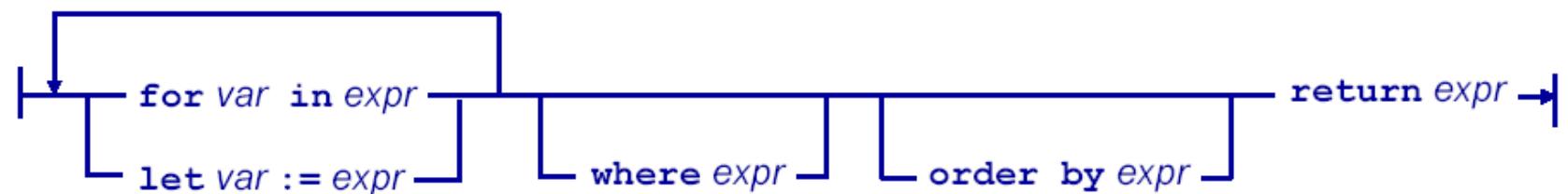
# XMLQUERY

- La función XMLQUERY devuelve un valor XML a partir de la evaluación de una expresión XQuery utilizando los argumentos de entrada especificados como variables XQuery.

```
SELECT id, nombre, XMLQuery (  
    'for $i in /direccion  
    where $i/CP = "12345"  
    return <detalles>  
        <CP codigo="{$i/CP}" />  
        <ciudad nombre="{$i/ciudad}" />  
    </detalles>'  
    PASSING domicilio RETURNING CONTENT)  
domicilio  
FROM personas;
```

# Expresiones FLWOR

- Una expresión FLWOR enlaza variables, las aplica a un predicado, y construye un nuevo resultado.



Las cláusulas FOR and LET generan una lista de tuplas de variables de límite, conservando el orden de entrada.

La cláusula WHERE aplica un predicado, eliminando algunas de las tuplas.

La cláusula ORDER BY impone un orden en las tuplas sobrevivientes.

La cláusula RETURN es ejecutada para cada tupla sobreviviente, generando una lista ordenada de salidas.

# Especificación de patrones (XPath)

/	Especifica el hijo inmediato
//	Selecciona a cualquier profundidad en el árbol
.	Selecciona el contexto actual
..	Selecciona el nodo padre
*	Selecciona todos los elementos del contexto actual
@	Selecciona un atributo
@*	Selecciona todos los atributos en el contexto actual.
:	Separador de namespaces
!	Aplica el método de información al nodo de referencia
()	Agrupa contenidos para precedencia
[]	Aplica un patrón de filtro

# Operadores Lógicos

\$and\$	&&	AND lógico
\$or\$		OR lógico
\$not\$		NO lógico
\$eq\$	=	Igual
\$ieq\$		Igual no sensitivo a mayúsculas
\$ne\$	!=	No igual
\$ine\$		No igual no sensitivo a mayúsculas
\$lt\$	<	Menor que
\$ilt\$		Menor que no sensitivo a mayúsculas
\$le\$	<=	Menor o igual que
\$ile\$		Menor o igual que no sensitivo a mayúsculas
\$gt\$	>	Mayor que
\$igt\$		Mayor que no sensitivo a mayúsculas
\$ge\$	>=	Mayor o igual que
\$ige\$		Mayor o igual que no sensitivo a mayúsculas
\$all\$		Operación de conjunto que regresa verdadero si la condición es verdadera para todos los items de la condición
\$any\$		Operación de conjunto que regresa verdadero si la condición es verdadera para cualquier item de la condición

# Funciones XPath

<code>boolean(arg1)</code>	Convierte el argumento <i>arg1</i> a un valor booleano
<code>concat(arg1, arg2, ...)</code>	Concatena los argumentos como una cadena de caracteres
<code>contains(arg1, arg2)</code>	Prueba cuando <i>arg1</i> contiene a <i>arg2</i> como subcadena. Es case-sensitive
<code>count(arg1)</code>	Regresa el número de nodos que existen en el conjunto de nodos dado por <i>arg1</i>
<code>document(arg1)</code>	Busca un documento XML externo resolviendo la URL de referencia dada en <i>arg1</i> y regresa su nodo raíz.
<code>false()</code>	Comprueba el valor falso
<code>id(arg1)</code>	Regresa el conjunto de nodos contenido los nodos del documento que posee el id pasado como <i>arg1</i>
<code>key(arg1, arg2)</code>	Es usado para encontrar el nodo con el valor dado por la llave en <i>arg1</i> , definido en un elemento <code>&lt;xsl:key&gt;</code> . Arg2 es el valor que se están buscando para dicha llave
<code>local-name([arg1])</code>	Regresa la parte local del nombre de un nodo
<code>name([arg1])</code>	Regresa el nombre calificado de un nodo, incluyendo el <i>namespace</i>
<code>not(arg1)</code>	Niega el valor de <i>arg1</i>
<code>number(arg1)</code>	Convierte su argumento en un valor numérico. Si se omite <i>arg1</i> , toma el valor del nodo actual.
<code>position()</code>	Regresa el valor de la posición del nodo contexto.
<code>string([arg1])</code>	Convierte <i>arg1</i> a su valor de cadena de caracteres
<code>string-length(arg1)</code>	Regresa el número de caracteres de <i>arg1</i>
<code>substring(arg1, arg2, [arg3])</code>	Regresa parte de <i>arg1</i> , partir de la posición dada por <i>arg2</i> , y opcionalmente de longitud dada por <i>arg3</i>
<code>sum(arg1)</code>	Calcula el valor total de la suma de los valores de los nodos dados por el conjunto de nodos <i>arg1</i>
<code>true()</code>	Regresa el valor booleano verdadero (true)

# Ejemplos

```
<alumnos>
  <alumno NoBoleta="12345">
    <nombre>Juan</nombre>
    <nombre>Carlos</nombre>
    <apellidos>
      <paterno>Lopez</paterno>
      <materno>Perez</materno>
    </apellidos>
    <genero>M</genero>
    <edad>21</edad>
  </alumno>
</alumnos>
```

```
for $x in /alumnos/alumno
return $x

<alumno NoBoleta="12345">
  <nombre>Juan</nombre>
  <nombre>Carlos</nombre>
  <apellidos>
    <paterno>Lopez</paterno>
    <materno>Perez</materno>
  </apellidos>
  <genero>M</genero>
  <edad>21</edad>
</alumno>
```

```
for $x in //nombre
return $x
<nombre>Juan</nombre>
<nombre>Carlos</nombre>

for $x in //nombre
return <salida>{$x}</salida>
<salida>
  <nombre>Juan</nombre>
</salida>
<salida>
  <nombre>Carlos</nombre>
</salida>
```

```
<salida>
{
for $x in //nombre
return $x
}
</salida>
<salida>
<nombre>Juan</nombre>
<nombre>Carlos</nombre>
</salida>

<salida>
{
for $x in //nombre
return data($x)
}
</salida>
<salida>Juan Carlos</salida>
```

```
for $x in /alumnos
let $y := $x/alumno
return $y
<alumno NoBoleta="12345">
  <nombre>Juan</nombre>
  <nombre>Carlos</nombre>
  <apellidos>
    <paterno>Lopez</paterno>
    <materno>Perez</materno>
  </apellidos>
  <genero>M</genero>
  <edad>21</edad>
</alumno>
```

```
for $x in /alumnos
let $y := $x/alumno
return data($y)
JuanCarlosLopezPerezM21
```

```
for $x in /alumnos/alumno[@NoBoleta=12345]
return $x
<alumno NoBoleta="12345">
  <nombre>Juan</nombre>
  <nombre>Carlos</nombre>
  <apellidos>
    <paterno>Lopez</paterno>
    <materno>Perez</materno>
  </apellidos>
  <genero>M</genero>
  <edad>21</edad>
</alumno>
```

```
for $x in /alumnos/alumno  
where $x/@NoBoleta <= 12345  
return data($x)
```

JuanCarlosLopezPerezM21

```
for $x in /alumnos/alumno  
where $x/edad >= 18 and $x/edad <= 60  
return data($x)
```

JuanCarlosLopezPerezM21

```
for $x in /alumnos/alumno  
where $x/@NoBoleta <= 12345  
order by $x/apellidos/paterno,  
$x/apellidos/materno  
return data($x)
```

JuanCarlosLopezPerezM21

```
for $x in /alumnos  
where $x/alumno/nombre[1]  
return $x/alumno/nombre/text()
```

Juan  
Carlos

```
for $x in /alumnos  
where $x/alumno/nombre[1]  
return $x/alumno/nombre/upper-  
case(text())
```

JUAN  
CARLOS

```
for $x in /alumnos/alumno/nombre  
return if(starts-with($x, 'J'))  
then $x/lower-case(text())  
else $x/upper-case(text())
```

juan  
CARLOS

# XMLQuery()

```
SELECT id, nombre, XMLQuery( 'for $i in
/direccion where $i/CP = "12345" return
<detalles><CP codigo="{$i/CP}" /><ciudad
nombre="{$i/ciudad}" /><ciudad>{if ($i/ciudad =
"Saltillo") then "correcto" else
"incorrecto"}</ciudad><estado>{if ($i/estado =
"Coahuila") then "correcto" else
"incorrecto"}</estado></detalles>' PASSING
domicilio RETURNING CONTENT) domicilio FROM
personas;
```

ID	NOMBRE	PERSONA
1	John Smith	<detalles><CP codigo="12345"></CP><ciudad nombre="Saltillo"></ciudad><ciudad>correcto</ciudad> <estado>correcto</estado></detalles>

## UpdateXML()

---

Función que permite la actualización parcial de un documento almacenado como un valor XMLType.

---

Permite realizar múltiples cambios en una sola operación.

---

Cada cambio consiste en una expresión XPath que identifica el nodo a ser actualizado y el nuevo valor para ese nodo.

# Ejemplos

```
UPDATE persona SET domicilio =
updateXML(domicilio, '/direccion/ciudad/text()', 'Nogales',
'/direccion/estado/text()', 'Sonora')
WHERE existsNode(domicilio, '/direccion[CP=12345]' ) = 1;

UPDATE person SET domicilio =
updateXML(domicilio, '/direccion', XMLType ('<direccion><calle>Av
. Juarez #432</calle><ciudad>Nogales
</ciudad><estado>Sonora</estado><CP>23456</CP> </direccion>'))
WHERE existsNode(domicilio, '/direccion[CP=12345]' ) = 1;
```

# DeleteXML()

- Borra un nodo de cualquier clase

```
UPDATE person SET domicilio  
= deleteXML(domicilio, '/direccion/ciudad') WHERE exists  
Node(domicilio, '/direccion[CP="12345"]') = 1;
```

# SQL/XML: Funciones del estándar suministradas por ORACLE

- Funciones para generar datos XML con datos procedentes de una base de datos relacional:
  - XMLPARSE
  - XMLSERIALIZE
  - XMLELEMENT
  - XMLATTRIBUTES
  - XMLFOREST
  - XMLCONCAT
  - XMLAGG
  - XMLPI
  - XMLCOMMENT

## SQL/XML: Funciones del estándar

- Funciones del tipo de dato XML:
  - **XMLPARSE** - Convierte una cadena de caracteres que contiene datos XML en un valor (instancia) de tipo XML
  - **XMLSERIALIZE** - Obtiene una representación en string o LOB de un dato de tipo XML

```
INSERT INTO empleados VALUES ('123456', 'López', ...,
XMLPARSE (DOCUMENT '<?xml versión="1.0"?><cv
xmlns="http://www.cv.com/cv"><nombre>...</nombre><direccion>...</d
irección></cv>' PRESERVE WITHSPACE));
SELECT e.id, XMLSERIALIZE (DOCUMENT e.cv AS VARCHAR(2000)) AS
cv FROM empleados e WHERE e.id = '123456';
```

<u>ID</u>	<u>RESUME</u>
123456	<?xml versión="1.0" ? <cv xmlns="http://www.cv.com/cv"> <nombre>...</nombre> <direccion>...</direccion> </cv>

# XMLElement

- Devuelve un valor XML dado por:
  - Un identificador SQL que actúa como su *name*
  - Una lista opcional de declaraciones *namespace*
  - Una lista opcional de nombres y valores de sus atributos
  - Una lista opcional de expresiones que suministran su contenido

```
XMLELEMENT ( NAME etiqueta [ ,  
namespace] [ , atributos] [ { ,  
contenido } . . . ] )
```

# Ejemplo

```
SELECT e.employee_id,  
XMLELEMENT (NAME "NombreEmpleado",  
e.first_name) AS resultadoEnXML  
FROM HR.employees e
```

## ID

100	<NombreEmpleado>Steven</NombreEmpleado>
101	<NombreEmpleado>John</NombreEmpleado>
102	<NombreEmpleado>Paul</NombreEmpleado>

## Ejemplo

```
SELECT XMLEMENT ("Emp",
XMLEMENT ("NombreEmpleado",
e.first_name || ' ' || e.last_name ) ,  

XMLEMENT ("e-mail", e.email ) ) AS
ResultadoEnXML
FROM hr.employees e;
```

---

### RESULTADOENXML

```
<Emp><NombreEmpleado>Steven King</NombreEmpleado><e-
mail>kings@mail.com</e-mail></Emp>
```

```
<Emp><NombreEmpleado>John Smith</NombreEmpleado><e-
mail>smithj@mail.com</e-mail></Emp>
```

```
<Emp><NombreEmpleado>Paul Singer</NombreEmpleado><e-
mail>singerp@mail.com</e-mail></Emp>
```

# Ejemplo

```
SELECT XMLEMENT (name "Dpto",
                  XMLEMENT ("NombreDpto",
                             d.department_name),
                  XMLEMENT ("trabajadores", (select count(*)
                                              from employees e where e.department_id =
                                              d.department_id) ) ) AS ResultadoEnXML
FROM hr.departments d;
```

## RESULTADOENXML

```
<Dpto><NombreDpto>Administration</NombreDpto><trabajadores>1</trabajadores></Dpto>

<Dpto><NombreDpto>Marketing</NombreDpto><trabajadores>2</trabajadores></Dpto>

<Dpto><NombreDpto>Human Resources</NombreDpto><trabajadores>6</trabajadores></Dpto>
```

## Ejemplo

```
SELECT XMLELEMENT ("NombreEmpleado",  
XMLATTRIBUTES (e.email AS "eMail"),  
e.first_name || ' ' || e.last_name ) AS  
Resultado  
FROM hr.employees e;
```

### RESULTADO

```
<NombreEmpleado eMail="kings@mail.com">Steven King</NombreEmpleado>
```

```
<NombreEmpleado eMail="smithj@mail.com">John Smith</NombreEmpleado>
```

```
<NombreEmpleado eMail="singerp@mail.com">Paul Singer</NombreEmpleado>
```

## Ejemplo

```
SELECT XMLEMENT
  ("gestion:NombreEmpleado",
   XMLNAMESPACES ('http://www.gestion.com' AS
 "gestion"), XMLATTRIBUTES (e.email AS
 "gestion:eMail"), e.first_name || ' ' ||
 e.last_name ) AS resultado
FROM hr.employees e;
```

### RESULTADO

```
<gestion:NombreEmpleado xmlns="http://www.gestion.com" gestion:e-
mail="kings@mail.com">Steven King</gestion:NombreEmpleado>
```

```
<gestion:NombreEmpleado xmlns="http://www.gestion.com" gestion:e-
mail="smithj@mail.com">John Smith</gestion:NombreEmpleado>
```

```
<gestion:NombreEmpleado xmlns="http://www.gestion.com" gestion:e-
mail="singerp@mail.com">Paul Singer</gestion:NombreEmpleado>
```

