

# Bases de datos relacionales y SQL

permite enviar el código CERCA de los datos  
puede haber diferencias de ordenes de magnitud entre ejecutar el código en los datos o llevar los datos al código

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

el examen 5-6 preguntas. Algunas son dime lo que sabes de X (en un trozo pequeño) y algunas preguntas que tienen que ver con la práctica: por qué esta esa línea ahí y qué hace, qué hace esta consulta? etc (en teoría ESTA en el proyecto así que facilito)

**Curso 2023/2024**

- **Modelo relacional y creación del esquema**
- **Consulta de datos**
  - ▷ **Lenguajes formales: Álgebra y Cálculo Relacional**
  - ▷ **Sintaxis básica**
  - ▷ **Valores nulos**
  - ▷ **Agregación**
  - ▷ **Subconsultas**
  - ▷ **Operadores de conjunto**
  - ▷ **Joins**
- **Transacciones**
- **Otras funcionalidades**

# Modelo relacional y creación del esquema

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

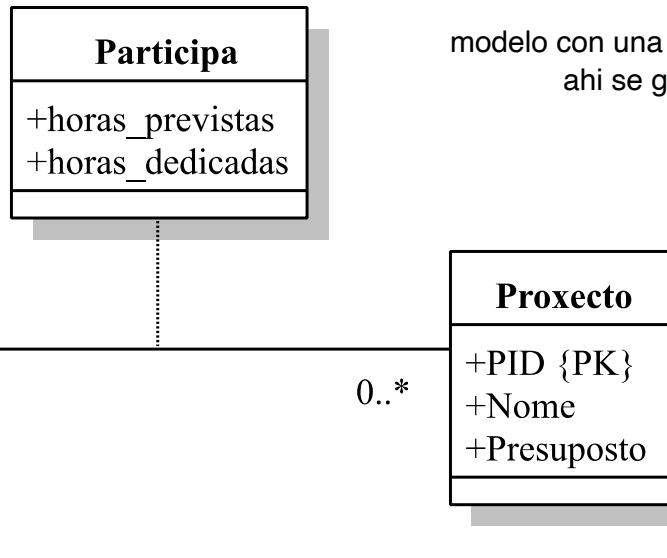
Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.



modelo con una unica estructura de datos: la tabla  
ahi se guardan todo tipo de datos

son tuplas: (1, tráfico, 198000), ...

## Proxecto

PID	Nome	Presupuesto
1	Tráfico	198000
2	Xardinería	97000
3	Festas	86000

## Empleado

DNI	Nome	Salario	Coste_hora
23456238	Alfredo	35000	45
25368964	Sofía	43000	60
58325647	Ricardo	29500	30
78878965	Elena	40500	55
78532564	Ernesto	41000	56

## Participa

Empleado	Proxecto	horas_previstas	horas_dedicadas
23456238	1	1800	95
25368964	1	800	30
23456238	2	600	10
78878965	2	800	5
78532564	2	300	100

en la tabla “participa” se referencian valores de las tablas “empleados” y “proxecto”

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.



## ■ Tipos de Dato

- ▷ **char(n)**: tamaño fijo
- ▷ **varchar(n)**: tamaño variable
- ▷ **int, smallint**
- ▷ **numeric(p, d)**: punto fijo
- ▷ numeros reales con numero fijo de decimales
- ▷ **real, double precision**: Punto flotante
- ▷ **float (n)**: mínima precisión de n dígitos
- ▷ **date, time, timestamp, interval**
- ▷ timestamp - timestamp = interval
- ▷ **Etc.**



# Modelo relacional y creación del esquema

## Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

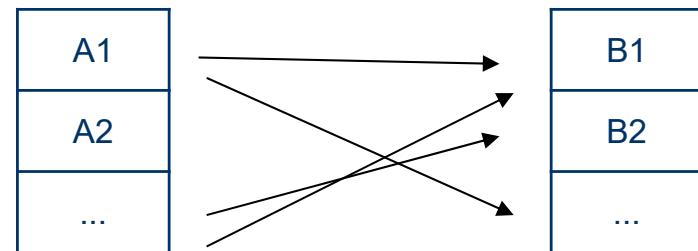
Oper. conjunto

Joins

Transacciones

Etc.

## TipoA      Relación      TipoB



## Relación

A: TipoA	B: TipoB
A1	B1
A1	B12
...	...

Tipos no  
necesariamente  
distintos

**NombreR**

A1: T1	A2: T2	...	AN: TN
A1x	A2u	...	ANr
A1y	A2v	...	ANs
...	...	...	...

## Esquema

Nombre de la relación + conjunto de pares (nombre de atributo, tipo de datos)

NombreR (A1:T1, A2:T2, ..., AN:TN)

## Contenido

Subconjunto del producto cartesiano  
de los N tipos de datos

## Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

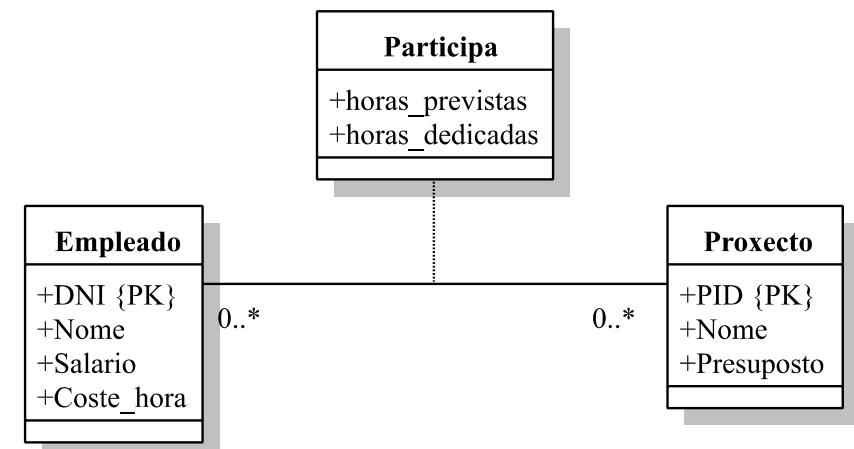
Etc.

## ■ Lenguaje de definición de datos (DDL): Sintaxis básica

- ▷ `CREATE TABLE Empleado ( dni char(8), nome varchar(500) NOT NULL, salario decimal(8,2), coste_hora decimal(6,2) NOT NULL DEFAULT 40, primary key (dni));`
- la clave primaria que no se puede repetir puede ser una combinacion de varias columnas
- ▷ `CREATE TABLE Proxecto( pid int primary key, nome varchar(500) NOT NULL, presuposto decimal(12, 2));`

Esquema y  
Restricciones

Se puede usar  
**UNIQUE (a1, a2, ... )**  
para definir otras  
claves candidatas



## Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

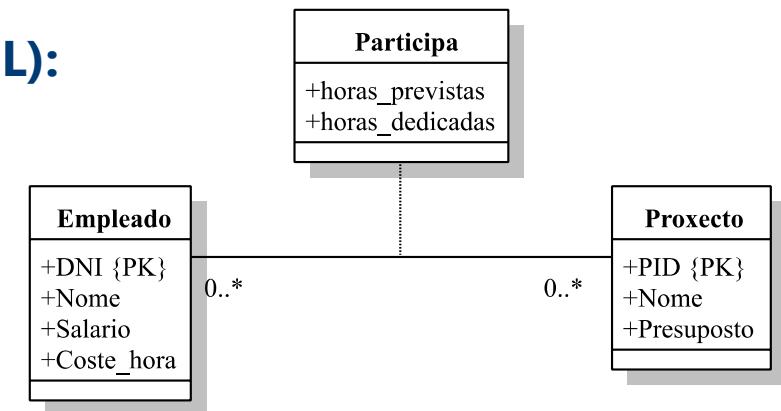
Transacciones

Etc.

## ■ Lenguaje de definición de datos (DDL): Restricciones

▷ `CREATE TABLE participa (`  
 `empleado char(8),`  
 `proxecto int,`  
 `horas_previstas int NOT NULL,`  
 `horas_dedicadas int DEFAULT 0,`  
 `primary key (empleado, proxecto),`  
 `foreign key (empleado) references Empleado(dni)`  
 `on delete restrict`  
 `on update cascade,`  
 `foreign key (proxecto) references Proxecto(pid)`  
 `on delete cascade`  
 `on update cascade,`  
 `check (horas_previstas >= horas_dedicadas));`

todo lo que no cumpla esta condicion dará error



- **restrict**
- **cascade**
- **set null**
- **set default**

cuando borras en la otra tabla,  
**restrict** es por defecto (no  
deja). En cascada borra la fila  
de ese empleado (en general  
sin avisar)

## Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

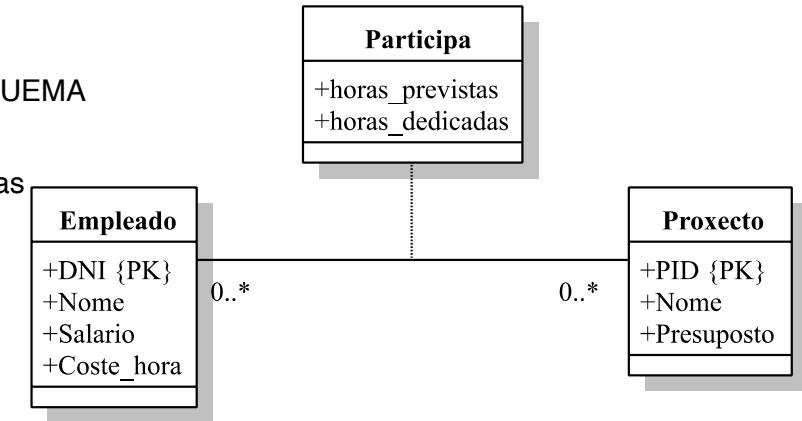
Joins

Transacciones

Etc.