## XMLConcat

- Produce un valor XML dado dos o más expresiones de tipo XML
- Si alguna de las expresiones se evalúa como nulo, es ignorada

```
SELECT XMLCONCAT (  
    XMLELEMENT ("NombreEmpleado", e.first_name),  
    XMLELEMENT ("Apellido", e.last_name ) ) AS  
    Resultado  
FROM hr.employees e;
```

### RESULTADO

```
<NombreEmpleado>Steven</NombreEmpleado><Apellido>King</Apellido>  
  
<NombreEmpleado>John</NombreEmpleado><Apellido>Smith</Apellido>  
  
<NombreEmpleado>Paul</NombreEmpleado><Apellido>Singer</Apellido>
```

## XMLForest

- Produce una secuencia de elementos XML de los argumentos que se le pasan.
- XMLFOREST permite realizar consultas de forma más compacta, y además tiene como ventaja con respecto a XMLEMENT que elimina los nulos. Sin embargo, no permite incluir atributos.

```
SELECT XMLFOREST (e.first_name AS "Nombre", e.email AS
"e-mail")  
FROM hr.employees e
```

### RESULTADO

```
<Nombre>Steven</Nombre><e-mail>kings@mail.com</e-mail>  
  
<Nombre>John</Nombre><e-mail>smithj@mail.com</e-mail>  
  
<Nombre>Paul</Nombre><e-mail>singerp@mail.com</e-mail>
```

XMLPI

- Permite generar instrucciones de procesamiento

```
SELECT XMLPI (NAME "OrderAnalysisComp",
               'imported, reconfigured, disassembled')
      AS pi FROM DUAL;
```

PI

```
<?OrderAnalysisComp imported, reconfigured, disassembled?>
```

## XMLComment

- Permite crear un comentario
- ```
SELECT XMLComment ('Texto del  
comentario')  
AS cmnt FROM DUAL;
```
- 
- ```
<!--Texto del comentario -->
```

# XMLAgg

- Función de agregación similar a SUM, AVG, etc.

```
SELECT e.department_id,  
       XMLEMENT ("Dpto", XMLELAGG (  
           XMLEMENT ("NombreEmpleado", e.first_name )  
           ORDER BY e.first_name) )  
FROM hr.employees e  
GROUP BY e.department_id;
```

ID	RESULTADO
100	<Dpto><NombreEmpleado>John</NombreEmpleado></Dpto>
101	<Dpto><NombreEmpleado>Alan</NombreEmpleado><NombreEmpleado>Jennifer</NombreEmpleado><NombreEmpleado>Mitch</NombreEmpleado></Dpto>
102	<Dpto><NombreEmpleado>Michelle</NombreEmpleado><NombreEmpleado>Susan</NombreEmpleado></Dpto>

# SQL/XML: Funciones propias de ORACLE

- Funciones para generar datos XML con datos procedentes de la BD relacional:
  - [SYS\\_XMLGEN\(\)](#)
  - [XMLSEQUENCE\(\)](#)
  - [XMLCOLATTVAL\(\)](#)
  - [SYS\\_XMLAGG\(\)](#)

## SYS\_XMLGEN()

- Similar a la función XMLElement(), pero en este caso, la función recibe un único argumento y devuelve un documento XML bien formado.

```
SELECT SYS_XMLGen (XMLEMENT ("Empleado",
(XMLEMENT ("NombreEmpleado", XMLATTRIBUTES
(e.email), e.first name ||' '|| e.last name)),
XMLEMENT ("Departamento", e.department_id), XMLEMENT ("Telefono", e.phone_number))) AS xml FROM
hr.employees e;
```

---

### XML

```
<?xml versión="1.0?><ROW><Empleado><NombreEmpleado email="Sking@mail.com">Steven
King</NombreEmpleado>
<Departamento>90</Departamento><Telefono>515.123.4567</Telefono></Empleado></ROW>
<?xml versión="1.0?><ROW><Empleado><NombreEmpleado email="Nkochar@mail.com">Neena
Kochar</NombreEmpleado>
<Departamento>60</Departamento><Telefono>515.123.4568</Telefono></Empleado></ROW>
<?xml versión="1.0?><ROW><Empleado><NombreEmpleado email="Ldehan@mail.com">Lex de
Han</NombreEmpleado>
<Departamento>60</Departamento><Telefono>515.123.4569</Telefono></Empleado></ROW>
<?xml versión="1.0?><ROW><Empleado><NombreEmpleado email="Psmith@mail.com">Paul
Smith</NombreEmpleado>
<Departamento>90</Departamento><Telefono>515.123.4570</Telefono></Empleado></ROW>
```

## XMLSEQUENCE()

- Realiza la función inversa de SYS\_XMLGen. Devuelve un varray de instancias de XMLType. Al devolver una colección, se debe utilizar en el FROM de la consulta

```
SELECT * FROM dual
(XMLSequence (EXTRACT (XMLType ('<A><B>V1</B><B>V2</B><B>V3</B></A>') , '/A/B')) ) AS
tabla;
```

### COLUMN\_VALUE

```
<B>V1</B>
<B>V2</B>
<B>V3</B>
```

## XMLCOLATTVAL()

- Crea un fragmento XML, con etiqueta COLUMN y un atributo NAME, que lo iguala al nombre de la columna. Podemos cambiar el valor del atributo mediante el alias

```
SELECT XMLCOLATTVAL(e.first_name as  
nombre) FROM hr.employees e;
```

```
XMLCOLATTVAL (E.FIRST_NAME AS NOMBRE)
```

```
<column name="NOMBRE">Ellen</column>  
<column name="NOMBRE">Sundar</column>  
<column name="NOMBRE">Mozhe</column>  
<column name="NOMBRE">David</column>  
<column name="NOMBRE">Shelli</column>  
<column name="NOMBRE">Amit</column>
```

- Agrega todas las instancias XML que se le pasan como parámetro y devuelve un documento XML

## SYS\_XMLAGG()

```
SELECT SYS_XMLAGG(XMLELEMENT  
("NombreEmpleado", e.first_name  
|| ' ' || e.last_name)) AS xml FROM  
hr.employees e;
```

XML

```
<?xml versión="1.0"?><ROWSET><NombreEmpleado>Ellen  
Abel</NombreEmpleado> <NombreEmpleado>Sundar  
Ande</NombreEmpleado><NombreEmpleado>Mozhe  
Atkinson</NombreEmpleado><NombreEmpleado>Paul  
Smith</NombreEmpleado></ROWSET>
```

# Almacenamiento de datos con estructura compleja

- Muchas aplicaciones necesitan almacenar datos estructurados, pero no se modelan fácilmente como relaciones.
- Prefieren almacenar información en formato XML.
- Se han desarrollado normas basadas en XML para la representación de datos en XML de una gran variedad de aplicaciones especializadas, por ejemplo, ChemML (industria química), MathML (notación matemática), GML (sistemas geográficos), BPEL (notación de flujos de procesos), entre muchas más.

# JSON

---

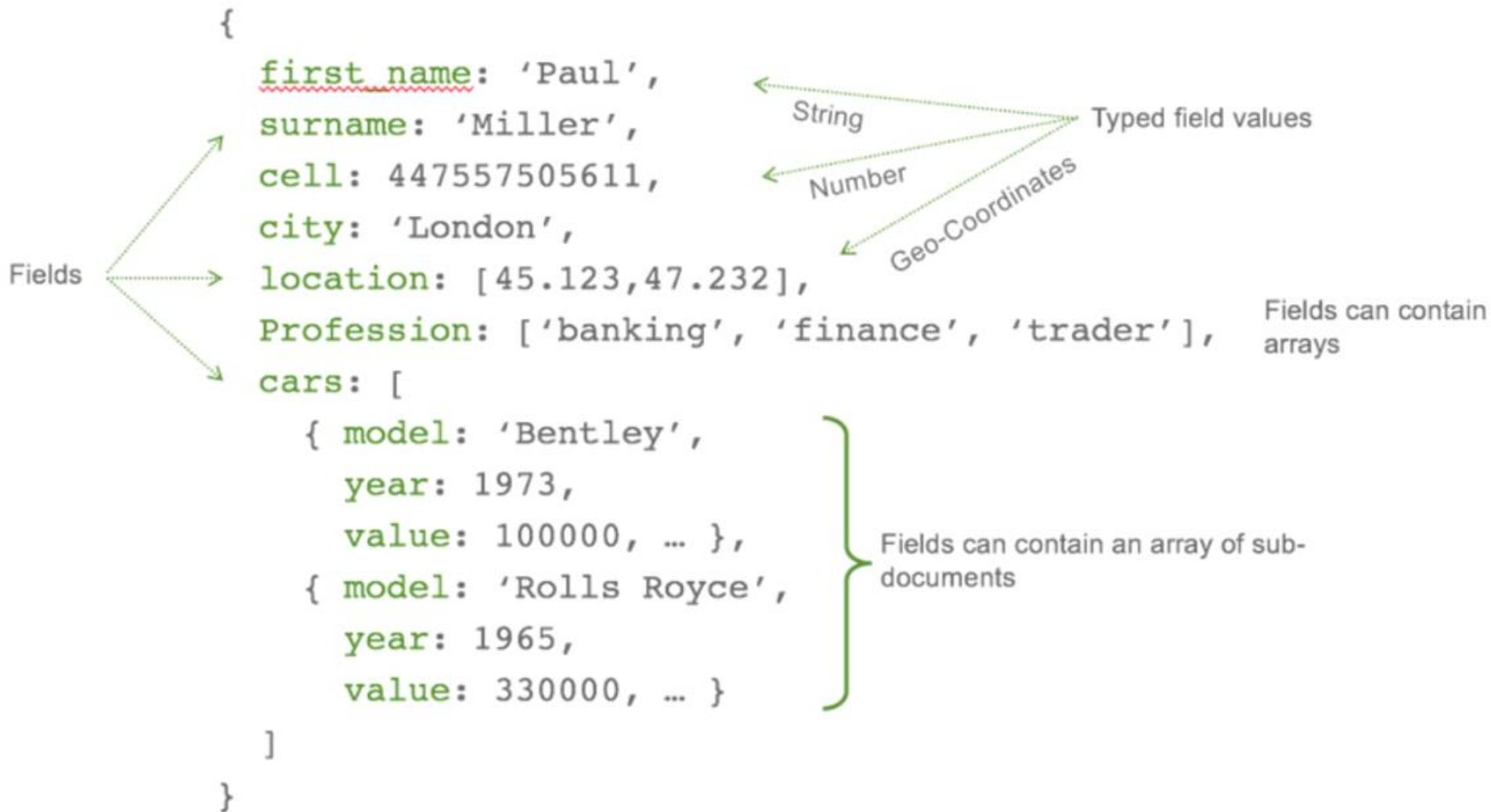
- **JSON** (acrónimo de **JavaScript Object Notation**, 'notación de objeto de JavaScript') es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera un formato independiente del lenguaje.
- Utiliza texto legible por humanos para almacenar y transmitir objetos de datos que consisten en pares atributo-valor y arreglos.
- Douglas Crockford especificó originalmente el formato JSON a principios de la década de 2000. Él y Chip Morningstar enviaron el primer mensaje JSON en abril de 2001.

# Reglas de sintaxis JSON

---

- La sintaxis JSON se deriva de la sintaxis de notación de objetos JavaScript
- Datos JSON: un nombre y un valor
  - Los datos JSON se escriben como pares nombre/valor (también conocidos como pares clave/valor).
  - Un par nombre/valor consiste en un nombre de campo (entre comillas dobles), seguido de dos puntos, seguido de un valor:  
`"nombre": "Juan"`
  - Los nombres JSON requieren comillas dobles.

# Estructura de JSON



# Tipos de datos

---

- Los tipos de datos básicos de JSON son:
  - **Número**: un número decimal con signo que puede contener una parte fraccionaria y puede usar la notación E exponencial. El formato no hace distinción entre entero y punto flotante.
  - **Cadena** : una secuencia de cero o más caracteres Unicode . Las cadenas están delimitadas con comillas dobles y admiten una sintaxis de escape de barra invertida .
  - **Booleano** : cualquiera de los valores **true** o **false**
  - **Array** : una lista ordenada de cero o más elementos, cada uno de los cuales puede ser de cualquier tipo. Los arreglos utilizan la notación de corchetes con elementos separados por comas.
  - **Objeto** : una colección de pares de nombre y valor donde los nombres (también llamados claves) son cadenas. Los objetos se delimitan con corchetes y usan comas para separar cada par, mientras que dentro de cada par el carácter de dos puntos ':' separa la clave o el nombre de su valor.
  - **null**: un valor vacío, usando la palabra **null**

Cadenas	Las cadenas en JSON deben escribirse entre comillas dobles.
	{ "nombre": "Juan" }
Números	Los números en JSON deben ser un entero o un coma flotante, sin comillas.
	{ "edad": 30 }
Objetos	Los valores en JSON pueden ser objetos.
	{ "empleado": { "nombre": "Juan", "edad": 30, "ciudad": "CDMX" } }
Arreglos	Los valores en JSON pueden ser arreglos.
	{ "empleados": [ "Juan", "Ana", "Pedro" ] }
Booleanos	Los valores lógicos pueden ser true/false.
	{ "venta": true }
Nulos	Los valores de atributos pueden ser nulos.
	{ "apellido": null }

## **JSON es como XML porque**

Tanto JSON como XML son "autodescriptibles" (legibles por humanos)

Tanto JSON como XML son jerárquicos (valores dentro de valores)

Tanto JSON como XML pueden ser analizados y utilizados por muchos lenguajes de programación

## **JSON es diferente a XML porque**

JSON no usa la etiqueta final de cierre

JSON es más corto

JSON es más rápido de leer y escribir

JSON puede usar matrices

## **La mayor diferencia es:**

XML debe analizarse con un analizador XML.

JSON puede ser analizado por una función JavaScript estándar (eval).

## **Porque JSON es mejor que XML**

XML es más difícil de analizar que JSON. JSON se analiza en un objeto JavaScript listo para usar.

Para las aplicaciones AJAX, JSON es más rápido y fácil que XML:

## **Uso de XML**

Obtener un documento XML

Utilizar el DOM XML para recorrer el documento en bucle

Extraer valores y almacenar en variables

## **Uso de JSON**

Obtener una cadena JSON

Analizar la cadena JSON

# MongoDB

---

- Lanzado por primera vez públicamente en 2009, MongoDB (a menudo llamado *mongo*) se convirtió rápidamente en una de las bases de datos NoSQL más amplias y utilizadas que existen para el desarrollo de aplicaciones Web.
- MongoDB fue diseñado como una base de datos escalable, el nombre Mongo proviene de "humongous" (extremadamente grande), con rendimiento y fácil acceso a los datos como objetivos principales.
- Es una base de datos de documentos, que le permite almacenar objetos anidados a la profundidad que requerida y consultarlos.
- No se basa en ningún esquema, por lo que los documentos pueden contener campos o tipos que ningún otro documento de la colección contiene.

# Ventajas de MongoDB sobre RDBMS

- **Sin esquema.** MongoDB es una base de datos de documentos en la que una colección contiene diferentes documentos. El número de campos, el contenido y el tamaño del documento pueden diferir de un documento a otro.
- **Capacidad de consulta profunda.** MongoDB admite consultas dinámicas en documentos utilizando un lenguaje de consulta basado en documentos que es casi tan potente como SQL.
- **Facilidad de escalabilidad horizontal:** MongoDB es fácil de escalar.

# Formato de almacenamiento

- Mongo es una base de datos de documentos JSON (aunque técnicamente los datos se almacenan en una forma binaria de JSON conocida como BSON).
- Un documento de Mongo se puede comparar con una fila de tabla relacional sin un esquema, cuyos valores pueden anidarse a una profundidad arbitraria.

```
{  
  "_id" : ObjectId("4d0b6da3bb30773266f39fea"),  
  "country" : {  
    "$ref" : "countries",  
    "$id" : ObjectId("4d0e6074deb8995216a8309e")  
  },  
  "famous_for" : [  
    "beer",  
    "food"  
  ],  
  "last_census" : "Sun Jan 07 2018 00:00:00 GMT -0700 (PDT)",  
  "mayor" : {  
    "name" : "Ted Wheeler",  
    "party" : "D"  
  },  
  "name" : "Portland",  
  "population" : 582000,  
  "state" : "OR"  
}
```

JSON document

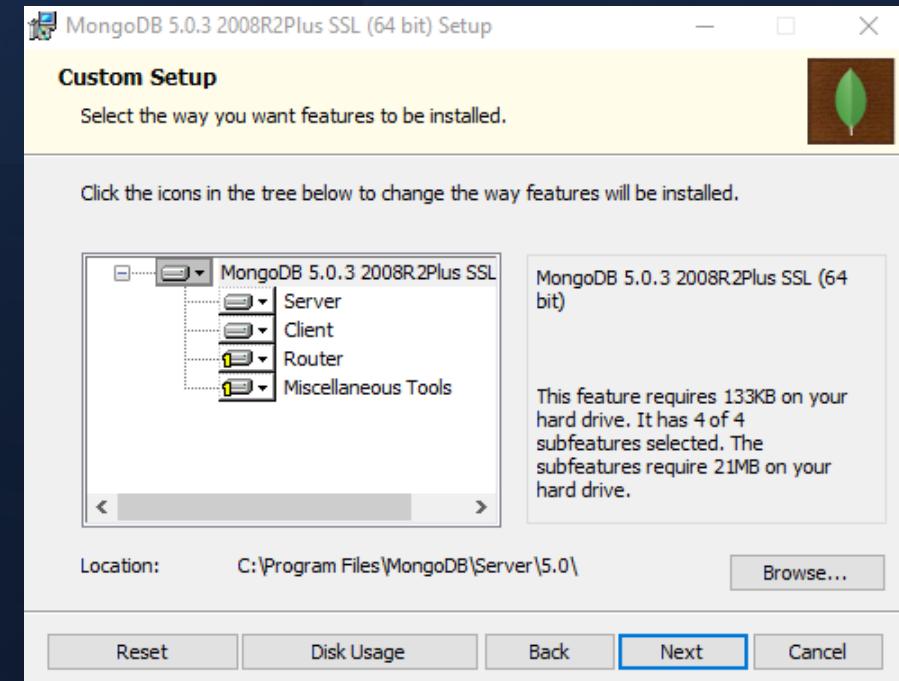
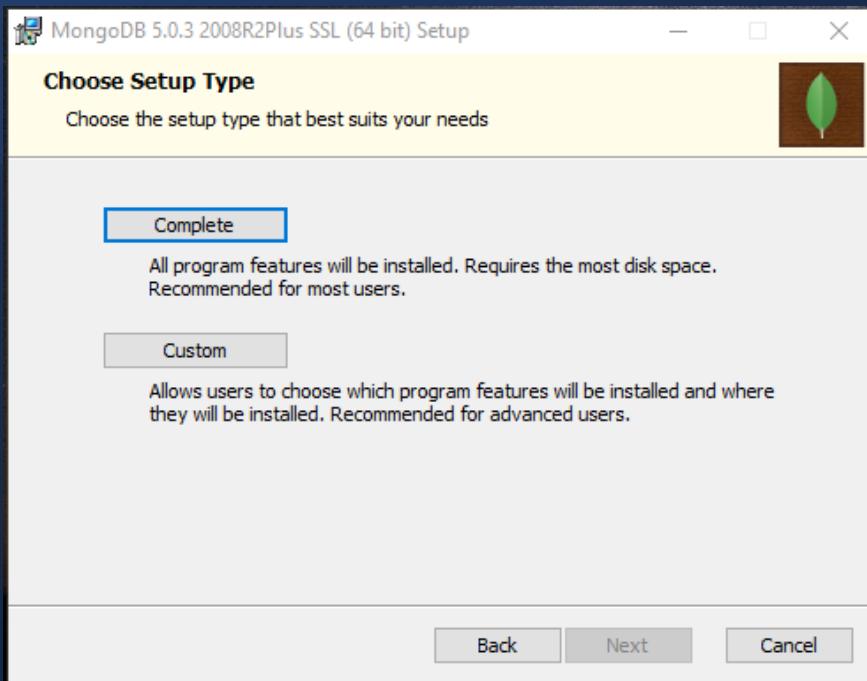
# Contenido de la base de datos

- **Base de datos**
  - La base de datos es un contenedor físico para colecciones.
  - Cada base de datos obtiene su propio conjunto de archivos en el sistema de archivos.
  - Un servidor MongoDB normalmente tiene varias bases de datos.
- **Colección**
  - La colección es un grupo de documentos de MongoDB.
  - Existe una colección dentro de una base de datos.
  - Las colecciones no aplican un esquema.
  - Los documentos dentro de una colección pueden tener diferentes campos.
  - Normalmente, todos los documentos de una colección tienen un propósito similar o relacionado.
- **Documento**
  - Un documento es un conjunto de pares clave-valor.
  - Los documentos tienen un esquema dinámico (los documentos de la misma colección no necesitan tener el mismo conjunto de atributos o estructura, y los atributos comunes en los documentos de una colección pueden contener diferentes tipos de datos).

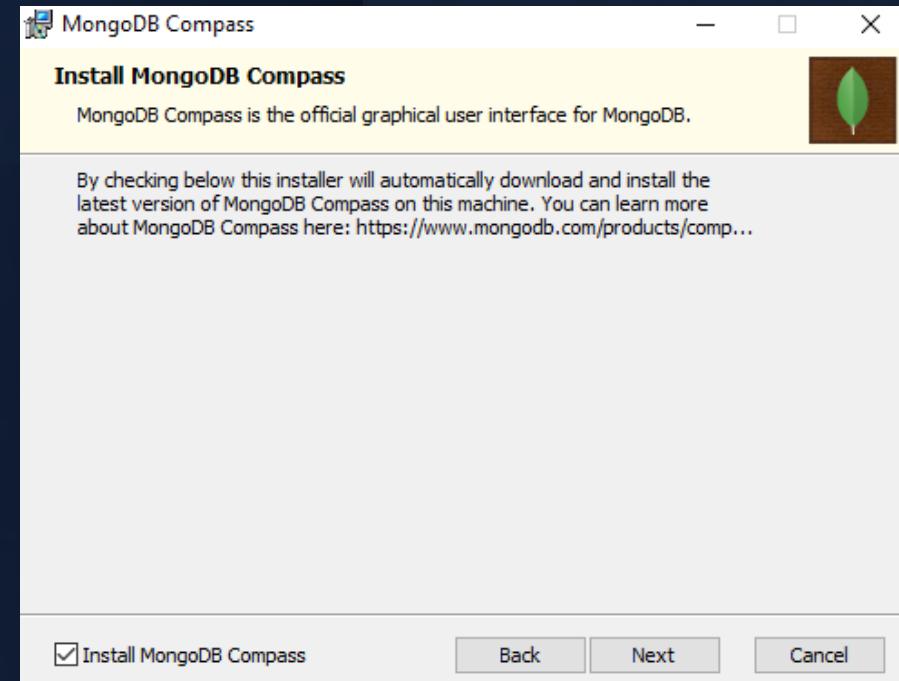
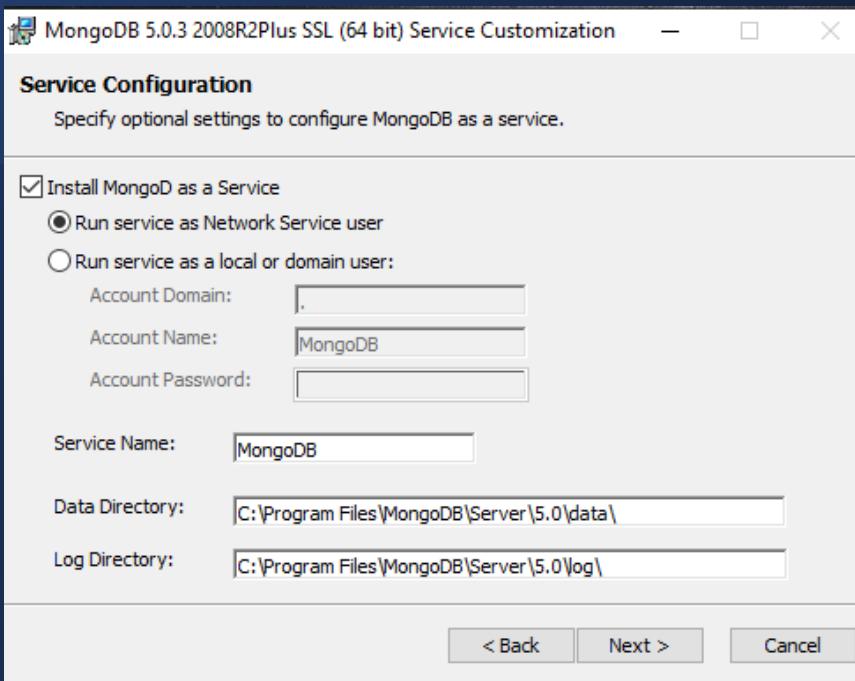
# Relación de concepto con respecto al modelo relacional

RDBMS	MongoDB
Base de datos	Base de datos
Tabla	Colección
Tupla/Fila	Documento
Columna	Atributo
Reunión de tablas	Documentos incrustados
Clave principal	Clave principal (clave predeterminada <code>_id</code> proporcionada por Mongo)

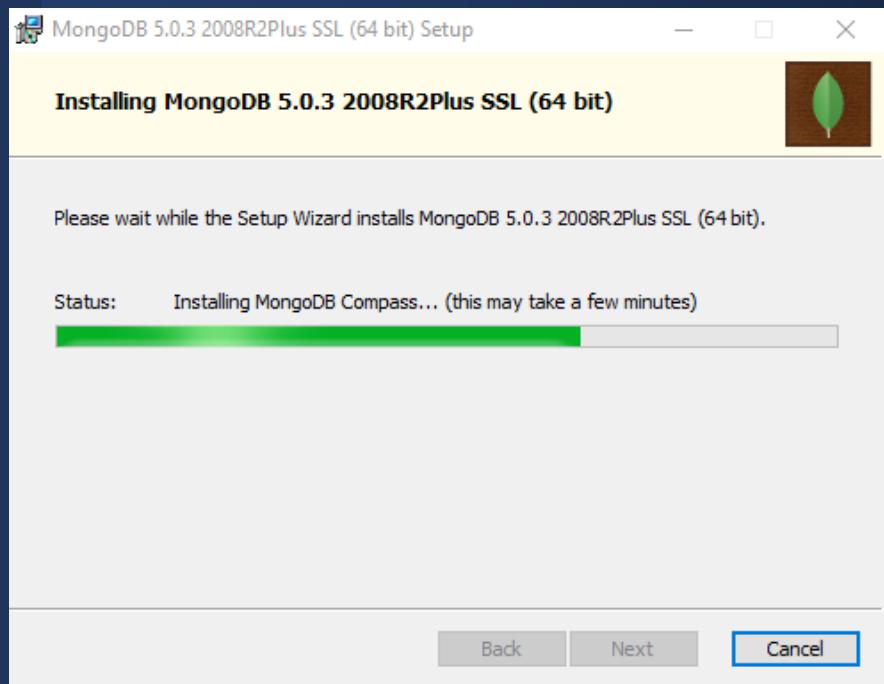
# Proceso de instalación



# Continuación



# Finalización y ejecución



Administrador de tareas						
Procesos		Rendimiento	Histórico de aplicaciones	Inicio	Usuarios	
Nombre	PID	Descripción			Estado	Grupo
MapsBroker		Administrador de mapas descargados			Detenido	NetworkService
McpManagementService		McpManagementService			Detenido	McpManage...
MessagingService		MessagingService			Detenido	UnistackSvGr...
MessagingService_19800ae		MessagingService_19800ae			Detenido	UnistackSvGr...
MicrosoftEdgeElevationService...		Microsoft Edge Elevation Service (MicrosoftEdgeElevationService)			Detenido	
MixedRealityOpenXRService		Windows Mixed Reality OpenXR Service			Detenido	
MongoDB	7872	MongoDB Server (MongoDB)			En ejecución	LocalSystemN...
mpssvc	2828	Firewall de Windows Defender			En ejecución	LocalServiceN...
MSDTC		Coordinador de transacciones distribuidas			Detenido	
MSiSCSI		Servicio del iniciador iSCSI de Microsoft			Detenido	netsvcs
msiserver		Windows Installer			Detenido	
MsKeyboardFilter		Filtro de teclado de Microsoft			Detenido	netsvcs
NaturalAuthentication		Autenticación natural			Detenido	netsvcs
NcaSvc		Asistente para la conectividad de red			Detenido	NetSvcs
NcbService		Agente de conexión de red			En ejecución	LocalSystemN...
NcdAutoSetup		Configuración automática de dispositivos conectados a la red			Detenido	LocalServiceN...
Netlogon		Net Logon			Detenido	
Netman		Conexiones de red			Detenido	LocalSystemN...
netprofm		Servicio de lista de redes			En ejecución	LocalService
NetSetupSvc	1320	Servicio de configuración de red			Detenido	netsvcs
NetTcpPortSharing		Servicio de uso compartido de puertos Net.Tcp			Detenido	
NgcCtnrSvc	6088	Contenedor de Microsoft Passport			En ejecución	LocalServiceN...
NgcSvc	11888	Microsoft Passport			En ejecución	LocalSystemN...
NlaSvc	1952	Reconoc. ubicación de red			En ejecución	NetworkService
nsi		Servicio Interfaz de almacenamiento en red			En ejecución	LocalService
OneSyncSvc	1820	Sincronizar host			Detenido	UnistackSvGr...
OneSyncSvc_19800ae	11692	Sincronizar host_19800ae			En ejecución	UnistackSvGr...
OracleJobSchedulerXE		OracleJobSchedulerXE			Detenido	
OracleMTSRecoveryService		OracleMTSRecoveryService			Detenido	
OracleServiceXE	6928	OracleServiceXE			En ejecución	
OracleXEClrAgent		OracleXEClrAgent			Detenido	

# Inicialización

- Antes de comenzar, es necesario crear una carpeta (en Windows), en donde se almacenarán los archivos de la base de datos. En una terminal del sistema puede ejecutar los siguientes comandos:

C:\md data

C:\md data\db

- Este es el valor por omisión y se recomienda mantenerlo.
- Para iniciar el servidor, es necesario ir a la ruta de instalación y ejecutar el archivo *mongod.exe*