## ■ Lenguaje de definición de datos (DDL): Eliminación y modificación del esquema

- ▷ **DROP TABLE** participa;
- ▷ **ALTER TABLE Empleado** modifica el ESQUEMA  
**ADD email varchar(500);**  
no podrías poner NOT NULL de primeras
- ▷ **ALTER TABLE Empleado**  
**DROP COLUMN email;**
- ▷ ETC.



Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.



## ■ Álgebra relacional: Operaciones primitivas

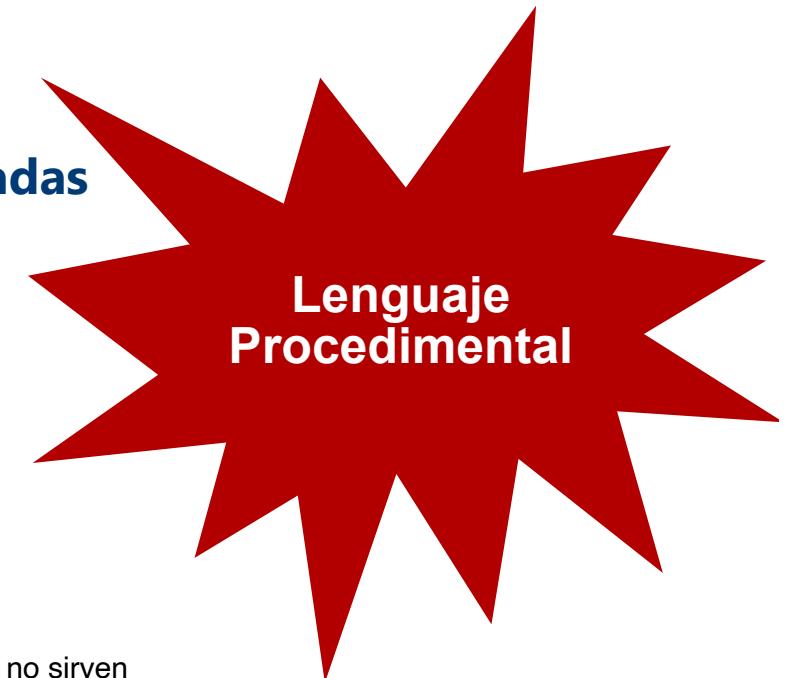
- ▷ Selección:  $\sigma_p(R)$  selecciona filas que cumplen el predicado P
- ▷ Proyección (proyección generalizada):  $\pi_{A_1, A_2, \dots, A_n}(R)$
- ▷ Producto cartesiano:  $R_1 \times R_2$  construye una tabla con todas las combinaciones posibles entre las tablas
- ▷ Unión:  $R_1 \cup R_2$
- ▷ Diferencia:  $R_1 - R_2$

## ■ Álgebra relacional: Operaciones derivadas

- ▷ Intersección:  $R_1 \cap R_2$
- ▷ Join
  - Theta Join:  $R_1 \bowtie_p R_2 \equiv \sigma_p(R_1 \times R_2)$
  - Natural Join:  $R_1 \bowtie R_2$

join es similar al producto cartesiano pero cargándose las filas que no sirven

lenguaje procedimental



Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

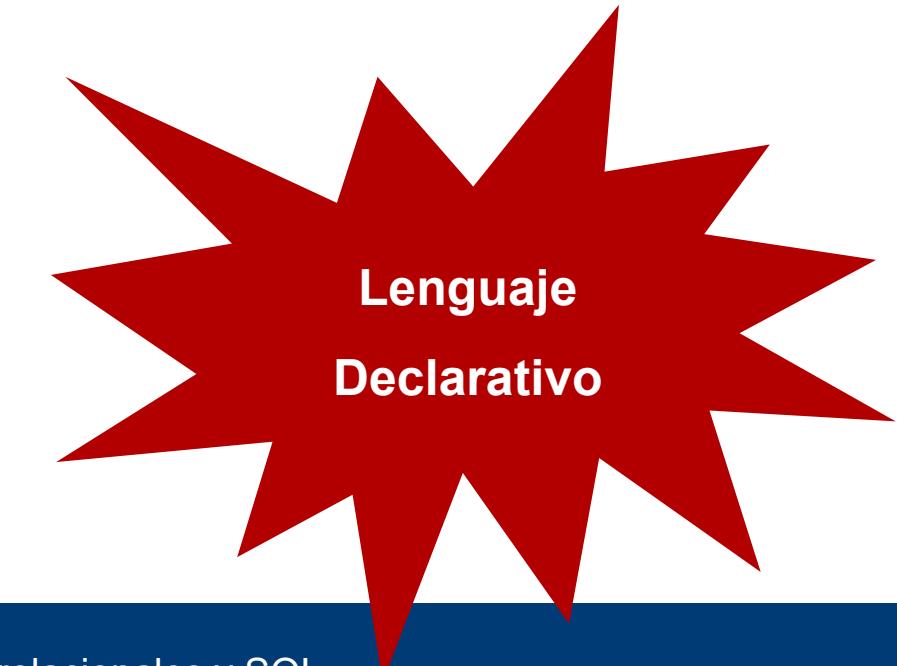
Etc.



## ■ Cálculo de predicados (lógica de predicados)

- ▷ **Constantes, variables, predicados, funciones**
- ▷ **Conectores:**  $\vee, \wedge, \rightarrow, \leftrightarrow, \neg$
- ▷ **Cuantificadores:**  $\forall, \exists$
- ▷ **Expresiones:**

–  $\text{numero}(x) \wedge \neg (\exists y(\text{numero}(y) \wedge \text{mayor}(x, y)))$



Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

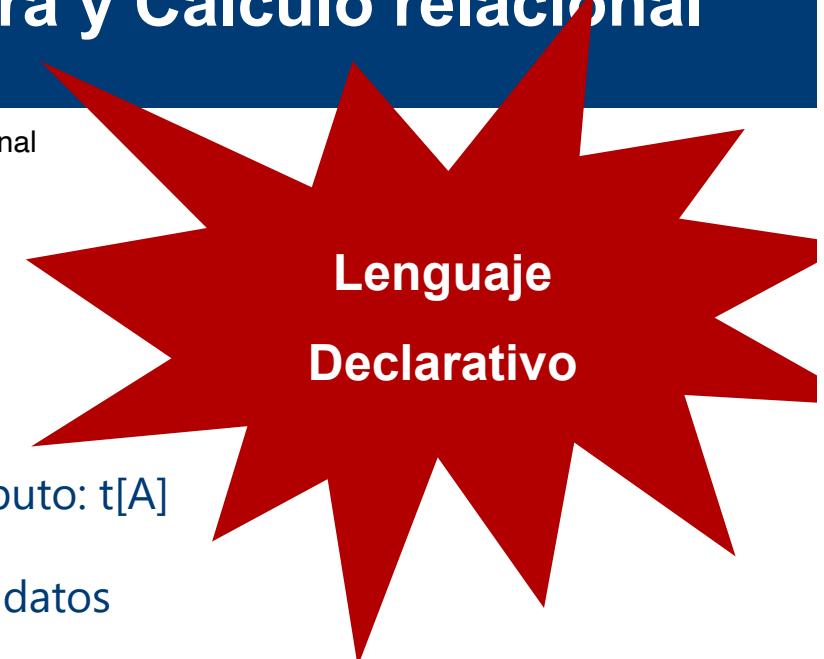
Transacciones

Etc.