C:\[ruta\_instalacion]\bin\mongod.exe

- Dejar minimizada esta ventana (no cerrar). Abrir otra terminal y ejecutar el cliente:

C:\[ruta\_instalacion]\bin\mongo.exe [nombre\_base\_datos]

MongoDB shell version v5.0.8

connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb

Implicit session: session { "id" : UUID("cf4e7331-61cc-4e01-9ee1-2f292c52cf97") }

MongoDB server version: 5.0.8

=====

- A partir de aquí comienza la interacción con MongoDB
- El comando `db.help()` regresa un listado de los comandos disponibles en el intérprete de MongoDB

# Creación de una base de datos

- El comando `show dbs` muestra todas las bases de datos del sistema. El comando `use` permite utilizar (cambiarse a ) otra base de datos. Se puede verificar la base de datos actual con el comando `db` .
- Crear una colección en Mongo es posible con solo añadir un registro inicial a dicha colección. Debido a que Mongo no maneja esquemas, no es necesario definir algo previamente. Además, la base de datos *empresa* no existe realmente hasta que se agreguen valores en ella, mediante la colección llamada `empleados` con la función `insertOne()`

```
>db.empleados.insertOne({  
  nombre: "Juan",  
  apellidos: "López Pérez",  
  fecha_nacimiento: ISODate("2000-07-01"),  
  genero: "M",  
  domicilio : {  
    calle : "Juan Escutia #26",  
    colonia : "Independencia",  
    ciudad: "Cuernavaca",  
    CP: 34672  
  },  
  idiomas: [ "español", "inglés", "francés" ]  
})  
{ acknowledged: true,  
  insertedId: ObjectId("634df5f3c0c93b307a64da70") }
```

# Inserción de documentos

- La función `insertOne()` recibe como parámetro un documento JSON con la estructura de datos definida por el usuario. Debe de estar conforme a la definición de un documento JSON para que pueda ser insertado, aunque no es necesario (a menos que se defina un validador) que el contenido sea consistente en cada documento insertado.
- La función `insertMany()` permite la inserción de varios documentos en una sola operación. En este caso se debe vigilar que el parámetro sea un arreglo de documentos:

```
>db.empleados.insertMany([{"nombre": "Ana", "apellidos": "Pérez Gallegos", "fecha_nacimiento": ISODate("2001-09-20"), "genero": "F", "domicilio": {"calle": "Norte 122 #98-C", "colonia": "Agrícola", "ciudad": "Juchitán", "CP": 25673}, "idiomas": ["español"]}, {"nombre": "José Juan", "apellidos": "Monroy Escobedo", "fecha_nacimiento": ISODate("1998-10-10"), "genero": "M", "domicilio": {"calle": "Zaragoza #101", "colonia": "Nueva Industrial", "ciudad": "Coatzacoalcos", "CP": 45326}, "idiomas": ["español", "italiano"]}, {"nombre": "Paula", "apellidos": "Martínez López", "fecha_nacimiento": ISODate("2000-11-21"), "genero": "F", "domicilio": {"calle": "República Dominicana #268", "colonia": "Mundial", "ciudad": "Torreón", "CP": 34672}, "idiomas": [null]}])  
{ "acknowledged": true,  
  "insertedIds":  
    { '0': ObjectId("634e13cac0c93b307a64da72"),  
     '1': ObjectId("634e13cac0c93b307a64da73"),  
     '2': ObjectId("634e13cac0c93b307a64da74") } }
```

# Consultas

- Para listar el contenido de una colección, se usa la función **find()**.

```
> db.empleados.find()
{ _id: ObjectId("634e1321c0c93b307a64da71"),
  nombre: 'Ana', [REDACTED],
  apellidos: 'Pérez Gallegos',
  fecha nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F', [REDACTED],
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola', [REDACTED],
      ciudad: 'Juchitán',
      CP: 25673 },
    idiomas: [ 'español' ] }
{ id: ObjectId("634e13cac0c93b307a64da72"),
  nombre: 'Ana', [REDACTED],
  apellidos: 'Pérez Gallegos',
  fecha nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F', [REDACTED],
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola', [REDACTED],
      ciudad: 'Juchitán',
      CP: 25673 },
    idiomas: [ 'español' ] }
{ id: ObjectId("634e13cac0c93b307a64da73"),
  nombre: 'José Juan', [REDACTED],
  apellidos: 'Monroy Escobedo',
  fecha nacimiento: 1998-10-10T00:00:00.000Z,
  genero: 'M', [REDACTED],
  domicilio:
    { calle: 'Zaragoza #101',
      colonia: 'Nueva Industrial',
      ciudad: 'Coatzacoalcos',
      CP: 45326 },
    idiomas: [ 'español', 'italiano' ] }
{ id: ObjectId("634e13cac0c93b307a64da74"),
  nombre: 'Paula', [REDACTED],
  apellidos: 'Martínez López',
  fecha nacimiento: 2000-11-21T00:00:00.000Z,
  genero: 'F', [REDACTED],
  domicilio:
    { calle: 'República Dominicana #268',
      colonia: 'Mundial',
      ciudad: 'Torreón',
      CP: 34672 },
    idiomas: [ null ] }
```

# Creación de colecciones

- El comando `show collections` muestra las colecciones creadas hasta el momento en el sistema.
- > `show collections`  
`empleados`
- También es posible crear una colección con el comando `createCollection()` sobre la base de datos en uso.
- > `use test`  
`'switched to db test'`
- > `db.createCollection("prueba")`  
`{ ok: 1 }`
- > `show collections`  
`prueba`
- Es preferible usar `createCollections()` cuando se desea establecer los parámetros de configuración de la base de datos, aunque por lo regular se crea automáticamente cuando se inserta el primer documento.

```
> db.createCollection("demo",
{capped:true, autoIndexId:true,
size:6142800, max:10000})
{ ok: 1 }
> show collections
demo
prueba
```

Campo	Tipo	Descripción
<code>capped</code>	Boolean	(Opcional) Si es <code>true</code> , habilita una colección con límites. La colección de tamaño fijo sobrescribe automáticamente sus entradas más antiguas cuando alcanza su tamaño máximo. Si especifica <code>true</code> , también debe especificar el parámetro de tamaño.
<code>autoIndexId</code>	Boolean	(Opcional) Si es <code>true</code> , crea automáticamente un índice en el campo <code>_id</code> . El valor predeterminado es <code>false</code> .
<code>size</code>	number	(Opcional) Especifica un tamaño máximo en bytes para una colección con límites. Si <code>capped</code> es <code>true</code> , también debe especificar este campo.
<code>max</code>	number	(Opcional) Especifica el número máximo de documentos permitidos en la colección con límite ( <code>capped</code> ).
<code>validator</code>	document	(Opcional) Realiza una validación conforme al documento JSON Schema pasado como parámetro.

# JSON Schema

- JSON Schema es una especificación para el formato JSON para definir su estructura.
- Fue escrito bajo el borrador del IETF que expiró en 2011. Esquema JSON:
  - Describe el formato de datos existente.
  - Documentación clara, legible por humanos y máquinas.
  - Validación estructural completa, útil para pruebas automatizadas.
  - Validación estructural completa, validando los datos enviados por el cliente.

[JSON Schema Validation: A Vocabulary for Structural Validation of JSON \(json-schema.org\)](https://json-schema.org)

- MongoDB puede utilizar una versión adaptada de JSON Schema para validación de documentos (a partir de la versión 4.x)

# Estructura de un objeto JSON Schema

```
db.createCollection("colección", {  
    validator: {  
        $jsonSchema: {  
            bsonType: "object",  
            title: "título del esquema",  
            required: [ 'att1', 'att2', 'att3', ... , 'attn' ],  
            properties: {  
                att1: {  
                    bsonType: 'tipo de dato',  
                    minimum: entero,  
                    maximum: entero,  
                    minLength: entero,  
                    pattern : "expresión regular",  
                    enum: [ "dato1", "dato2", ..., "datoN" ],  
                    description: "descripción del atributo"  
                }  
            }  
        }  
    }  
})
```

# Elementos de JSON Schema

Elemento	Tipo	Definición	Comportamiento
title	N/A	string	Se usa para dar un título al esquema. No tiene efecto en la validación
description	N/A	string	Se usa para dar una breve descripción del esquema. No tiene efecto en la validación
type	Todos los tipos (no duplicados)	String o arreglos de string (no duplicados)	Indica el tipo de dato del valor, se permiten ("null", "boolean", "object", "array", "number", "string"). Los tipos "integer" son soportados mediante el tipo bsonType y los tipos "int" o "long".
properties	objeto	objeto	Objeto que define los atributos, sus valores mínimos y máximos, y demás restricciones.
required	objeto	Arreglo de string (no duplicados)	Es un arreglo de atributos que serán obligatorios.
minimum	número	número	Restricción que representa el valor mínimo aceptable para un atributo.
exclusiveMinimum	número	booleano	Tiene un valor booleano, El valor de un atributo es válido si es estrictamente mayor que el valor establecido en <b>minimum</b> .
maximum	número	número	Restricción que representa el valor máximo aceptable para un atributo.
exclusiveMaximum	número	booleano	Tiene un valor booleano, El valor de un atributo es válido si es estrictamente mayor que el valor establecido en <b>maximum</b> .
enum	Todos los tipos	Arreglo de valores	Un arreglo, de al menos un elemento, de posibles valores únicos para un atributo
maxLength	string	integer	Longitud máxima del valor de un atributo de tipo texto.
minLength	string	integer	Longitud mínima del valor de un atributo de tipo texto.
pattern	String	String como una expresión regular	Un valor de un atributo se considera válido si la expresión regular evalúa en verdadero.
bsonType	Todos los tipos	String alias o arreglo de string alias	Acepta string aliases

# Uso de JSON Schema en MongoDB

- Es posible emplear la validación JSON en MongoDB, ya sea al momento de crear una colección, o si ya existe, aplicar la validación a los documentos existentes y a los nuevos.
- Para agregar un validador al momento de crear una colección, se pasa como parámetro a la función *createCollection* un objeto que cumpla con JSON Schema:

```
db.createCollection("colección", {  
    validator: { . . .  
})
```

- Para saber si una colección tiene un validador JSON, se emplea la siguiente instrucción:

```
db.getCollectionInfos({name: "colección"})[0].options.validator
```

# Validación en línea

- Es posible hacer una validación "al vuelo", asignando un objeto JSON Schema a una variable JavaScript, y posteriormente revisar los documentos que cumplen (o no) con el esquema. Suponga la definición de un objeto de tipo esquema en la variable **miesquema**:

```
var miesquema={  
    $jsonSchema:{  
        required:["item","cantidad","enalmacen"],  
        properties:{  
            item:{bsonType:"string"},  
            cantidad:{bsonType:"int"},  
            enalmacen:{bsonType:"bool"},  
            tamanio:{  
                bsonType:"object",  
                required:["unidadmedida"],  
                properties:{  
                    unidadmedida:{bsonType:"string"},  
                    altura:{bsonType:"double"},  
                    anchura:{bsonType:"double"}  
                }  
            }  
        }  
    }  
}  
  
db.prueba.insertOne({item:"zapatos", cantidad:2, enalmacen:true,  
tamanio:{unidadmedida:"caja", altura:25.6, anchura:37.9}});  
db.prueba.insertOne({item:"calcetas", cantidad:"uno",  
enalmacen:false, tamanio:{unidadmedida:"par", altura:23.0,  
anchura:6.0}});
```

- Para verificar que los documentos en la colección cumplan con el esquema, se usa el comando **find** sobre el esquema:

```
db.coleccion.find(miesquema)
```

- Si se quieren obtener aquellos documentos que no cumplen con el esquema, se emplea:

```
db.coleccion.find({$nor:[miesquema] })
```

Y para eliminarlos:

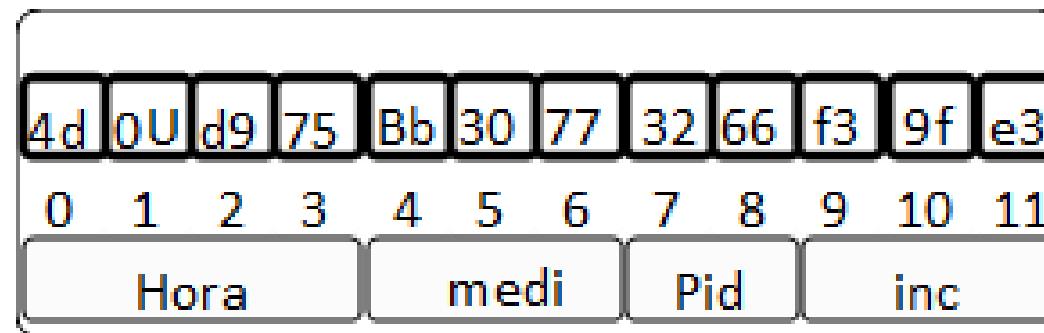
```
db.coleccion.deleteMany({$nor:[miesquema] })
```

- Como forma de control de los registros que no cumplen con el esquema, se puede agregar un atributo de tipo "bandera", indicando que cumple o no cumple con el esquema:

```
db.coleccion.updateMany(  
    {  
        $nor:[miesquema]  
    },  
    {  
        $set:{esValido:false}  
    }  
)
```

# Consultas

- La salida JSON contiene un campo `_id` de tipo `ObjectId`. Es similar al tipo SERIAL de una base de datos relacional.
  - El `ObjectId` es siempre de 12 bytes, compuesto por una estampa de tiempo, un ID de equipo cliente, un ID de proceso de cliente y un contador incrementado secuencial de 3 bytes.
- Cada proceso en cada máquina puede manejar su propia generación de ID sin colisionar con otras instancias `mongod`.



# JavaScript

- La lengua materna de Mongo es JavaScript.
- `db` es un objeto JavaScript que contiene información sobre la base de datos actual.
- `db.x` es un objeto JavaScript que representa una colección (denominada `x`).
- Los comandos son funciones de JavaScript.

```
> typeof db
'object'
> typeof db.empleados
'object'
> typeof db.empleados.insertOne
'function'
```

- Es posible crear una función propia que encapsule la inserción de datos en JavaScript:

```
>inserta_empleado = function(nomb,apell,fn,gen,dom,ids)
{db.empleados.insertOne({nombre:nomb,apellidos:apell,
fecha_nacimiento:ISODate(fn),genero:gen,domicilio:dom,idiomas:ids
}) }
```

```
[Function: inserta_empleado]
```

La función recién creada se puede invocar con los valores a insertar:

```
> inserta_empleado("María","López Martínez","1998-05-28",
{calle:"Buenaventura #10",colonia:"Benito |
Juárez",ciudad:"Torreón",CP:45329 }, ["español", "inglés",
"portugués"] )
```

- La función `find()` sin parámetros se usa para obtener todos los documentos, sin condiciones. Para acceder a un documento específico y único, se emplea la función `findOne()` y como parámetro, un objeto con la búsqueda de un criterio que garantice un solo documento de retorno. Regularmente se emplea el atributo `_id`. El atributo `_id` se debe convertir a una cadena de texto con la función `ObjectId(" _id")`. Si la expresión regresa un conjunto de documentos, solo se muestra el primero.

```
> db.empleados.findOne({_id: ObjectId("634e1321c0c93b307a64da71") })
```

```
{_id: ObjectId("634e1321c0c93b307a64da71"),
  nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola',
      ciudad: 'Juchitán',
      CP: 25673 },
  idiomas: [ 'español' ]}
```

- El valor del atributo `_id` es asignado por el sistema, pero también es posible asignarlo por parte del usuario, como una cadena de texto.

```
> db.empleados.insertOne({  
  _id: "empleado10",  
  nombre: "Guadalupe",  
  apellidos: "García Barrera",  
  fecha_nacimiento: ISODate("1999-01-18"),  
  genero: "F",  
  domicilio: {  
    calle: "calle #129",  
    colonia: "Progreso",  
    ciudad: "Cuernavaca",  
    CP: 34672  
  }  
})
```

# Búsqueda de un solo registro

---

- De igual manera, la búsqueda de un solo documento puede hacerse con la función `findOne()` y el atributo `_id` con el valor de texto asignado

```
db.empleados.findOne({_id:"empleado10"})
{_id: 'empleado10',
 nombre: 'Guadalupe',
 apellidos: 'García Barrera',
 fecha_nacimiento: 1999-01-18T00:00:00.000Z,
 genero: 'F',
 domicilio:
 { calle: 'calle #129',
   colonia: 'Progreso',
   ciudad: 'Cuernavaca',
   CP: 34672 } }
```

- Las funciones `find()` y `findOne()` también aceptan un segundo parámetro opcional: un objeto `fields` para filtrar los atributos a recuperar.
  - Para obtener solo el nombre (junto con el `_id`), hay pasar como objeto el atributo deseado con el valor a `1` (o `true`).

```
> db.empleados.find({_id: ObjectId("634e1321c0c93b307a64da71") , {nombre:1}})  
  
{_id: ObjectId("634e1321c0c93b307a64da71"),  
 nombre: 'Ana'}
```

- Para recuperar todos los atributos, excepto `nombre`, hay que establecerlo a `0` (o `false`).

```
> db.empleados.find({_id: ObjectId("634e1321c0c93b307a64da71") , {nombre:0}})  
  
{_id: ObjectId("634e1321c0c93b307a64da71"),  
 apellidos: 'Pérez Gallegos',  
 fecha_nacimiento: 2001-09-20T00:00:00.000Z,  
 genero: 'F',  
 domicilio:  
   { calle: 'Norte 122 #98-C',  
     colonia: 'Agrícola',  
     ciudad: 'Juchitán',  
     CP: 25673 },  
     idiomas: [ 'español' ]}
```

- En Mongo, se pueden construir consultas sobre los atributos, rangos, o una combinación de criterios.
- Para encontrar todos los empleados cuyo nombre comience con la letra *P* (mayúscula) y sean de género femenino (carácter ‘F’), se puede usar la siguiente expresión:

```
db.empleados.find({nombre:/^P/, genero: 'F'}, {_id:0,nombre:1,apellidos:1})  
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

- La expresión `/^P/` es una expresión regular. En la siguiente diapositiva se muestran algunos ejemplos de construcción y uso de las expresiones regulares.

# Notación de Regex

A continuación, se muestran los símbolos empleados en las expresiones regulares, para escribir consultas de búsqueda de patrones:

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
$^abc$	abc at the <i>beginning</i> of the string
abc\$	abc at the <i>end</i> of the string
a b	either of a and b
$^abc abc\$$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does <i>not</i> contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\B	perl when <i>not</i> followed by a word boundary (e.g. in perlert but not in perl stuff)

- Los operadores condicionales unarios en Mongo siguen el formato de `{atributo : { $operador : valor}}`, donde `$op` es un operador de comparación, como `$ne` (no igual a) o `$gt` (mayor que). En la siguiente diapositiva se muestran algunos de los operadores empleados en JavaScript.
- Los operadores binarios siguen el formato `{ $operador :[ {atributo : valor}, {atributo : valor}] }` en donde el operador binario tiene como valor un arreglo de dos objetos como condiciones. Es necesario que sean dos condiciones para que no sea un error.
- Es posible construir el operador de comparación como un objeto de JavaScript, y posteriormente invocarlo dentro de la especificación de la consulta, como en los siguientes criterios:

```
var condicion = {$eq: 'F'}
db.empleados.find({nombre:/^P/, genero:condicion}, {_id:0,nombre:1,apellidos:1})
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# Operadores de comparación

OPERADOR	DESCRIPCIÓN
<code>\$regex</code>	Coincidencia por cualquier cadena de expresión regular compatible con PCRE (o usar los delimitadores //)
<code>\$ne</code>	No es igual a
<code>\$lt</code>	Menor que
<code>\$lte</code>	Menor o igual que
<code>\$gt</code>	Mayor que
<code>\$gte</code>	Mayor o igual que
<code>\$exists</code>	Comprobar la existencia de un atributo
<code>\$all</code>	Hacer coincidir todos los elementos de un arreglo
<code>\$in</code>	Hacer coincidir cualquier elemento de un arreglo
<code>\$nin</code>	No coincide con ningún elemento de un arreglo
<code>\$elemMatch</code>	Coincidencia de todos los atributos en un arreglo de documentos anidados
<code>\$or</code>	o
<code>\$nor</code>	Negación de o
<code>\$size</code>	Coincidencia de arreglo de tamaño dado
<code>\$mod</code>	Operador módulo
<code>\$type</code>	Coincidencia si el atributo es de un tipo de dato dado
<code>\$not</code>	Negación de la comparación

Operación	Sintaxis	Ejemplo	Equivalente
Igualdad	{<key>:<value>}	db.mycol.find({ "nombre": "Ana" })	where nombre = 'Ana'
Menos que	{<key>:{ \$lt:<value>} }	db.mycol.find({ "edad": { \$lt:30 } })	where edad < 30
Menos que o igual	{<key>:{ \$lte:<value>} }	db.mycol.find({ "edad": { \$lte:30 } })	where edad <= 30
Mayor que	{<key>:{ \$gt:<value>} }	db.mycol.find({ "edad": { \$gt:30 } })	where edad > 30
Mayor que o igual	{<key>:{ \$gte:<value>} }	db.mycol.find({ "edad": { \$gte:30 } })	where edad >= 30
No es igual	{<key>:{ \$ne:<value>} }	db.mycol.find({ "edad": { \$ne:30 } })	where edad != 30

Comparativa de operadores RDBMS con MongoDB

- Para consultas con rangos de datos, se utilizan los operadores de comparación, empleando la función *ISODate()*, la cuál hace la conversión de una cadena de texto, con el formato *YYYY-MM-DD*, a un objeto de tipo *Date*, como en el siguiente ejemplo, que obtiene los nombres de los empleados que nacieron posterior al 1 de enero de 2000:

```
db.empleados.find({fecha nacimiento:{$gt:ISODate('2000-01-01')}},{_id:0,nombre:1,apellidos:1})  
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }  
{ nombre: 'Juan', apellidos: 'López Pérez' }  
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# Operadores de comparación binarios

- Para las expresiones de consulta que usen un operador binario, como los operadores *\$or* y *\$and*, es necesario incluir como valor del operador un arreglo de al menos dos elementos, los cuales serán objetos que tienen las condiciones a aplicar:

```
db.empleados.find({$or:[{fecha_nacimiento:{$gt:ISODate('2000-01-01')}}, {genero:'F'}]}, {_id:0,nombre:1,apellidos:1})  
{ nombre: 'Paula', apellidos: 'Martínez López' }  
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }  
{ nombre: 'Juan', apellidos: 'López Pérez' }
```

- En el método **find()**, si se pasan varias claves separándolas por ',' (coma), entonces MongoDB lo trata como una condición *\$and*.

```
db.empleados.find({$and:[{fecha_nacimiento:{$gt:ISODate('2000-01-01')}}, {genero:'F'}]}, {_id:0,nombre:1,apellidos:1})  
{ nombre: 'Paula', apellidos: 'Martínez López' }  
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
```

- Los operadores `$in`, `$nin` y `$all` pueden obtener los valores coincidentes dentro de un arreglo para una expresión condicional. El siguiente ejemplo regresa los empleados que hablan los idiomas *español* o *inglés*:

```
db.empleados.find({idiomas:{$in:['ingles','español']}},{_id:0,nombre:1,apellidos:1,idiomas:1})  
{ nombre: 'Ana', [REDACTED]  
  apellidos: 'Pérez Gallegos', [REDACTED]  
  idiomas: [ 'español' ] } [REDACTED]  
{ nombre: 'José Juan', [REDACTED]  
  apellidos: 'Monroy Escobedo', [REDACTED]  
  idiomas: [ 'español', 'italiano' ] } [REDACTED]  
{ nombre: 'Juan', [REDACTED]  
  apellidos: 'López Pérez', [REDACTED]  
  idiomas: [ 'español', 'inglés', 'francés' ] }
```

- La búsqueda de atributos que estén contenidos dentro de otros objetos (objetos anidados) se realiza mediante el operador `.` (punto). Debe tener cuidado de poner los atributos entrecomillados para que no se genere un error. En el ejemplo se obtiene el *nombre* y *apellidos* del empleado que radica en la ciudad de *Cuernavaca*:

```
db.empleados.find({ 'domicilio.ciudad': 'Cuernavaca' }, { _id:0, nombre:1, apellidos:1 })
{ nombre: 'Juan', apellidos: 'López Pérez' }
```

- Si se desea saber si existe un atributo en el documento, se usa el operador `$exists`. Por ejemplo, para saber si algún empleado tiene datos de los idiomas que habla:

```
db.empleados.find({ idiomas:{$exists:true} }, { _id:0, nombre:1, apellidos:1, idiomas:1 })
{ nombre: 'Juan', apellidos: 'López Pérez', idiomas: [ 'español', 'inglés', 'francés' ] }
{ nombre: 'Paula', apellidos: 'Martínez López', idiomas: [ null ] }
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo', idiomas: [ 'español', 'italiano' ] }
{ nombre: 'Ana', apellidos: 'Pérez Gallegos', idiomas: [ 'español' ] }
```

- Se observa en el resultado que un empleado tiene el valor `null` asignado al atributo *idiomas*, que es diferente que si no existiera ese atributo.

# Salida formateada

- La función `pretty()` se usa para presentar los resultados de la función `find()` con una lectura más cómoda, y se escribe al final de la invocación de `find`. Los resultados son los mismos.

```
db.empleados.find().pretty()
{ _id: ObjectId("634e1321c0c93b307a64da71"),
  nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola',
      ciudad: 'Juchitán',
      CP: 25673 },
  idiomas: [ 'español' ] }
{ _id: ObjectId("634e13cac0c93b307a64da72"),
  nombre: 'Ana',
  apellidos: 'Pérez Gallegos',
  fecha_nacimiento: 2001-09-20T00:00:00.000Z,
  genero: 'F',
  domicilio:
    { calle: 'Norte 122 #98-C',
      colonia: 'Agrícola',
      ciudad: 'Juchitán',
      CP: 25673 },
  idiomas: [ 'español' ] }
```

# Limitar registros

- Para mostrar solo los primeros documentos, se debe utilizar la función **limit()**. Acepta un argumento numérico, que es el número de documentos que desea mostrar.

```
db.empleados.find({ }, { _id:0, nombre:1, apellidos:1 }).limit(3)
{ nombre: 'Juan', apellidos: 'López Pérez' }
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo' }
```

# Saltar documentos

- La función `skip()` se utiliza para omitir (saltar) los documentos indicados por el parámetro numérico, a partir del primer resultado. Se emplea junto con `limit()`

```
db.empleados.find({},{_id:0,nombre:1,apellidos:1}).limit(3).skip(1)
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo' }
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# Ordenamiento de registros

- La función **sort()** realiza un ordenamiento del resultado obtenido por la función **find()**. Acepta un documento que contiene una lista de atributos junto con su orden de clasificación.
- Para especificar el orden de clasificación se utilizan los valores enteros 1 (ascendente) y -1 (descendente).

```
db.empleados.find({}, {nombre:1, apellidos:1, _id:0}).sort({nombre:  
1, apellidos:-1})  
{ nombre: 'Ana', apellidos: 'Pérez Gallegos' }  
{ nombre: 'Guadalupe', apellidos: 'García Barrera' }  
{ nombre: 'José Juan', apellidos: 'Monroy Escobedo' }  
{ nombre: 'Juan', apellidos: 'López Pérez' }  
{ nombre: 'María', apellidos: 'López Martínez' }  
{ nombre: 'Paula', apellidos: 'Martínez López' }
```

# countDocuments() y distinct()

- Para obtener el total (conteo) de los documentos en una colección, se emplea la función `countDocuments()` sin parámetros :

```
db.empleados.countDocuments()  
4
```

- Se puede pasar como parámetro una condición que será evaluada y regresará el conteo de documentos que cumplieron la condición:

```
db.empleados.countDocuments({genero:'F'})  
4
```

- El método `distinct()` devuelve cada valor diferente contenido dentro de un atributo. Se tiene que pasar como primer parámetro obligatorio, el nombre del atributo, como texto, y opcionalmente como segundo parámetro un objeto como condición:

```
db.empleados.distinct('genero')  
[ 'F', 'M' ]  
db.empleados.distinct('nombre', {genero:'F'})  
[ 'Ana', 'Martha', 'Paula' ]
```

# Agregación

- Las operaciones de agregación procesan registros de datos y devuelven resultados calculados.
- Las operaciones de agregación agrupan los valores de varios documentos juntos y pueden realizar una variedad de operaciones en los datos agrupados para devolver un único resultado.
- Para la agregación en MongoDB, debe utilizar el método **aggregate ()**
  - Permite especificar una lógica en estilo tubería que consta de etapas tales como:
    - **\$match** - filtros que devuelven conjuntos específicos de documentos;
    - **\$group** - funciones que agrupan con base de algún atributo;
    - **\$sort()** - ordena los documentos por una clave de ordenación; y otros más.