## ■ Cálculo relacional (de tuplas)

sql basado en calculo relacional

- ▷ Expresiones:  $\{t \mid P(t)\}$
- ▷ Variables referencian tuplas:  $t$
- ▷ Función que accede al valor de un atributo:  $t[A]$
- ▷ Constantes son valores de los tipos de datos
- ▷ Predicados
  - Relaciones:  $\text{Empleado}(t)$
  - Otros predicados implementados en el sistema:  $x > y$ ,  $x = y$ , etc.
- ▷ Conectores:  $\vee, \wedge, \rightarrow, \neg$
- ▷ Cuantificadores:  $\exists x(\text{Empleado}(x) \wedge x[\text{salario}] > y[\text{salario}])$



Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Cálculo relacional (de tuplas)

### ▷ **Expresiones Seguras** (las expresiones no pueden generar conjuntos infinitos)

- Una expresión podría generar un resultado infinito
  - $\{ t \mid \neg \text{Empleado}(t) \}$
- El lenguaje debería de evitar que se puedan construir expresiones como esas
- Por ejemplo, forzar a que todas las variables tengan que estar referenciadas dentro de un predicado de relación sin negar.
  - $\{ t \mid \neg \text{Empleado}(t) \wedge \text{cliente}(t) \}$

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Procesamiento de consultas

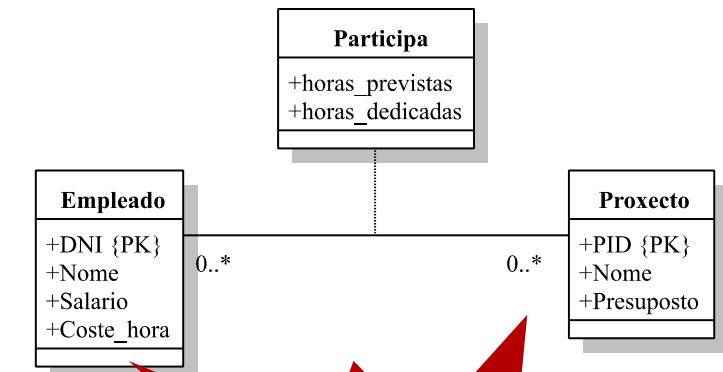
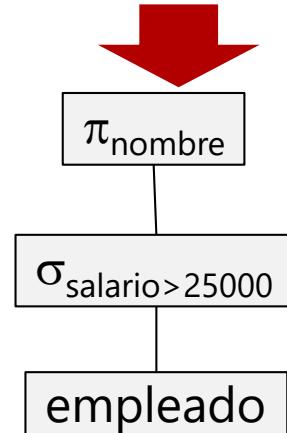
- ▷ Nombre de los empleados con un salario mayor de 25000

$\{t[\text{nombre}] \mid \text{empleado}(t) \wedge t[\text{salario}] > 25000\}$



va leyendo y mandando los datos, no los almacena y luego envia

$\pi_{\text{nombre}}(\sigma_{\text{salario}>25000}(\text{Empleado}))$



Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

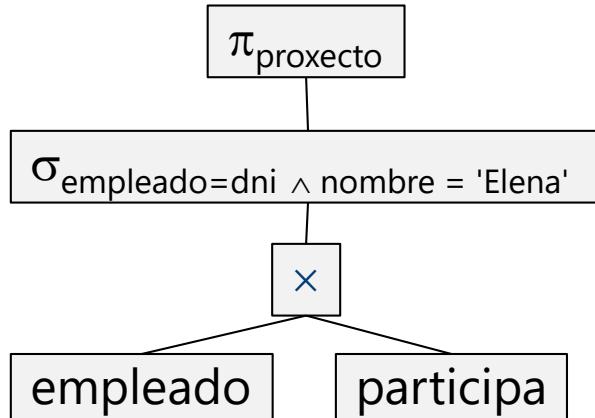
Etc.

## ■ Procesamiento de consultas

- ▷ Identificadores de proyecto de los proyectos en los que ha participado 'Elena'

```
{  
    p[proxecto] |  
    participa(p) ∧ empleado(e) ∧  
    p[empleado]=e[dni] ∧  
    e[nombre]='Elena'  
}
```

mal arbol, producto es muy lento



Ejercicio



Participa
+horas_previstas
+horas_dedicadas

Empleado
+DNI {PK}
+Nome
+Salario
+Coste_hora

Proxecto
+PID {PK}
+Nome
+Presupuesto

```
{  
    p[proxecto] |  
    participa(p) ∧  
    \exists e(empleado(e) ∧  
            p[empleado]=e[dni] ∧  
            e[nombre]='Elena')  
}
```

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

- **Sintaxis amigable (parecido a expresarlo en Inglés)**
- **Expresiones seguras del cálculo relacional de tuplas**
- **Generación de árboles de ejecución basados en álgebra relacional**
- **Uso de multi conjuntos en lugar de conjuntos**

③

**SELECT** e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub>

$\pi_{e_1, e_2, \dots, e_n}$

①

**FROM** R<sub>1</sub> as r<sub>1</sub>, R<sub>2</sub> as r<sub>2</sub>, ..., R<sub>m</sub> as r<sub>m</sub>

$R_1 \times R_2 \times \dots \times R_m$

②

**WHERE** p

$\sigma_p$

{ e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub> |  
 R<sub>1</sub>(r<sub>1</sub>)  $\wedge$  R<sub>2</sub>(r<sub>2</sub>)  $\wedge$  ...  $\wedge$  R<sub>m</sub>(r<sub>m</sub>)  
 $\wedge$  p (r<sub>1</sub>, r<sub>2</sub>, ..., r<sub>m</sub>) }

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

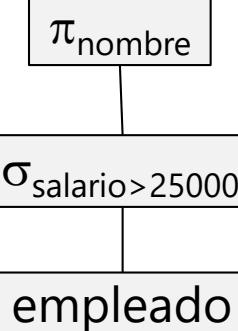
- **Nombre de los empleados con un salario mayor de 25000**

Ejercicio

```
SELECT nome
FROM empleado as e
WHERE e.salario > 25000
```

```
SELECT nome
FROM empleado
WHERE salario > 25000
```

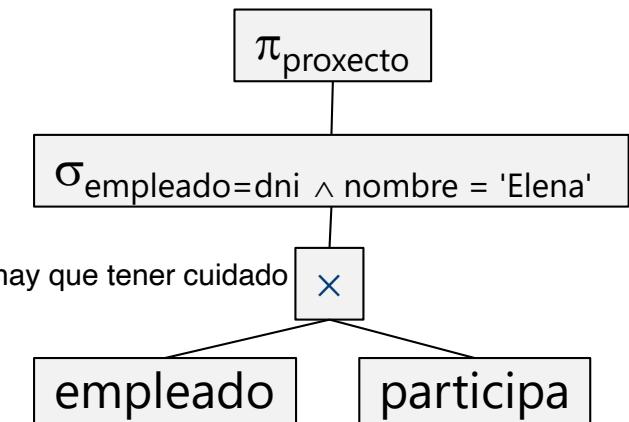
si solo hay una tabla no hace falta usar un alias



- **Identificadores de los proyectos en los que ha participado 'Elena'**

```
SELECT p.proxecto
FROM empleado as e, participa as p
WHERE e.dni=p.empleado and
      e.nombre='Elena'
```

la condicion nunca esta en el enunciado, hay que tener cuidado



la condicion debe enlazar las dos tablas (suele ser igualar la clave primaria de una con la clave foranea de la otra que apunta a la principal. Así, nos quitamos las filas que no queremos del producto, sino estará mal y será lento)

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

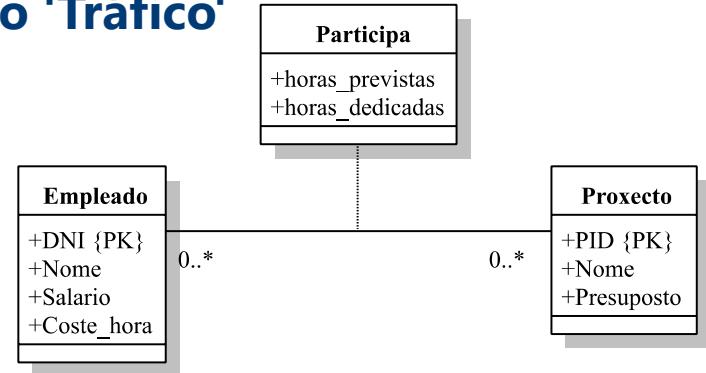
Oper. conjunto

Joins

Transacciones

Etc.

- Nombres y salarios de los empleados que ya han dedicado entre 20 y 40 horas al proyecto 'Tráfico'
- Ordena el resultado por salario de forma descendente y después por nombre de forma ascendente



③ **SELECT e.nome, e.salario**  
el select crea nuevas columnas que usa el order by, el where NO crea columnas

① **FROM empleado as e, participa pa, proxecto pr**

② **WHERE e.dni=pa.empleado**  
enlazamos empleado a participa y proxecto a participa

and pr.pid=pa.proxecto

and pa.horas\_dedicadas between 20 and 40

and pr.nombre = 'Tráfico'

④ **ORDER BY e.salario DESC, e.nombre ASC**

ASC esta por defecto

Ya no es  
una  
relación,  
es un  
listado

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

- Nombres de los empleados que han participado en algún proyecto con un presupuesto mayor de 90000

Ejercicio

## Empleado

DNI	Nome	Salario	Coste_hora
23456238	Alfredo	35000	45
25368964	Sofía	43000	60
58325647	Ricardo	29500	30
78878965	Elena	40500	55
78532564	Ernesto	41000	56

## Participa

Empleado	Proyecto	horas_previstas	horas_dedicadas
23456238	1	1800	95
25368964	1	800	30
23456238	2	600	10
78878965	2	800	5
78532564	2	300	100

elimina repetidos de las filas (no colocar parentesis). e.nome y (e.nome) harán lo mismo aquí

```
SELECT e.nome
FROM empleado as e, participa pa,
      proxecto pr
WHERE e.dni=pa.empleado
      and pr.pid=pa.proyecto
      and pr.presupuesto > 90000
```

Nome
Alfredo
Sofía
Alfredo
Elena
Ernesto

```
SELECT DISTINCT e.nome
FROM empleado as e, participa pa,
      proxecto pr
WHERE e.dni=pa.empleado
      and pr.pid=pa.proyecto
      and pr.presupuesto > 90000
```

Nome
Alfredo
Sofía
Elena
Ernesto

Alfredo sale dos veces porque el resultado es un multiconjunto

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

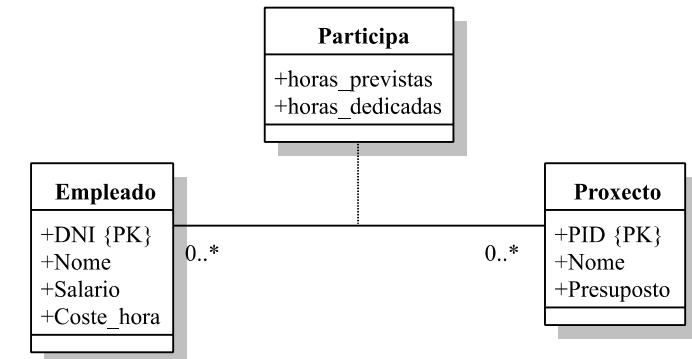
## ■ Cada tipo de dato tiene un valor especial **NULL**

- ▷ Valor no conocido, todavía no insertado, etc.
- ▷ También el tipo de datos Boolean tiene el valor **NULL**. Las siguientes expresiones booleanas evalúan a **NULL**
  - **NULL = NULL**
  - **A = NULL**
- ▷ Cláusula WHERE obtiene solo las tuplas para las que el predicado se evalúa a **TRUE**



```
SELECT *
FROM empleado
WHERE salario = NULL
```

esta devuelve una tabla (conjunto) vacío



esto si queremos las que tiene salario NULO o NO NULO

```
SELECT *
FROM empleado
WHERE salario IS NULL
```

```
SELECT *
FROM empleado
WHERE salario IS NOT NULL
```

DDL

Lenguajes  
Formales

Sintaxis básica

Nulos

**Agregación**

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Obtener la media del salario de todos los empleados

Función  
de  
Agregado

```
SELECT AVG(salario)
FROM empleado
```

se aplican a un conjunto y devuelven un único valor,  
por lo que hay solo una fila como resultado

Empleado

DNI	Nome	Salario	Coste_hora
23456238	Alfredo	35000	45
25368964	Sofía	43000	60
58325647	Ricardo	29500	30
78878965	Elena	40500	55
78532564	Ernesto	41000	56

Participa

Empleado	Proyecto	horas_previstas	horas_dedicadas
23456238	1	1800	95
25368964	1	800	30
23456238	2	600	10
78878965	2	800	5
78532564	2	300	100

al calcular en vertical, todo es homogéneo  
(mismo tipo de datos), pero no tengo un número  
fijo de datos, el contenido cambia mucho

$$\text{AVG} = 37800$$

```
SELECT AVG(salario), MIN(salario), MAX(salario), COUNT(*), COUNT(DISTINCT coste_hora)
FROM empleado
```

```
SELECT AVG(salario), nome
FROM empleado
```

da error porque uno es una fila y lo otro muchas

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

- Para cada proyecto, obtener su nombre y el número de empleados que participan

```

④ SELECT pr.nome, COUNT(*)
① FROM proxecto pr, participa pa
② WHERE pr.pid = pa.proxecto
③ GROUP BY pr.pid, pr.nome
  
```

la tabla grande combina proxecto con participa. Da una fila por cada fila de la tabla participa, completada con los datos del respectivo proyecto. Como queremos una fila por cada proyecto, con el nombre y el id, agrupamos y contamos

Empleado				Participa			
DNI	Nome	Salario	Coste_hora	Empleado	Proyecto	horas_previstas	horas_dedicadas
23456238	Alfredo	35000	45	23456238	1	1800	95
25368964	Sofía	43000	60	25368964	1	800	30
58325647	Ricardo	29500	30	23456238	2	600	10
78878965	Elena	40500	55	78878965	2	800	5
78532564	Ernesto	41000	56	78532564	2	300	100

Empleado	Proyecto	horas_previstas	horas_dedicadas	PID	Nome	Presupuesto
23456238	1	1800	95	1	Tráfico	198000
25368964	1	800	30	1	Tráfico	198000
23456238	2	600	10	2	Xardinería	97000
78878965	2	800	5	2	Xardinería	97000
78532564	2	300	100	2	Xardinería	97000

la cláusula de ordenar por el id es por si hay dos proyectos con el mismo nombre (no hay restricción sobre eso)

Nome	count
Tráfico	2
Xardinería	3

# Agregación

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación



- Para cada empleado, obtener la suma de horas que ha dedicado ya a sus proyectos



Proxecto

PID	Nome	Presupuesto
1	Tráfico	198000
2	Xardinería	97000
3	Festas	86000

Empleado

DNI	Nome	Salario	Coste_hora	Empleado	Proxecto	horas_previstas	horas_dedicadas
23456238	Alfredo	35000	45	23456238	1	1800	95
25368964	Sofía	43000	60	25368964	1	800	30
58325647	Ricardo	29500	30	23456238	2	600	10
78878965	Elena	40500	55	78878965	2	800	5
78532564	Ernesto	41000	56	78532564	2	300	100

Participa

```
SELECT e.dni, e.nombre, SUM(pa.horas_dedicadas)
FROM empleado e, participa p
WHERE e.dni = p.empleado
GROUP BY e.dni
```

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

- Para cada empleado que participe en dos o más proyectos, obtener el número de horas que tiene previsto dedicar en media a cada proyecto

⑤ **SELECT** e.dni, e.nombre,  
AVG(horas\_previstas)  
**FROM** empleado e, participa p  
**WHERE** e.dni = p.empleado  
**GROUP BY** e.dni  
**HAVING** count(\*) >= 2

si no se han agrupado los datos no se pueden poner funciones de agregado  
filtra los grupos generados, como un where

Empleado

DNI	Nome	Salario	Coste_hora
23456238	Alfredo	35000	45
25368964	Sofía	43000	60
58325647	Ricardo	29500	30
78878965	Elena	40500	55
78532564	Ernesto	41000	56

Participa

Empleado	Proxecto	horas_previstas	horas_dedicadas
23456238	1	1800	95
25368964	1	800	30
23456238	2	600	10
78878965	2	800	5
78532564	2	300	100

Subconsultas

dni	Nome	Salario	coste_hora	Empleado	Proxecto	horas_previstas	horas_dedicadas
23456238	Alfredo	35000	45	23456238	1	1800	95
23456238	Alfredo	29500	30	23456238	2	600	10
25368964	Sofía	43000	60	25368964	1	800	30
78878965	Elena	40500	55	78878965	2	800	5
78532564	Ernesto	41000	56	78532564	2	300	100

Oper. conjunto

Joins

Transacciones

Etc.

dni	Nome	horas_previstas
23456238	Alfredo	1200

Proxecto

PID	Nome	Presupuesto
1	Tráfico	198000
2	Xardinería	97000
3	Festas	86000

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

- Para cada proyecto que tenga más de dos empleados, obtener su identificador, nombre, presupuesto total, presupuesto previsto para personal y presupuesto previsto para otros gastos



Proxecto

PID	Nome	Presupuesto
1	Tráfico	198000
2	Xardinería	97000
3	Festas	86000

Empleado

DNI	Nome	Salario	Coste_hora	Empleado	Proxecto	horas_previstas	horas_dedicadas
23456238	Alfredo	35000	45	23456238	1	1800	95
25368964	Sofia	43000	60	25368964	1	800	30
58325647	Ricardo	29500	30	23456238	2	600	10
78878965	Elena	40500	55	78878965	2	800	5
78532564	Ernesto	41000	56	78532564	2	300	100

```

SELECT pr.pid, pr.nome, pr.presupuesto,
       SUM(pa.horas_previstas*e.coste_hora) AS personal,
       pr.presupuesto - SUM(pa.horas_previstas*e.coste_hora) AS otros_gastos
FROM proxecto pr, participa pa, empleado e
WHERE pr.pid=pa.proxecto and e.dni=pa.empleado
GROUP BY pr.pid
HAVING count(*) > 2
    
```

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

**Subconsultas**

Oper. conjunto

Joins

Transacciones

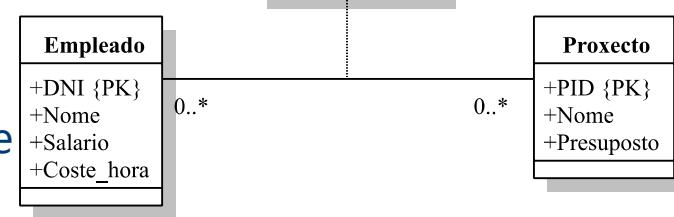
Etc.