# Ejemplo

- Suponer un documento con la siguiente composición:

```
{"nombre" : "Roberto", "apellido" : "Canales", "peso" : 80, "sede" : "Madrid", "puesto" : "Desarrollo",  
"gastos" : [{"fecha" : "Enero", "Comercio" : "ElCorteDeManga",  
    "Compra" : [{"Concepto" : "Electronica", "Importe" : 1000},  
        {"Concepto" : "Alimentación", "Importe" : 2000},  
        {"Concepto" : "Deportes", "Importe" : 2000}]],  
    {"fecha" : "Febrero", "Comercio" : "Garranfur",  
    "Compra" : [{"Concepto" : "Electronica", "Importe" : 400},  
        {"Concepto" : "Alimentación", "Importe" : 1000},  
        {"Concepto" : "Deportes", "Importe" : 2000}]}  
}
```

- Insertados en la base de datos:

```
db.compras.insertOne({"nombre" : "Roberto", "apellido" : "Canales", "peso" : 80, "sede" : "Madrid", "puesto" : "Desarrollo", "gastos" : [{"fecha" : "Enero", "Comercio" : "ElCorteDeManga", "Compra" : [{"Concepto" : "Electronica", "Importe" : 1000}, {"Concepto" : "Alimentación", "Importe" : 2000}, {"Concepto" : "Deportes", "Importe" : 2000}], {"fecha" : "Febrero", "Comercio" : "Garranfur", "Compra" : [{"Concepto" : "Electronica", "Importe" : 400}, {"Concepto" : "Alimentación", "Importe" : 1000}, {"Concepto" : "Deportes", "Importe" : 2000}]}]})  
{ acknowledged: true, insertedId: ObjectId("63e3dfc2c51c76656d27b66b") }
```

```
db.compras.insertOne({"nombre" : "Jose Maria", "apellido" : "Toribio", "peso" : 78, "sede" : "Madrid", "puesto" : "Operaciones", "gastos" : [{"fecha" : "Enero", "Comercio" : "EFVNAF", "Compra" : [{"Concepto" : "Electronica", "Importe" : 1000}, {"Concepto" : "Deportes", "Importe" : 2000}], {"fecha" : "Febrero", "Comercio" : "Garranfur", "Compra" : [{"Concepto" : "Electronica", "Importe" : 400}, {"Concepto" : "Alimentación", "Importe" : 1000}]}])  
{ acknowledged: true, insertedId: ObjectId("63e3e106c51c76656d27b66c") }
```

# Filtrando resultados con \$match

- El comando **\$match** permite obtener aquellos elementos que coincidan con el criterio de búsqueda:

```
db.compras.aggregate({$match:{sede:"Madrid"}})
{ _id: ObjectId("63e3dfc2c51c76656d27b66b"),
  nombre: 'Roberto',
  apellido: 'Canales',
  peso: 80,
  sede: 'Madrid',
  puesto: 'Desarrollo',
  gastos:
    [ { fecha: 'Enero',
        Comercio: 'ElCorteDeManga',
        Compra:
          [ { Concepto: 'Electronica', Importe: 1000 },
            { Concepto: 'Alimentación', Importe: 2000 },
            { Concepto: 'Deportes', Importe: 2000 } ] },
      { fecha: 'Febrero',
        Comercio: 'Garranfur',
        Compra:
          [ { Concepto: 'Electronica', Importe: 400 },
            { Concepto: 'Alimentación', Importe: 1000 },
            { Concepto: 'Deportes', Importe: 2000 } ] ] ],
  _id: ObjectId("63e3e106c51c76656d27b66c"),
  nombre: 'Jose Maria',
  apellido: 'Toribio',
  peso: 78,
  sede: 'Madrid',
  puesto: 'Operaciones',
  gastos:
    [ { fecha: 'Enero',
        Comercio: 'EFVNAF',
        Compra:
          [ { Concepto: 'Electronica', Importe: 1000 },
            { Concepto: 'Deportes', Importe: 2000 } ] },
      { fecha: 'Febrero',
        Comercio: 'Garranfur',
        Compra:
          [ { Concepto: 'Electronica', Importe: 400 },
            { Concepto: 'Alimentación', Importe: 1000 } ] ] ] }
```

- Se puede observar que regresa todos los documentos, ya que cumplen con el criterio de filtrado.

# Obtener elementos de un arreglo

- Como en el ejemplo se tienen elementos contenido dentro de un arreglo, para que se pueda aplicar el agrupamiento es necesarios obtenerlos como documentos, empleando el comando `$unwind`:

```
db.compras.aggregate({$match:{sede:"Madrid"}},{$unwind:"$gastos"},{$unwind:"$gastos.Compra"})  
{ _id: ObjectId("63e3e106c51c76656d27b66c"),  
  nombre: 'Jose Maria',  
  apellido: 'Toribio',  
  peso: 78,  
  sede: 'Madrid',  
  puesto: 'Operaciones',  
  gastos:  
    { fecha: 'Enero',  
      Comercio: 'ElCorteDeManga',  
      Compra: { Concepto: 'Electronica', Importe: 1000 } } }  
{ _id: ObjectId("63e3dffc2c51c76656d27b66b"),  
  nombre: 'Roberto',  
  apellido: 'Canales',  
  peso: 80,  
  sede: 'Madrid',  
  puesto: 'Desarrollo',  
  gastos:  
    { fecha: 'Enero',  
      Comercio: 'ElCorteDeManga',  
      Compra: { Concepto: 'Alimentación', Importe: 2000 } } }  
{ _id: ObjectId("63e3dffc2c51c76656d27b66b"),  
  nombre: 'Roberto',  
  apellido: 'Canales',  
  peso: 80,  
  sede: 'Madrid',  
  puesto: 'Desarrollo',  
  gastos:  
    { fecha: 'Enero',  
      Comercio: 'ElCorteDeManga',  
      Compra: { Concepto: 'Deportes', Importe: 2000 } } }  
{ _id: ObjectId("63e3dffc2c51c76656d27b66b"),  
  nombre: 'Roberto',  
  apellido: 'Canales',  
  peso: 80,  
  sede: 'Madrid',  
  puesto: 'Desarrollo',  
  gastos:  
    { fecha: 'Febrero',  
      Comercio: 'Garranfur',  
      Compra: { Concepto: 'Electronica', Importe: 400 } } }  
{ _id: ObjectId("63e3e106c51c76656d27b66c"),  
  nombre: 'Jose Maria',  
  apellido: 'Toribio',  
  peso: 78,  
  sede: 'Madrid',  
  puesto: 'Operaciones',  
  gastos:  
    { fecha: 'Febrero',  
      Comercio: 'Garranfur',  
      Compra: { Concepto: 'Alimentación', Importe: 1000 } } }
```

# Agrupando resultados con \$group

- El comando **\$group** permite formar grupos a partir de algún atributo elegido para el agrupamiento. En el siguiente ejemplo se desean obtener los elementos agrupados con base en el atributo *fecha*, y calculando el subtotal de compras por cada grupo, con el operador **\$sum**:

```
db.compras.aggregate({$match:{sede:"Madrid"}},{$unwind:"$gastos"},{$unwind:"$gastos.Compra"},{$group:{_id:{fecha:"$gastos.fecha"},subtotal:{$sum:"$gastos.Compra.Importe"}}})  
{ _id: { fecha: 'Enero' }, subtotal: 8000 }  
{ _id: { fecha: 'Febrero' }, subtotal: 4800 }
```

- Se pueden cambiar los criterios de agrupamiento, así como agregar atributos de resultado, o cambiar el operador de agregación:

```
db.compras.aggregate({$match:{sede:"Madrid"}},{$unwind:"$gastos"},{$unwind:"$gastos.Compra"},{$group:{_id:{fecha:"$gastos.fecha",comercio:"$gastos.Comercio"},subtotal:{$sum:"$gastos.Compra.Importe"},cantidad:{$count:{}}}})  
{ _id: { fecha: 'Febrero', comercio: 'Garranfur' },  
  subtotal: 4800,  
  cantidad: 5 }  
{ _id: { fecha: 'Enero', comercio: 'ElCorteDeManga' },  
  subtotal: 5000,  
  cantidad: 3 }  
{ _id: { fecha: 'Enero', comercio: 'EFVNAF' },  
  subtotal: 3000,  
  cantidad: 2 }
```

Expresión	Descripción	Ejemplo
<b>\$sum</b>	Resume el valor definido de todos los documentos de la colección.	db.mycol.aggregate([{\$group:{_id:"\$key", total:{ <b>\$sum</b> :"\$att_num" }}}])
<b>\$avg</b>	Calcula el promedio de todos los valores dados de todos los documentos de la colección.	db.mycol.aggregate([{\$group:{_id:"\$key", prom:{ <b>\$avg</b> :"\$att_num" }}}])
<b>\$min</b>	Obtiene el mínimo de los valores correspondientes de todos los documentos de la colección.	db.mycol.aggregate([{\$group:{_id:"\$key", min:{ <b>\$min</b> :"\$att" }}}])
<b>\$max</b>	Obtiene el máximo de los valores correspondientes de todos los documentos de la colección.	db.mycol.aggregate([{\$group:{_id:"\$key", max:{ <b>\$max</b> :"\$att" }}}])
<b>\$push</b>	Inserta el valor en una matriz del documento resultante.	db.mycol.aggregate([{\$group:{_id:"\$key", array:{ <b>\$push</b> :"\$att" }}}])
<b>\$addToSet</b>	Inserta el valor en una matriz del documento resultante, pero no crea duplicados.	db.mycol.aggregate([{\$group:{_id:"\$key", array:{ <b>\$addToSet</b> :"\$att" }}}])
<b>\$first</b>	Obtiene el primer documento de los documentos de origen según la agrupación. Por lo general, esto solo tiene sentido junto con alguna etapa de "\$sort" aplicada anteriormente.	db.mycol.aggregate([{\$group:{_id:"\$key", first:{ <b>\$first</b> :"\$att" }}}])
<b>\$last</b>	Obtiene el último documento de los documentos de origen según la agrupación. Por lo general, esto solo tiene sentido junto con alguna etapa de "\$sort" aplicada anteriormente.	db.mycol.aggregate([{\$group:{_id:"\$key", last:{ <b>\$last</b> :"\$att" }}}])

# Modelado de relaciones

- Las relaciones en MongoDB representan cómo varios documentos se relacionan lógicamente entre sí. Las relaciones se pueden modelar a través de enfoques **integrados** y **referenciados**. Tales relaciones pueden ser 1:1, 1:N, N:1 o N:N.
- Consideremos el caso de almacenar varias direcciones para los usuarios. Por lo tanto, hace que esta sea una relación 1:N.

```
{  
  "_id": ObjectId("52ffc33cd85242f436000001"),  
  "nombre": "Tom Hanks",  
  "telefono": 987654321,  
  "fecha_nac": "1991-01-01"  
}  
  
{  
  "_id": ObjectId("52ffc4a5d85242602e000000"),  
  "domicilio": "22 A, Indiana Apt",  
  "CP": 123456,  
  "ciudad": "Los Angeles",  
  "estado": "California"  
}
```

# Modelado de relaciones incrustadas

- En el enfoque incrustado, incrustaremos el documento de dirección dentro del documento de usuario.
- Este enfoque mantiene todos los datos relacionados en un solo documento, lo que facilita su recuperación y mantenimiento. El documento completo se puede recuperar en una sola consulta como:
- El inconveniente es que si el documento incrustado sigue creciendo demasiado, puede afectar el rendimiento de lectura/escritura.

```
>db.personas.findOne({"nombre":"Tom Benzamin"}, {"direccion":1})
```

```
> db.personas.insert({  
  "_id":ObjectId("52ffc33cd85242f436000001"),  
  "telefono": "987654321",  
  "fecha_nac": "01-01-1991",  
  "nombre": "Tom Benzamin",  
  "direccion": [  
    { "domicilio": "22 A, Indiana Apt",  
      "CP": 123456,  
      "ciudad": "Los Angeles",  
      "estado": "California"  
    },  
    {  
      "domicilio": "170 A,  
      Acropolis Apt",  
      "CP": 456789,  
      "ciudad": "Chicago",  
      "estado": "Illinois"  
    }  
  ]  
})
```

# Modelado de relaciones referenciadas

- En este enfoque, tanto el documento de usuario como el de dirección se mantendrán por separado, pero el documento de usuario contendrá un campo que hará referencia al campo de identificación del documento de dirección.
- Como se muestra, el documento de usuario contiene el campo de arreglo *direcciones\_ids* que contiene los *ObjectIds* de las direcciones correspondientes. Usando estos *ObjectIds*, podemos consultar los documentos de dirección y obtener sus detalles.
- Con este enfoque, necesitaremos dos consultas: primero para obtener los campos *direcciones\_ids* del documento de usuario, y después para obtener estas direcciones del arreglo de direcciones .

```
{ "_id": ObjectId("52ffc33cd85242f436000001"),  
  "telefono": "987654321",  
  "fecha_nac": "01-01-1991",  
  "nombre": "Tom Benzamin",  
  "direcciones_ids": [  
    ObjectId("52ffc4a5d85242602e000000"),  
    ObjectId("52ffc4a5d85242602e000001")  
  ]  
}
```

```
>var res = db.personas.findOne({"nombre": "Tom  
Benzamin"}, {"direcciones_ids": 1})  
>var dirs =  
db.personas.find({"_id": {"$in": res["direcciones_ids"]}})
```

# DBRefs

- En los casos en que un documento contenga referencias de diferentes colecciones, podemos usar las ***DBRefs*** .
- Como escenario de ejemplo, donde usaríamos ***DBRefs*** en lugar de referencias manuales, considere una base de datos donde almacenamos diferentes tipos de direcciones (casa, oficina, correo, etc.) en diferentes colecciones (dirección\_casa, dirección\_oficina, dirección\_correo, etc.). Cuando el documento de una colección de usuarios hace referencia a una dirección, también debe especificar qué colección buscar en función del tipo de dirección. En tales escenarios donde un documento hace referencia a documentos de muchas colecciones, debemos usar ***DBRefs***.
- Hay tres campos en ***DBRefs***:
  - ***\$ref*** : este campo especifica la colección del documento al que se hace referencia
  - ***\$id*** : este campo especifica el campo *\_id* del documento al que se hace referencia
  - ***\$db*** : este es un campo opcional y contiene el nombre de la base de datos en la que se encuentra el documento al que se hace referencia.

# Definición y uso de DBRefs

- Considere un documento de usuario que tenga una dirección en un campo de tipo *DBRef*, como se muestra en el fragmento de código:
- El campo de dirección *DBRef* especifica que el documento de dirección al que se hace referencia se encuentra en la colección *direccion\_casa* en la base de datos *agenda* y tiene una identificación de *534009e4d852427820000002*.
- El siguiente código busca dinámicamente en la colección especificada por el parámetro *\$ref* un documento con la identificación especificada por el parámetro *\$id* en *DBRef*.

```
>var usr = db.personas.findOne({ "nombre": "Tom Benzamin" })  
>var dbRef = usr.direccion  
>db[dbRef.$ref].findOne({ "_id": (dbRef.$id) })  
  
{ "_id": ObjectId("53402597d852426020000002") ,  
  "direccion": {  
    "$ref": "direccion_casa",  
    "$id": ObjectId("534009e4d852427820000002") ,  
    "$db": "agenda" } ,  
  "telefono": "987654321" ,  
  "fecha_nac": "01-01-1991" ,  
  "nombre": "Tom Benzamin" }
```

# Consideraciones al diseñar en MongoDB

Diseñar el esquema de acuerdo con los requisitos del usuario.

Combinar objetos en un solo documento si se usarán juntos. De lo contrario, separarlos.

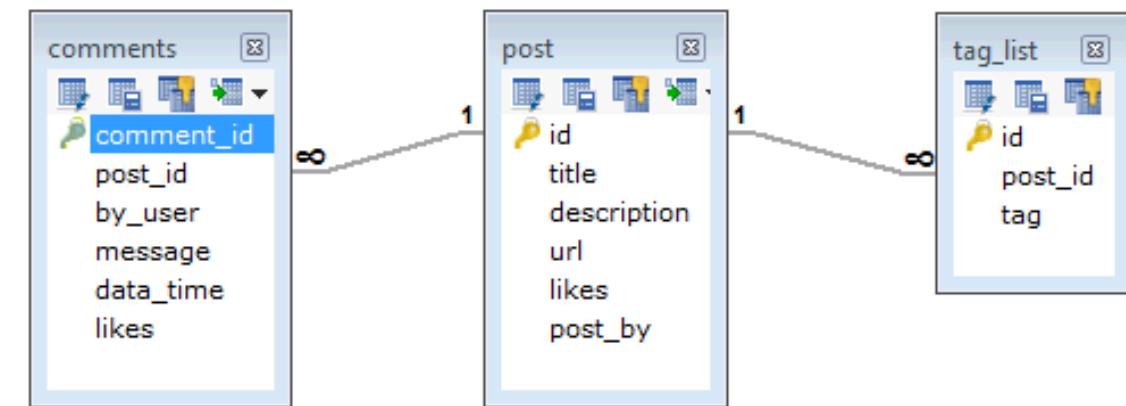
Duplicar datos (pero de forma limitada), porque el espacio en disco es barato en comparación con el tiempo de cómputo.