## ■ Subconsultas en la cláusula FROM

```
SELECT e1, e2, ..., en
  FROM R1 as r1, (SELECT ...) as r2, ..., Rm as rm
 WHERE p
```

Devuelve  
una tabla

Ejercicio



- ▷ Permiten implementar estrategias de tipo **divide y vencerás**
- ▷ Para cada proyecto, obtén el número de empleados con un coste por hora mayor de 40 y el número de empleados con un coste por hora menor o igual que 40

cte - common table expressions: para simplificar la lectura de este código (no lo vamos a ver)

```
SELECT p.pid, p.nome, Tmas40.mas40, Tmenos40.menos40
  FROM proyectos p, combina dos tablas generados con la propia tabla de proyectos
```

```
(SELECT p.proxecto, count(*) as mas40
  FROM participa p, empleado e
 WHERE p.empleado=e.dni and coste_hora > 40
 GROUP BY p.proxecto) as Tmas40,
```

una tabla

```
(SELECT p.proxecto, count(*) as menos40
  FROM participa p, empleado e
 WHERE p.empleado=e.dni and coste_hora <= 40
 GROUP BY p.proxecto) as Tmenos40
```

otra tabla

```
WHERE p.pid = Tmas40.proxecto and p.pid = Tmenos40.proxecto
```

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

**Subconsultas**

Oper. conjunto

Joins

Transacciones

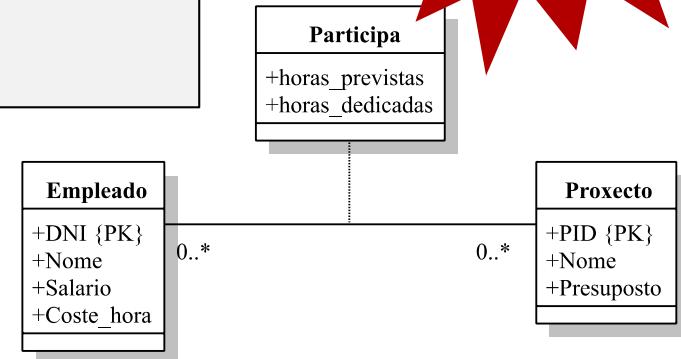
Etc.

## ■ Subconsultas en otras subcláusulas

```
SELECT e1, (SELECT ...) as e2, ..., en
FROM R1 as r1, R2 as r2, ..., Rm as rm
WHERE ... (SELECT ...) ...
GROUP BY ...
HAVING ... (SELECT ...) ...
```

Devuelve  
un único  
valor

Ejercicio



- ▷ Obtén todos los datos del empleado con el máximo salario

```
SELECT *
FROM empleado
WHERE salario = (SELECT MAX(salario) FROM empleado)
```

numero

si esto SOLO devuelve una fila, lo convertira a numero

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

**Subconsultas**

Oper. conjunto

Joins

Transacciones

Etc.

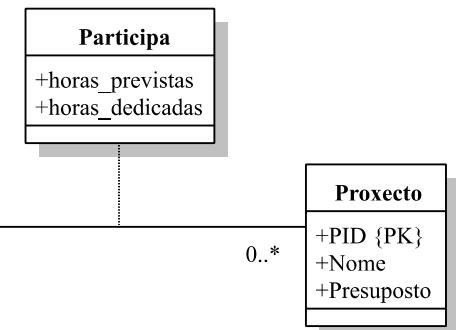
## ■ Predicados de conjuntos y cuantificadores

- ▷ Se usan en la cláusulas **WHERE** y **HAVING**
  - **IN**
  - Modificadores de comparaciones (**ANY**, **SOME**, **ALL**)
  - **EXISTS**
- ▷ Obtener los datos de los empleados que han participado en algún proyecto con presupuesto mayor de 90000

```
SELECT DISTINCT e.* esta ya la sabíamos
FROM empleado e, participa pa, proxecto pr
WHERE e.dni = pa.empleado
    and pr.pid = pa.proxecto
    and pr.presupuesto > 90000
```

```
SELECT *
FROM empleado
WHERE dni IN
    (SELECT empleado
     FROM participa pa, proxecto pr
     WHERE pa.proxecto=pr.pid
        and pr.presupuesto > 90000)
```

```
SELECT *
FROM empleado
WHERE dni = ANY
    (SELECT empleado
     FROM participa pa, proxecto pr
     WHERE pa.proxecto=pr.pid
        and pr.presupuesto > 90000)
```



Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

**Subconsultas**

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Predicados de conjuntos y cuantificadores

- ▷ Se usan en la cláusulas **WHERE** y **HAVING**
  - **IN**
  - Modificadores de comparaciones (**ANY**, **SOME**, **ALL**)
  - **EXISTS**
- ▷ Obtener los datos del empleado con el mayor salario

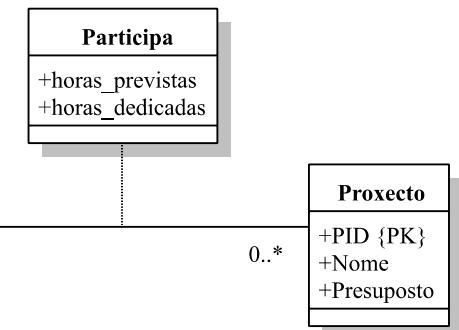
```
SELECT *
FROM empleado
WHERE salario = (SELECT MAX(salario)
                  FROM empleado)
```

```
SELECT *
FROM empleado
WHERE salario > = ALL (SELECT salario
                           FROM empleado)
```

```
SELECT *
FROM empleado e1
WHERE NOT EXISTS (SELECT * FROM empleado e2
                           WHERE e1.salario < e2.salario)
```

de estas pone siempre una en el examen

**Ejercicio**



$$\{e1 | \text{empleado}(e1) \wedge \neg \exists e2(\text{empleado}(e2) \wedge e1[\text{salario}] < e2[\text{salario}])\}$$

# Operadores de Conjunto

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Union, intersección y diferencia de conjuntos

une en vertical las filas  
y elimina duplicados  
por defecto

si colocas ALL no se eliminan duplicados (poner  
esta si estamos seguros de que no hay duplicados)

```
SELECT ...
FROM ...
...
UNION
SELECT ...
FROM ...
...
ORDER BY ...
```

```
SELECT ...
FROM ...
...
UNION ALL
SELECT ...
FROM ...
...
ORDER BY ...
```

```
SELECT ...
FROM ...
...
EXCEPT
SELECT ...
FROM ...
...
ORDER BY ...
```

```
SELECT ...
FROM ...
...
INTERSECT
SELECT ...
FROM ...
...
ORDER BY ...
```

las columnas deben ser  
las mismas (y el tipo)

**Recordar que las  
tablas deben ser  
compatibles**

**Por defecto se  
eliminan  
duplicados**

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ INNER Theta Join (cualquier predicado)

- ▷ Mostrar los datos de los empleados junto con los datos de su departamento

si alguien no tuviera departamento por ejemplo, no saldría en la consulta global

```
SELECT *
FROM empleado e, departamento d
WHERE e.depId = d.depId
```

theta join porque en el where se puede colocar cualquier expresión booleana

DNI	Nome	Salario	Coste_hora	depId	depId	nom_dept
23456238	Alfredo	35000	45	1	1	Marketing
25368964	Sofía	43000	60	1	1	Marketing
58325647	Ricardo	29500	30	2	2	Ventas
78878965	Elena	40500	55	2	2	Ventas

## Empleado

DNI	Nome	Salario	Coste_hora	depId
23456238	Alfredo	35000	45	1
25368964	Sofía	43000	60	1
58325647	Ricardo	29500	30	2
78878965	Elena	40500	55	2
78532564	Ernesto	41000	56	

## Departamento

depId	nom_dept
1	Marketing
2	Ventas
3	Compras

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ INNER Theta Join (cualquier predicado)

- ▷ Mostrar los datos de los empleados junto con los datos de su departamento



```
SELECT *
FROM empleado e INNER JOIN departamento d
ON (e.depId = d.depId)
```

DNI	Nome	Salario	Coste_hora	depId	depId	nom_dept
23456238	Alfredo	35000	45	1	1	Marketing
25368964	Sofía	43000	60	1	1	Marketing
58325647	Ricardo	29500	30	2	2	Ventas
78878965	Elena	40500	55	2	2	Ventas
78532564	Ernesto	41000	56			