Optimizar el esquema para los casos de uso más frecuentes.

Realizar agregaciones complejas en el esquema.

# Ejemplo

- Suponer que un cliente necesita un diseño de base de datos para su blog/sitio web con los siguientes requisitos:
  - Cada publicación tiene un título, una descripción y una URL únicos.
  - Cada publicación puede tener una o más etiquetas.
  - Cada publicación tiene el nombre de su editor y el número total de “Me gusta”.
  - Cada publicación tiene comentarios proporcionados por los usuarios junto con su nombre, mensaje, tiempo de datos y “me gusta”.
  - En cada publicación, puede haber cero o más comentarios.
- En el esquema RDBMS, el diseño para los requisitos anteriores tendrá un mínimo de tres tablas:



- Mientras esté en el esquema MongoDB, el diseño tendrá una publicación de colección y la siguiente estructura:
- Entonces, mientras que, en un RDBMS, una consulta necesita unir tres tablas, en MongoDB, los datos se mostrarán de una sola una colección.

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user: 'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user: 'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

# Actualización de un solo documento

---

- La función `updateOne (condicion, operacion)` requiere dos parámetros.
  - El primero es una consulta basada en una condición para obtener el documento a actualizar. Se debe establecer una condición de búsqueda de un documento que garantice el retorno de solo uno.
  - El segundo es un valor u objeto que actualizará o reemplazará al atributo(s) indicado(s). Se usa el operador `$set` para establecer el conjunto de atributos con su correspondiente valor.

```
db.empleados.updateOne({ _id:ObjectId("634e29f1c0c93b307a64da77") }, { $set:{idiomas: ['español'] } })
{ acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0 }
db.empleados.findOne({ id:ObjectId("634e29f1c0c93b307a64da77") }, { idiomas:1 })
{ _id: ObjectId("634e29f1c0c93b307a64da77"),
  idiomas: [ 'español' ] }
```

# Actualización de varios documentos

---

- La función `updateMany (condicion, operacion)` funciona de forma similar a la función `updateOne ()`, con la diferencia de que el criterio de búsqueda puede regresar más de un documento.

```
db.empleados.updateMany({fecha_nacimiento:{$gt:ISODate("2000-01-01")}}, {$set:{tipo:"reciente"})  
{ acknowledged: true,  
  insertedId: null,  
  matchedCount: 3,  
  modifiedCount: 3,  
  upsertedCount: 0 }  
  
db.empleados.find({tipo:"reciente"}, {nombre:1, tipo:1})  
{ _id: ObjectId("634df5f3c0c93b307a64da70"),  
  nombre: 'Juan',  
  tipo: 'reciente' }  
{ _id: ObjectId("634e29f1c0c93b307a64da75"),  
  nombre: 'Ana',  
  tipo: 'reciente' }  
{ _id: ObjectId("634e29f1c0c93b307a64da77"),  
  nombre: 'Paula',  
  tipo: 'reciente' }
```

# Comando usados en actualización

Comando	Descripción
\$set	Establece el campo dado con el valor dado
\$unset	Quita el campo
\$inc	Incrementa el campo dado por el número dado
\$pop	Quita el último (o el primer) elemento de un arreglo
\$push	Agrega el valor a un arreglo
\$pushAll	Agrega todos los valores a un arreglo
\$addToSet	Similar a push, pero no duplica valores
\$pull	Quita los valores coincidentes de un arreglo
\$pullAll	Quita todos los valores coincidentes de un arreglo

# Eliminación de documentos

- De igual forma que en la actualización, la eliminación de documentos presenta dos funciones: `deleteOne(criterio)` y `deleteMany(criterio)`. Cuando el criterio es verdadero, se eliminará el documento completo.

```
>db.empleados.deleteOne({_id:ObjectId("634e29f1c0c93b307a64da75") })
```

- En caso de que se requiera la eliminación de varios documentos que cumplan el criterio, se debe usar la función `deleteMany(criterio)`.

```
>db.empleados.deleteMany({tipo:"reciente"}) )
```

# Operaciones atómicas

---

- En versiones recientes de MongoDB (4.0+), se agregaron las siguientes funciones:
  - `findOneAndUpdate (condicion, operacion)`
  - `findOneAndDelete (condicion, operacion)`
  - `findOneAndReplace (condicion, operacion)`
- las cuales combinan en una sola función las operaciones de modificación y eliminación de un solo documento, para hacer una equivalencia con respecto a una transacción.

```
db.collection.findAndModify ({  
    query: <document>,  
    sort: <document>,  
    remove: <boolean>,  
    update: <document or aggregation pipeline>, // Changed in MongoDB 4.2  
    new: <boolean>,  
    fields: <document>,  
    upsert: <boolean>,  
    bypassDocumentValidation: <boolean>,  
    writeConcern: <document>,  
    collation: <document>,  
    arrayFilters: [ <filterdocument1>, ... ],  
    let: <document> // Added in MongoDB 5.0  
});
```

Parámetro	Tipo	Descripción
<b>query</b>	documento	Opcional. Los criterios de selección para la modificación. El campo query emplea los mismos <a href="#">selectores de consulta</a> que se utilizan en el método <code>db.collection.find()</code> . Aunque la consulta puede coincidir con varios documentos, <code>db.collection.findAndModify()</code> sólo seleccionará un documento para modificar. Si no se especifica, el valor predeterminado es un documento vacío.
<b>sort</b>	documento	Opcional. Determina qué documento modifica la operación si la consulta selecciona varios documentos y modifica el primer documento en el orden de clasificación especificado por este argumento.
<b>remove</b>	booleano	Debe especificar el campo remove o update. Elimina el documento especificado en el campo query. Establézcalo en true para eliminar el documento seleccionado. El valor predeterminado es false.
<b>update</b>	documento o arreglo	Debe especificar el campo remove o update. Realiza una actualización del documento seleccionado.
<b>new</b>	booleano	Opcional. Cuando true, devuelve el documento modificado en lugar del original. El valor predeterminado es false.
<b>fields</b>	documento	Opcional. Un subconjunto de campos para devolver.
<b>upsert</b>	booleano	Opcional. Se utiliza junto con el campo update. Cuando es true: Crea un nuevo documento si ningún documento coincide con el query. Actualiza un único documento que coincide con el query. El valor predeterminado es false, que no inserta un nuevo documento cuando no se encuentra ninguna coincidencia.
<b>bypassDocumentValidation</b>	booleano	Opcional. Habilita para eludir la validación de documentos durante la operación. Esto le permite actualizar documentos que no cumplen con los requisitos de validación.
<b>writeConcern</b>	documento	Opcional. Un documento que expresa la <a href="#">inquietud de escritura</a> . Omita el uso de la preocupación de escritura predeterminada. No establezca explícitamente la preocupación de escritura para la operación si se ejecuta en una transacción.
<b>maxTimeMS</b>	entero	Opcional. Especifica un límite de tiempo en milisegundos para procesar la operación.
<b>collation</b>	documento	Opcional. Permite a los usuarios especificar reglas del idioma para la comparación de cadenas, como reglas para mayúsculas y minúsculas, y acentos.
<b>arrayFilters</b>	formación	Opcional. Una matriz de documentos de filtro que determina qué elementos de matriz modificar para una operación de actualización en un campo de matriz.
<b>let</b>	documento	Opcional. Especifica un documento con una lista de variables. Esto le permite mejorar la legibilidad de los comandos separando las variables del texto de la consulta.

# Operaciones atómicas

- Considere el escenario en el que se tenga la disponibilidad de los productos y la información sobre quién ha comprado productos, por separado.
  - Primero se comprueba si el producto está disponible para posteriormente actualizar la información de compra.
  - Sin embargo, es posible que algún otro usuario haya comprado el producto y ya no esté disponible. Sin saber esto, se actualizará la información de compra, haciendo que la base de datos sea inconsistente porque se ha vendido un producto que ya no está disponible.
- El enfoque recomendado para mantener la atomicidad es mantener toda la información relacionada, que con frecuencia se actualiza junta en un solo documento, utilizando documentos incrustados . Esto aseguraría que todas las actualizaciones de un solo documento sean atómicas.

```
>db.productos.findAndModify({  
    query: {_id:2, producto_disponible:{$gt:0}} ,  
    update: {  
        $inc: {producto_disponible:-1} ,  
        $push: {producto_comprado:{cliente:"Pedro", fecha:"2022-05-21"} }  
    }  
})
```

- El uso de la función *findAndModify* garantiza que la información de compra del producto se actualice solo si el producto está disponible. Y la totalidad de esta transacción es atómica.

# Eliminación de colecciones

---

- MongoDB utiliza la función `drop()` para eliminar una colección de la base de datos.
- Primero se recomienda revisar la colección existente y posteriormente eliminarla

```
>use badedatos
switched to db basedatos
>show collections
empleados
system.indexes
>db.empleados.drop()
true
```

# Eliminación de la base de datos

- La función `dropDatabase()` permite la eliminación completa de una base de datos. Se debe tener en uso la base de datos a eliminar, en caso contrario eliminará la base de datos `test`.

```
>use basedatos
switched to db basedatos
>db.dropDatabase()
>{ "dropped" : "basedatos", "ok" : 1 }
>show databases
local          0.78125GB
test           0.23012GB
```

# Indexado

- MongoDB debe escanear todos los documentos de una colección para seleccionar aquellos documentos que coincidan con la condición. Este escaneo es altamente ineficiente y requiere que procese un gran volumen de datos.
- Los índices son estructuras de datos especiales, que almacenan una pequeña parte del conjunto de datos en una forma fácil de recorrer. El índice almacena el valor de un atributo específico o conjunto de atributos, ordenado por el valor del atributo especificado en el índice.
- MongoDB proporciona varias estructuras de datos para el indexado, como los árboles B, así como otras adicionales, como los índices geoespaciales, bidimensionales y esféricos.

# Creación de índices

- Para crear un índice, debe utilizar la función `createIndex(atributos)` incluyendo como parámetro el conjunto de atributos a indexar.

```
>db.empleados.createIndex ({nombre:1,apellidos:-1})  
'nombre_1_apellidos_-1'
```

- Se emplea un valor entero 1 para el orden ascendente y -1 para el orden descendente.
- Cada vez que se crea una nueva colección, Mongo crea automáticamente un índice para el atributo `id`. Estos índices se pueden listar mediante la función `getIndexes()`:

```
>db.empleados.getIndexes()  
[  
  { v: 2, key: { _id: 1 }, name: '_id_' },  
  {  
    v: 2,  
    key: { nombre: 1, apellidos: -1 },  
    name: 'nombre_1_apellidos_-1'  
  }  
]
```



# Replicación

---

La replicación es el proceso de sincronización de datos entre varios servidores.

---

La replicación proporciona redundancia y aumenta la disponibilidad de los datos con varias copias de datos en diferentes servidores de bases de datos.

---

La replicación protege una base de datos de la pérdida de un único servidor.

---

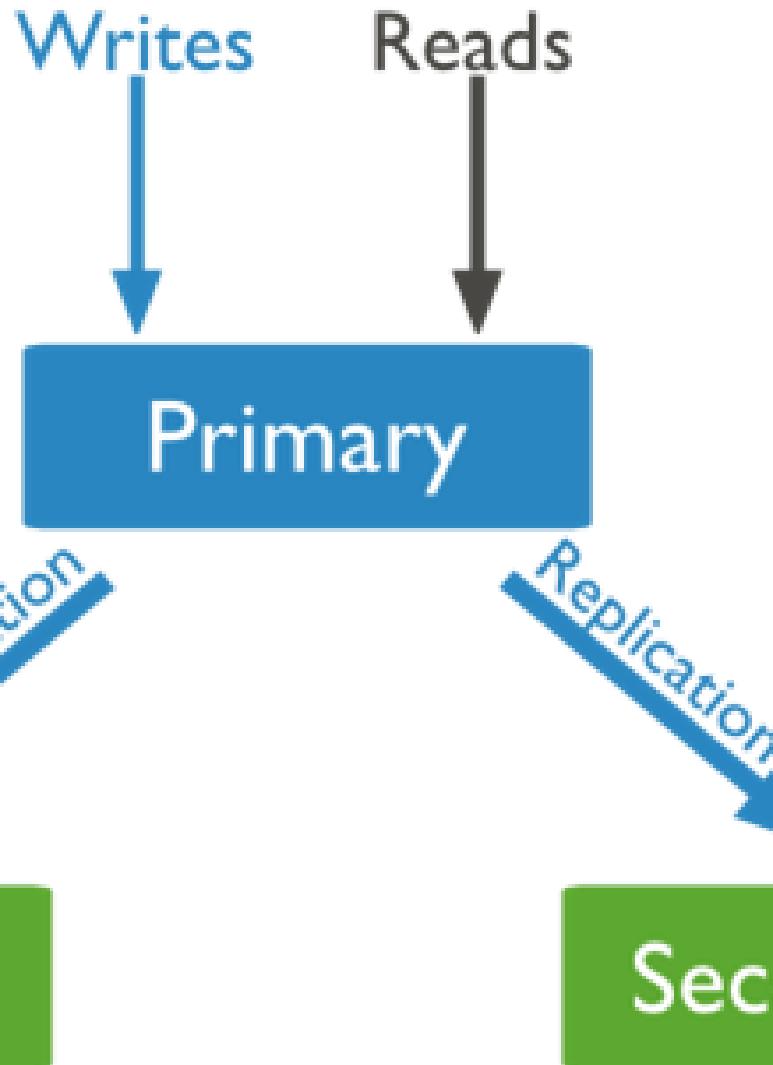
La replicación también le permite recuperarse de fallas de hardware e interrupciones del servicio.

---

Con copias adicionales de los datos, puede dedicar una a la recuperación ante desastres, la generación de informes o la copia de seguridad.

# Client Application

## Driver



# Conjuntos de réplicas

- Mongo fue diseñado para escalar horizontalmente, no para ejecutarse en modo independiente.
- Fue construido para la consistencia de los datos y la tolerancia de particionamiento, pero la fragmentación de datos tiene un costo: si se pierde una parte de una colección, todo se ve comprometido.
- Rara vez se debe ejecutar una sola instancia de Mongo en producción; en su lugar, se deben replicar los datos almacenados en varios servicios.
- Un conjunto de réplicas es un grupo de instancias **mongod** que alojan el mismo conjunto de datos.
- En una réplica, un nodo es el nodo principal que recibe todas las operaciones de escritura.
- Todas las demás instancias, como las secundarias, aplican operaciones desde el primario para que tengan el mismo conjunto de datos.
  - El conjunto de réplicas sólo puede tener un nodo principal.

## Consideraciones de la replicación

- El conjunto de réplicas es un grupo de dos o más nodos (generalmente se requieren un mínimo de 3 nodos).
- En un conjunto de réplicas, un nodo es el nodo primario y los restantes son secundarios.
- Todos los datos se replican desde el nodo primario al secundario.
- En el momento de la conmutación automática por error o mantenimiento, se elige un nuevo nodo primario.
- Despues de la recuperación del nodo fallido, se une nuevamente al conjunto de réplicas y funciona como un nodo secundario.