## Empleado

DNI	Nome	Salario	Coste_hora	depId
23456238	Alfredo	35000	45	1
25368964	Sofía	43000	60	1
58325647	Ricardo	29500	30	2
78878965	Elena	40500	55	2
78532564	Ernesto	41000	56	

## Departamento

depId	nom_dept
1	Marketing
2	Ventas
3	Compras

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ INNER Equi Join (predicado =)

- ▷ Mostrar los datos de los empleados junto con los datos de su departamento

Mismo nombre  
de columna

Se elimina una  
copia del  
resultado

con el using se elimina la columna repetida

```
SELECT *
FROM empleado e JOIN departamento d
USING (depId)
```

DNI	Nome	Salario	Coste_hora	depId
23456238	Alfredo	35000	45	1
25368964	Sofía	43000	60	1
58325647	Ricardo	29500	30	2
78878965	Elena	40500	55	2
78532564	Ernesto	41000	56	

## Departamento

depId	nom_dept
1	Marketing
2	Ventas
3	Compras

DNI	Nome	Salario	Coste_hora	depId	nom_dept
23456238	Alfredo	35000	45	1	Marketing
25368964	Sofía	43000	60	1	Marketing
58325647	Ricardo	29500	30	2	Ventas
78878965	Elena	40500	55	2	Ventas

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ INNER NATURAL JOIN

- ▷ Mostrar los datos de los empleados junto con los datos de su departamento

Equi Join por todos los atributos con el mismo nombre

`SELECT * FROM empleado e NATURAL JOIN departamento d`

el asterisco significa todas las columnas que tiene y tendrá, así que si se cambian cosas en la aplicación puede darse un comportamiento no deseado

DNI	Nome	Salario	Coste_hora	depId
23456238	Alfredo	35000	45	1
25368964	Sofía	43000	60	1
58325647	Ricardo	29500	30	2
78878965	Elena	40500	55	2
78532564	Ernesto	41000	56	

## Departamento

depId	nom_dept
1	Marketing
2	Ventas
3	Compras

DNI	Nome	Salario	Coste_hora	depId	nom_dept
23456238	Alfredo	35000	45	1	Marketing
25368964	Sofía	43000	60	1	Marketing
58325647	Ricardo	29500	30	2	Ventas
78878965	Elena	40500	55	2	Ventas

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ OUTER JOINS

- Se pierden los datos de Ernesto y del departamento de Compras

```
SELECT *
FROM empleado e
NATURAL LEFT OUTER JOIN
departamento d
```

outer se puede quitar aqui, porque left o right ya son outer

```
SELECT *
FROM empleado e NATURAL RIGHT JOIN departamento d
```

```
SELECT *
FROM empleado e NATURAL FULL JOIN departamento d
```

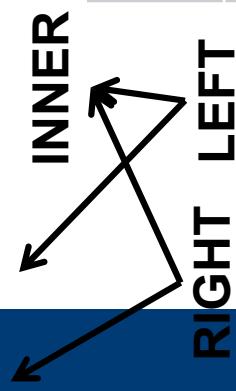
## Empleado

DNI	Nome	Salario	Coste_hora	depId
23456238	Alfredo	35000	45	1
25368964	Sofía	43000	60	1
58325647	Ricardo	29500	30	2
78878965	Elena	40500	55	2
78532564	Ernesto	41000	56	

## Departamento

depId	nom_dept
1	Marketing
2	Ventas
3	Compras

DNI	Nome	Salario	Coste_hora	depId	nom_dept
23456238	Alfredo	35000	45	1	Marketing
25368964	Sofía	43000	60	1	Marketing
58325647	Ricardo	29500	30	2	Ventas
78878965	Elena	40500	55	2	Ventas
78532564	Ernesto	41000	56		
				3	Compras



FULL

↓

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Lenguaje de Manipulación de datos: Inserción de datos

pueden colocarse subconsultas o resultados de consultas en cada una de las v

```
INSERT INTO nombre_tabla (A1, A2, ..., An)
VALUES (v11, v12, ..., v1n),
        (v21, v22, ..., v2n),
        ...
        (vm1, vm2, ..., vmn);
```

```
INSERT INTO nombre_tabla (A1, A2, ..., An)
SELECT e1, e2, ..., en
FROM ...
```

Si no se especifican  
se consideran todos  
los atributos de la  
tabla

Los no especificados  
se rellenan con valor  
por defecto o con  
nulos

Si no tienen valor  
por defecto y no  
admiten nulos se  
genera una  
excepción

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones



Etc.

## ■ Borrado de datos

```
DELETE FROM nombre_tabla  
WHERE p;
```

## ■ Modificación de datos

```
UPDATE nombre_tabla  
SET a1 = e1, a2 = e2, ..., an = en  
WHERE p;
```

e = expresion

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## TRANSACCIÓN

Unidad de ejecución de un programa que accede y posiblemente modifica varios elementos de datos

ACID (Atomicity, consistency, isolate, durability)

- Unidad simple e indivisible para el usuario (**Atomicidad**)
  - ▷ Complejo: Datos en memoria y en disco
- Operan sin interferencia de las operaciones de otras transacciones (**Aislamiento**)
- Sus efectos perduran a pesar de caídas del sistema (**Durabilidad**)
- Su ejecución aislada y atómica deja la base de datos en un estado consistente (**Consistencia**)
  - ▷ Dependiente de la aplicación

T1



## Begin Transaction

Operación

Operación

Operación

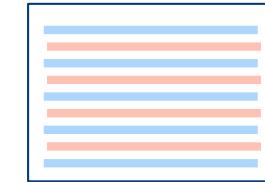
....

Operación

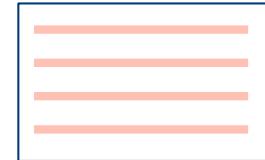
## End Transaction

Deshacer

T2

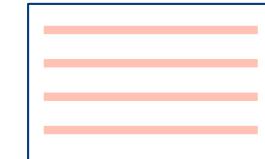


T3



T3

la mezcla debe debe ser conmutativa



el gestor se encarga de A, I, D,  
si el programador se encarga de  
la C (comutativa) todo okey

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

**Transacciones**

Etc.

## ■ Propiedades ACID

### ▷ Atomicidad (Atomicity)

- Se ejecutan todas las operaciones o no se ejecuta ninguna

### ▷ Consistencia (Consistency)

- La ejecución aislada de una transacción preserva la consistencia de la base de datos

### ▷ Aislamiento (Isolation)

- Aunque varias transacciones se ejecuten de forma concurrente, el sistema garantiza que cada par de transacciones  $T_i$  y  $T_j$ , para  $T_i$  da la impresión de que o bien  $T_j$  terminó su ejecución antes de iniciarla  $T_i$  o  $T_j$  inició su ejecución después de terminarla  $T_i$ .

### ▷ Durabilidad (Durability)

- Despues de que una transacción termine con éxito, los cambio que ha hecho en la base de datos persisten, incluso si hay fallos en el sistema.

Asegurar el aislamiento  
puede tener un impacto  
grande en el  
rendimiento del  
sistema

Necesario?  
Depende la  
aplicación

estas propiedades no son necesarias, depende si solo queremos hacer una consulta o que

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Consistencia

- ▷ Suma de A y B debe permanecer inalterada en la transacción
  - T1 no debe crear ni destruir dinero
- ▷ **Responsabilidad del programador**
- ▷ Comprobación automática de restricciones de integridad puede ayudar a cumplir esta propiedad
  - PRIMARY KEY, FOREIGN KEY, CHECK, etc.

## ■ Atomicidad

- ▷ Estado inconsistente no provocado por la mala praxis del programador
- ▷ Estado inconsistente por el que pasa la transacción es temporal
  - No se dará si se ejecuta de forma atómica
- ▷ Se mantiene un registro (log) de valores antiguos
  - Restaurar valores antiguos en caso de fallo
- ▷ **Responsabilidad de SGBDs**
  - Sistema de recuperación

T1

```
leer(A);
A:= A - 50;
escribir(A);
leer(B);
B:=B+50;
escribir(B);
```

```
leer(A);
A:= A - 50;
escribir(A);
Fallo
```

Registro  
Histórico

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Durabilidad

- ▷ Una vez ha terminada la transacción, los cambios perduran en la base de datos, incluso si hay un fallo en el sistema después de completarse la transacción.
- ▷ Para garantizar la durabilidad
  - La información sobre las modificaciones de la transacción se guarda en el disco
  - Dicha información permite al SGBD reconstruir las modificaciones cuando el sistema se reinicia después de un fallo.
- ▷ Responsabilidad del SGBDs
  - Sistema de recuperación
    - También responsable de la atomicidad.