- Se simula un ambiente distribuido de varios servidores, ejecutando en terminales distintas.
  - El puerto predeterminado de Mongo es 27017, por lo que se inicia cada servidor en otros puertos.
  - Se recomienda crear un directorio individual para almacenamiento por cada uno de los servidores (p.e. c:\data\srv1, c:\data\srv2, etc.)
- Para levantar el servicio servidor en cada terminal se usa el siguiente comando:

```
mongod --port "PORT" --dbpath "DB_DATA_PATH" --replSet  
"REPLICA_SET_INSTANCE_NAME"
```

- En donde se tienen que establecer los valores de PORT (número de puerto de escucha), DB\_DATA\_PATH (ruta del directorio para los datos) y REPLICA\_SET\_INSTANCE\_NAME (nombre que identifica a la replicación)

- Para iniciar los servidores con replicación "empresa" (hay que asegurarse que los puertos indicados no estén asignados a otro proceso)

```
$ mongod --replSet empresa --dbpath c:\data\rep1 --port 27001  
$ mongod --replSet empresa --dbpath c:\data\rep2 --port 27002  
$ mongod --replSet empresa --dbpath c:\data\rep3 --port 27003
```

- A continuación hay que abrir una terminal para la ejecución del servidor primario, mediante el comando

```
$ mongo localhost:27001  
> rs.initiate({  
  id: "empresa",  
  members: [  
    {_id: 1, host: "localhost:27001"},  
    {_id: 2, host: "localhost:27002"},  
    {_id: 3, host: "localhost:27003"}  
  ]  
})  
> rs.status().ok
```

- Observe que en el arreglo **members** se establecen los servidores que van a participar en la replicación, con su identificador y su dirección IP con su puerto.
- El objeto **rs** se refiere al conjunto de replicas en uso (replica set)
- el comando **status()** nos permitirá saber cuándo se está ejecutando nuestro conjunto de réplicas

- Cada uno de las consolas mostrará un mensaje similar a

```
Member ... is now in state PRIMARY
```

```
Member ... is now in state SECONDARY
```

- dependiendo de cómo haya sido elegido de entre los nodos participantes (lo más probable es que sea el primero en la lista, con puerto 27001).
- Para comprobar la replicación, en la terminal del nodo designado como primario, se ejecuta una instrucción de mensaje:

```
> db.echo.insert({ say : 'HELLO!' })
```

y posteriormente se termina la consola (CRTL + C), y se observa que los mensajes de los otros nodos indican que alguno de ellos se promovió a primario.

- En esa consola, ejecutar

```
db.echo.find()()
```

- y se debe observar el valor enviado

- Para saber el estado y la configuración del nodo principal, se ejecutan:  
> db.status()  
> db.conf()
- Si se trata de insertar un valor en un nodo que no es el primario, se lanzará un error como el siguiente:

```
WriteResult({ "writeError" : { "code" : 10107, "errmsg" : "not master" } })
```

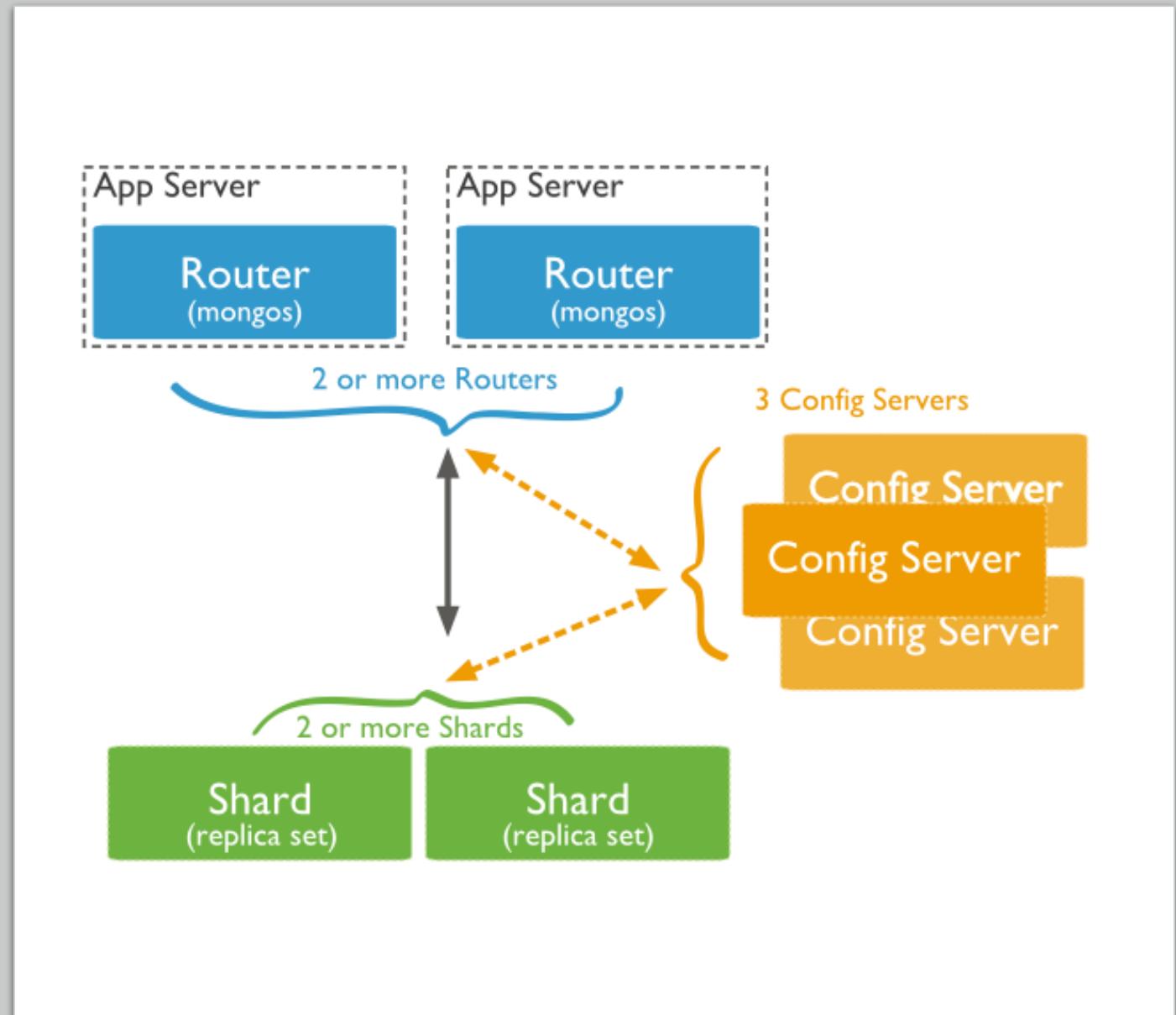
# Fragmentación

Uno de los objetivos principales de Mongo es proporcionar un manejo seguro y rápido de conjuntos de datos muy grandes.

El método más claro para lograr esto es a través de la fragmentación horizontal por rangos de valor.

- En lugar de un único servidor que aloja todos los valores de una colección, algún rango de valores se divide o *fragmenta* en otros servidores.
- Mongo hace esto más fácil mediante el *autosharding*, que puede ser configurado.

# Esquema de fragmentación



En el esquema anterior participan tres componentes principales:

**Fragmentos:** los fragmentos se utilizan para almacenar datos.

- Proporcionan alta disponibilidad y consistencia de datos.
- En el entorno de producción, cada fragmento es un conjunto de réplicas independientes.

**Servidores de configuración:** los servidores de configuración almacenan los metadatos del clúster. Estos datos contienen una asignación del conjunto de datos del clúster a los fragmentos.

- El enrutador de consultas utiliza estos metadatos para dirigir las operaciones a fragmentos específicos.
- En el entorno de producción, los clústeres fragmentados tienen exactamente 3 servidores de configuración.

**Enrutadores de consulta:** los enrutadores de consulta son básicamente instancias Mongo, la interfaz con aplicaciones cliente y las operaciones directas al fragmento apropiado.

- El enrutador de consultas procesa y dirige las operaciones a particiones y, a continuación, devuelve los resultados a los clientes.
- Un clúster fragmentado puede contener más de un enrutador de consultas para dividir la carga de solicitudes del cliente.
- Un cliente envía solicitudes a un enrutador de consultas. Generalmente, un clúster fragmentado tiene muchos enrutadores de consulta.

# Configuración de fragmentación

- Se necesita un servidor para realizar un seguimiento de las claves para saber las ciudades que van al servidor *frag1*, *frag2* o *frag3*.
- En Mongo, se crea un *servidor de configuración* (que es un servicio *mongod* normal) que realiza un seguimiento de qué servidor posee qué valores.
- Se deberá crear e inicializar un conjunto de réplicas para la configuración del clúster.
- `c:\[path]\mongod --configsvr --  
replicaSet empresa --dbpath c:\data\config --  
port 27020`
- Ahora, para el servidor de configuración *localhost:27020* se inicia el clúster de servidores de configuración (con solo un miembro para este ejemplo, pueden ser varios):
  - `mongo --host localhost --port 27020`
  - `> rs.initiate({  
_id: "empresa",  
configsvr: true,  
members: [{_id: 0, host: "localhost:27020"}]} )  
{ "ok" : 1}  
> rs.status().ok  
1`

- Para la creación de las réplicas, se debe cambiar el parámetro **shardsvr** necesario para ser un servidor de fragmentos (solo es capaz de fragmentar).
  - c:\...\mongod --**shardsvr** --**rep1Set** **empresa** --dbpath c:\data\frag1 --port 27011
  - c:\...\mongod --**shardsvr** --**rep1Set** **empresa** --dbpath c:\data\frag2 --port 27012
- c:\...\mongod --**shardsvr** --**rep1Set** **empresa** --dbpath c:\data\frag3 --port 27013
- Estos nodos servirán para los fragmentos.
- Finalmente, se debe ejecutar otro servidor como único punto de entrada para los clientes. Este servidor se conectará al servidor de configuración para realizar un seguimiento de la información de fragmentación almacenada allí.

\$ mongos --**configdb** **empresa/localhost:27020** --port 27030

- *mongos* es un clon ligero de un servidor *mongod*, lo que lo convierte en el intermediario perfecto para que los clientes se conecten a múltiples servidores fragmentados.

- Para establecer los criterios de fragmentación, se debe establecer una conexión en la terminal del servidor de configuración, en la base de datos de administración:

```
mongo --host localhost --port 27030
```

- Y creando los fragmentos:

```
> sh.addShard('localhost:27011')
{ "shardAdded" : "shard0000", "ok" : 1 }
> sh.addShard('localhost:27012')
{ "shardAdded" : "shard0001", "ok" : 1 }
```

- Para establecer el atributo de fragmentación:

```
> db.runCommand({ enablesharding : "empresa" })
{ "ok" : 1 }
> db.runCommand({ shardcollection : "empresa.empleados",
key : {apellidos : 1} })
{ "collectionssharded" : "empresa.empleados", "ok" : 1 }
```

# Consideraciones

En la configuración de MongoDB, un problema es decidir quién es promovido cuando un nodo maestro se cae.

- MongoDB se ocupa de esto dando un voto a cada servicio ***mongod***, y el que tiene los datos más recientes es elegido el nuevo maestro.

Cuando los nodos vuelven a aparecer, entran en un estado de recuperación e intentan resincronizar sus datos con el nuevo nodo maestro.

- ¿qué pasaría si el maestro original tuviera datos que aún no se propagan? Esas operaciones se abandonan.
- Una escritura en un conjunto de réplicas de Mongo no se considera correcta hasta que la mayoría de los nodos tienen una copia de los datos.

MongoDB espera un número impar de nodos totales en el conjunto de réplicas. Si hay problemas de conexión, el fragmento más grande tiene mayoría y puede elegir un maestro y continuar atendiendo las solicitudes. Sin una mayoría clara, no se pudo alcanzar el quórum.

Con un conjunto de réplicas de cuatro nodos, se tendrá al maestro original, pero debido a que no puede ver una *clara mayoría* de la red, el maestro se retira. El otro conjunto tampoco podrá elegir un maestro porque tampoco puede comunicarse con una clara mayoría de nodos. Ambos conjuntos ahora no pueden procesar solicitudes y el sistema está efectivamente caído.

Algunas bases de datos (como CouchDB) están diseñadas para permitir múltiples maestros, pero Mongo no lo está, por lo que no está preparado para resolver las actualizaciones de datos entre ellos. MongoDB se ocupa de los conflictos entre múltiples maestros simplemente no permitiéndolos.

Debido a que es un sistema CP, Mongo siempre conoce el valor más reciente actualizado; el cliente no necesita decidir. La preocupación de Mongo es una fuerte consistencia en las escrituras, y evitar en lo posible un escenario multimaestro.

# GridFS

- Mongo incorpora un sistema de archivos distribuido llamado GridFS.
- Mongo viene incluido con una herramienta de línea de comandos para interactuar con GridFS llamada mongofiles.
- GridFS divide un archivo en trozos y almacena cada fragmento de datos en un documento separado, cada uno de tamaño máximo 255k.
- De forma predeterminada, GridFS utiliza dos colecciones **fs.files** y **fs.chunks** para almacenar los metadatos del archivo y los fragmentos.
- Cada fragmento se identifica por su campo objectId único **\_id**. El archivo **fs.files** sirve como documento primario.
- El campo **files\_id** del documento **fs.chunks** vincula el fragmento a su elemento primario.

# GridFS

- Ejemplo de listado de archivos:

```
$ mongofiles -h localhost:27020 list
```

- Para subir cualquier archivo:

```
$ mongofiles -h localhost:27020 put archivo.txt
```

```
$ mongofiles -h localhost:27020 list
```

```
2017-05-11T20:04:39.019-0700 connected to: localhost:27020
```

```
archivo.txt 2032
```

```
>db.fs.files.find()
```

```
{  
  _id: ObjectId('534a811bf8b4aa4d33fdf94d'),  
  filename: "song.mp3",  
  chunkSize: 261120,  
  uploadDate: new Date(1397391643474), md5: "e4f53379c909f7bed2e9d631e15c1c41",  
  length: 10401959  
}  
>db.fs.chunks.find({files_id:ObjectId('534a811bf8b4aa4d33fdf94d') })
```

# Fortalezas de Mongo

La principal fortaleza de Mongo radica en su capacidad para manejar grandes cantidades de datos (y grandes cantidades de solicitudes) mediante replicación y escalamiento horizontal.

- Pero también tiene el beneficio adicional de un modelo de datos muy flexible. No es necesario ajustarse a un esquema.

Finalmente, MongoDB fue construido para ser fácil de usar.

- Esto no es por accidente y es una de las razones por las que Mongo tiene tanta participación entre las personas que han desertado del campo de la base de datos relacionales.

# Debilidades de Mongo

Mongo fomenta la desnormalización de los esquemas (al no tener ninguno) y eso puede ser demasiado para algunos.

- Puede ser peligroso insertar cualquier valor de cualquier tipo en una colección. Un solo error tipográfico puede causar horas de dolor de cabeza si no piensa en mirar los nombres de campo y los nombres de colección. La flexibilidad de Mongo generalmente no es importante si su modelo de datos ya está bastante maduro.

Debido a que Mongo se centra en grandes conjuntos de datos, funciona mejor en clústeres grandes, lo que puede requerir cierto esfuerzo para diseñar y administrar.

- A diferencia de algunas bases de datos cluster donde agregar nuevos nodos es un proceso transparente y relativamente indoloro, la configuración de un clúster de Mongo requiere un poco más de previsión.