**T1**

```
leer(A);  
A:= A - 50;  
escribir(A);  
leer(B);  
B:=B+50;  
escribir(B);
```

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

## ■ Aislamiento

- ▷ Incluso si la ejecución aislada de una transacción es Consistente, la ejecución concurrente con otra puede llevar a la base de datos a un estado inconsistente
  - El estado inconsistente puede mantenerse después del final de ambas transacciones
- ▷ Solución:
  - Ejecución secuencial de las transacciones
  - Solución poco eficiente en cuanto a rendimiento del sistema
- ▷ Esta propiedad garantiza que la ejecución concurrente de varias transacciones produce un resultado equivalente a la ejecución de las transacciones una detrás de la otra (secuencial) en algún orden
- ▷ **Responsabilidad del SGBDs**
  - Sistema de control de concurrencia

**T1**

leer(A);  
A:= A – 50;  
escribir(A);

**T2**

leer(A);  
Leer (B);  
S:=A+B  
escribir(S);

leer(B);  
B:=B+50;  
escribir(B);

# Transacciones

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

- Atomicidad y Durabilidad:  
Estados de una transacción

Después de ejecutarse la  
última instrucción.  
Todavía puede fallar

**Parcialmente Comprometida**

**Activa**

Falla  
antes de  
actualizar  
los datos  
en disco

**Fallida**

Tras descubrir que no  
puede continuar

**Comprometida**

REINICIAR: Si el fallo no  
depende de la lógica de la  
transacción

CANCELAR: Fallo depende  
de la lógica de la transacción

**Abortada**

Después de haber sido  
retrocedida. La BD tiene  
el estado anterior.

Información en  
disco suficiente  
para reproducir  
los cambios  
después de un  
fallo. **REGISTRO**

Tras completarse  
correctamente

NO SE PUEDE  
DESHACER, solo  
se puede hacer  
otra consulta que  
revierta ese  
estado

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

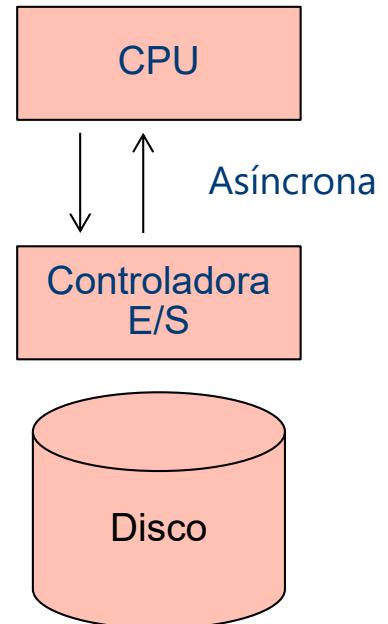
Etc.

## ■ Aislamiento

- ▷ Ejecución concurrente de varias transacciones puede generar inconsistencias
- ▷ Razones para no limitarse a la ejecución secuencial
  - Mejora del rendimiento y uso de los recursos
    - Ejecución paralela de instrucciones E/S y CPU
      - Disminuye **tiempo total de ejecución**, aumenta uso CPU y E/S
    - Reducción **tiempo medio de respuesta**
      - Transacciones cortas no tienen que esperar a que termine una larga
  - ▷ Necesarios planificadores que garanticen aislamiento

## ■ Aislamiento y Atomicidad

- ▷ ¿Qué ocurre si falla una transacción durante la ejecución concurrente de varias?
- ▷ Si T falla, es necesario deshacer sus efectos (**atomicidad**)
  - Cada  $T_i$ , que ha leído datos escritos por T debe deshacerse también!
    - Abortar transacciones que no han fallado
    - Si se han comprometido ya, nos e pueden abortar!



# Transacciones

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Niveles de aislamiento en el estándar SQL

los niveles en los que se puede colocar el cliente

### ▷ Secuenciable

mas garantías, –  
mas lento

Normalmente asegura la ejecución secuenciable de la transacción

- Algunos SGBDs utilizan protocolos que no lo garantizan en todos los casos

### ▷ Lectura repetible

- Sólo se pueden leer datos comprometidos (escritos por transacciones comprometidas)
- Entre dos lecturas del mismo dato en una transacción, otra no puede modificarlo
- No garantiza la secuencialidad

### ▷ Lectura comprometida

- Sólo permite leer datos comprometidos
- **NO** garantiza lecturas repetibles

### ▷ Lectura no comprometida

- menos garantía,  
mas rápido
- Permite leer datos no comprometidos
  - Nunca escrituras sucias
    - Escribir sobre un dato escrito por otra transacción no comprometida

TI	TJ
<pre>SELECT count(*) FROM empleado WHERE dept = 'Ventas'</pre>	<pre>INSERT INTO empleado (1, 'Juán', 23k, 'Ventas')</pre>

Las instrucciones de TI y TJ no deberían de tener conflicto ya que no actúan sobre los mismos elementos de datos

Sin embargo el orden de ejecución importa

HAI CONFLICTO

**Fenómeno  
Fantasma**

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Protocolos de implementación del aislamiento

### ▷ Estrategias Pesimistas

- Toman medidas que fuerzan esperas o retrocesos de transacciones como prevención
- Basados en bloqueos
  - Protocolo de bloqueo en dos fases
- Basados en marcas de tiempo

### ▷ Estrategias optimistas

- Permiten la ejecución normal y solo al final comprueban si hay problemas
- Basados en validación
- Basados en versiones múltiples
  - **Aislamiento de instantáneas**

este lo utiliza PostgreSQL

No garantiza la secuencialidad

Pueden ser necesarias medidas adicionales

Cláusula **FOR UPDATE**  
(trata las lecturas como escrituras)

# Otras Funcionalidades

Modelo

Lenguajes  
Formales

Sintaxis básica

Nulos

Agregación

Subconsultas

Oper. conjunto

Joins

Transacciones

Etc.

## ■ Vistas

```
CREATE VIEW nombre_vista AS  
SELECT ...  
FROM ...;
```

permite decir que columnas se ven y cuales no.  
En general, ocultar la complejidad de la base  
de datos y dar a cada usuario una vista

## ■ Creación de índices

estos indices aceleran (el join es la funcion mas compleja mas lenta)

## ■ Tipos, funciones, procedimientos definidos por el usuario

## ■ Disparadores

## ■ Seguridad

## ■ Acceso a la Base de datos desde un lenguaje de programación

## ■ Consultas Recursivas

## ■ Agregación avanzada y OLAP (... Inteligencia de Negocio)



# Bases de datos relacionales y SQL

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

# Bases de datos Objeto-Relacionales

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

- **Introducción**
- **Estructuras de datos complejas**
- **Referencias**
- **Herencia**
- **Representaciones abiertas**
  - ▷ **XML**
  - ▷ **JSON**

## Introducción

Estr. de datos complejas

Referencias

Herencia

XML

JSON

### ■ Objetivo

- ▷ Extender el modelo relacional para incorporar las características deseables de la orientación a objetos
- ▷ Conservar el modelo compatible con el modelo relacional
  - No perder 35 años de investigación en el modelo relacional
    - Con un **fundamento teórico muy sólido**

### ■ Generaciones de SGBDs

ya que la mayoría de las soluciones funcionan con el modelo relacional

- ▷ **Pre-relacionales**
  - Modelos jerárquicos y Modelos en red
- ▷ **Relacionales**
  - Modelo relacional y Lenguaje SQL
- ▷ **Objeto-relacionales (finales de los 90)**
  - Extensión del modelo relacional
    - Tipos de dato: Estructuras de datos más complejas y operaciones
    - Incorporación de objetos: tablas con tipo como clases de objetos (referencias)
  - Extensión del lenguaje SQL
    - A partir del SQL:1999 y SQL:2003
      - Incorpora otras características no O-R (Recursividad)

esto es lo que añade

en aplicaciones se utilizan muy poco estas nuevas características



Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos de dato SQL objeto-relacional

### ▷ Tipos incorporados (Built-in)

- \_ Boolean, Char, Varchar, Numeric, BLOB, CLOB, Date, Time, Timestamp, Interval, etc.
- \_ 10 categorías de tipos con tipado fuerte en cada categoría

### ▷ Constructores de tipo

- \_ Definición de datos

```
CREATE TABLE departamento (
    id_dep INTEGER PRIMARY KEY,
    nombre VARCHAR(50),
    direccion ROW(calle VARHCAR(100), num INTEGER, loc VARCHAR(10),
    presupuesto DECIMAL(12, 2),
);
```

```
CREATE TABLE empleado (
    id_emp INTEGER PRIMARY KEY,
    nombre VARCHAR(50),
    direccion ROW(calle VARHCAR(100), num INTEGER, loc VARCHAR(10),
    salario DECIMAL(9, 2),
    hijos VARCHAR(50) ARRAY,
    cursos ROW(nombre VARCHAR(50), nota DECIMAL(3, 1)) MULTISET
    dep INTEGER,
    FOREIGN KEY dep REFERENCES departamento(id_dep)
);
```

el array tiene orden (podemos acceder al primero al segundo etc) el multiset no

multiset es un conjunto que permite repetidos

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos de dato

### ▷ Constructores de tipo

#### – Consultas

##### Desanidar

```
SELECT e.nombre, h.hijo
FROM empleado AS e, UNNEST (e.hijos) AS h(hijo)
```

```
SELECT e.nombre, AVG(c.nota)
FROM empleado AS e, UNNEST (e.cursos) AS c(nota)
WHERE e.direccion.loc = 'Santiago' and c.nota >= 5
GROUP BY e.nombre
```

##### Anidar

```
SELECT d.nombre, d.presupuesto,
MULTISET(SELECT e2.nombre, e2.salario, count(h.hijo) AS Hijos
        FROM empleado AS e2, UNNEST (e2.hijos) AS h(hijo)
        WHERE d.id_dep = e2.dep
        GROUP BY e2.nombre, e2.salario),
SUM(e.salario) AS SalarioTotal
FROM departamento AS d, empleado AS e
WHERE d.id_dep = e2.dep
GROUP BY d.nombre, d.presupuesto
HAVING d.presupuesto > SUM(e.salario)
```

```
CREATE TABLE departamento (
id_dep INTEGER PRIMARY KEY,
nombre VARCHAR(50),
direccion ROW(calle VARCHAR(100), num INTEGER, loc VARCHAR(10),
presupuesto DECIMAL(12, 2),
);
```

```
CREATE TABLE empleado (
id_emp INTEGER PRIMARY KEY,
nombre VARCHAR(50),
direccion ROW(calle VARCHAR(100), num INTEGER, loc VARCHAR(10),
salario DECIMAL(9, 2),
hijos VARCHAR(50) ARRAY,
cursos ROW(nombre VARCHAR(50), nota DECIMAL(3, 1)) MULTISET
dep INTEGER,
FOREIGN KEY dep REFERENCES departamento(id_dep)
);
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos de dato

### ▷ Tipos estructurados

#### – Definición de las estructuras de datos

```
CREATE TYPE direccion AS (
    calle VARCHAR (50),
    num INTEGER,
    loc VARCHAR (15)
) NOT FINAL;
```

```
CREATE TYPE empleado AS (
    nombre VARCHAR (50),
    dir DIRECCION,
    salario_base DECIMAL (9, 2),
    complementos DECIMAL (9,2)
) NOT FINAL;
```

```
CREATE TABLE departamento (
    nombre VARCHAR (50),
    dir DIRECCION,
    presupuesto DECIMAL(10,2),
    secretario EMPLEADO
);
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

- Tipos de dato
  - ▷ Tipos estructurados
    - Definición de los métodos

```
CREATE TYPE empleado AS (
    nombre VARCHAR (50),
    dir DIRECCION,
    salario_base DECIMAL (9, 2),
    complementos DECIMAL (9,2))
INSTANTIABLE NOT FINAL
INSTANCE METHOD salario()
RETURNS DECIMAL(9, 2);
```

```
CREATE INSTANCE METHOD salario()
RETURNS DECIMAL (9, 2) FOR empleado
BEGIN
    RETURN SELF.salario_base +
        SELF.complementos
END
```

```
SELECT nombre, secretario.salario()
FROM Departamento
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

- Tipos de dato
  - ▷ Tipos estructurados
    - Definición de los métodos (Constructores)

```
CREATE TYPE circulo AS (
    radio DECIMAL(5, 1))
INSTANTIABLE NOT FINAL
CONSTRUCTOR METHOD circulo(
    radio DECIMAL(5, 1))
RETURNS circulo
SELF AS RESULT;
```

```
CREATE METHOD circulo(
    radio DECIMAL(5, 1))
RETURNS circulo FOR circulo
BEGIN
    SET SELF.radio = radio
    RETURN SELF
END
```

```
INSERT INTO circulos
VALUES (3, NEW circulo(5));
```

# Estructuras de datos complejas

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos de dato

### ▷ Tipos estructurados

#### – Definición de los métodos (**Observers** y **Mutators**)

nombre(): metodo que devuelve el objeto

```
CREATE TYPE empleado AS (
    nombre VARCHAR (50),
    dir DIRECCION,
    salario_base DECIMAL (9, 2),
    complementos DECIMAL (9,2))
INSTANTIABLE NOT FINAL
INSTANCE METHOD salario()
RETURNS DECIMAL(9, 2);
```

```
SELECT secretario.nombre()
FROM Departamento
WHERE secretario.salario_base > 34
```

```
UPDATE Departamento
SET secretario.nombre = 'Juan'
WHERE secretario.salario() < 123
```

```
INSERT INTO Departamento (nombre,
secretario)
VALUES (
'Ventas',
NEW empleado().nombre('Juan')
)
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos de dato

### ▷ Tipos estructurados

#### – Definición de columnas de tablas y atributos de otros tipos

##### Definición de datos

“paquetes” con las funciones predefinidas, algunos libres y otros de pago

```
CREATE TYPE punto AS (
    x FLOAT, y FLOAT)
```

```
CREATE TYPE rectangulo AS (
    x1 FLOAT, y1 FLOAT,
    x2 FLOAT, y2 FLOAT)
```

```
CREATE FUNCTION distancia (g1 geo, g2 geo) RETURNS FLOAT
    BEGIN ... END
```

```
CREATE TYPE linea AS (
    mbr rectangulo,
    coords punto ARRAY)
INSTANCE METHOD longitud()
RETURNS FLOAT
```

```
CREATE TYPE poligono AS (
    mbr rectangulo,
    coords punto ARRAY)
INSTANCE METHOD area()
RETURNS FLOAT
```

##### Consultas

```
SELECT sum(trazado.longitud)
FROM carreteras
WHERE propietario = 'Municipal' and estado = 'Malo'
```

```
SELECT sum(camas)
FROM carreteras AS c, centros_salud AS cs
WHERE c.estado = 'bueno' and c.longitud < 2000
and distancia(c.trazado, cs.posicion) < 1000
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos de dato

### ▷ Tipos estructurados

#### - Definición de los tipos de las filas de una tabla (Tablas con tipo)

no hay acuerdo entre si es bueno o malo: aquí se reutiliza un tipo para generar una nueva tabla

oid - identificador del objeto

```
CREATE TYPE direccion AS (
    calle VARCHAR (50),
    num INTEGER,
    loc VARCHAR (15)) NOT FINAL;
```

REF USING INTEGER  
REF FROM (id\_emp)

```
CREATE TYPE empleado AS (
    id_emp VARCHAR (8),
    nombre VARCHAR (50),
    salario DECIMAL (9, 2),
    dir DIRECCION)
NOT FINAL
REF IS SYSTEM GENERATED;
```

Opción por defecto

```
CREATE TABLE empleados
OF empleado
(REF IS oid SYSTEM GENERATED)
```

REF IS oid USER GENERATED  
REF IS oid DERIVED

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos Referencia

### ▷ Definición de datos

se puede usar el oid en otra tabla para referenciar a ese objeto  
dep REF es una referencia a departamento, ahí se almacena un oid

```
CREATE TYPE Empleado AS (
    nombre VARCHAR (50),
    salario_base DECIMAL (9, 2),
    complementos DECIMAL(9, 2),
    dep REF(Departamento))
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD salario()
RETURNS DECIMAL(9, 2);
```

```
CREATE TABLE Empleados
OF Empleado
(REF IS oid SYSTEM GENERATED,
dept WITH OPTIONS SCOPE Departamentos);
```

```
CREATE TYPE Departamento AS (
    nombre VARCHAR (50),
    dir DIRECCION,
    emps REF(Empleado) MULTISET,
    director REF(Empleado))
NOT FINAL
REF IS SYSTEM GENERATED;
```

```
CREATE TABLE Departamentos
OF Departamento
(REF IS oid SYSTEM GENERATED,
emps WITH OPTIONS SCOPE Empleados,
director WITH OPTIONS SCOPE Empleados);
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Tipos Referencia

### ▷ Derreferenciación de atributos

```
SELECT e.nombre, e.dept->nombre
FROM Empleados e
WHERE e.dep->dir.loc = 'Santiago'
```

esos oid son basicamente punteros

```
SELECT e.nombre, DEREF(e.dept).nombre
FROM Empleados e
WHERE DEREF(e.dept).dir.loc = 'Santiago'
```

```
SELECT e.nombre, (SELECT d.nombre FROM departamentos d WHERE d.oid = e.dep)
FROM Empleados e
WHERE (SELECT dir.loc FROM departamentos d WHERE d.oid = e.dep) = 'Santiago'
```

```
SELECT e.nombre, d.nombre
FROM Empleados e LEFT JOIN Departamentos d ON (e.dep = d.oid)
WHERE d.dir = 'Santiago'
```

en principio no tiene beneficios esta forma, tienen rendimientos parecidos cualquiera de estas 4

### ▷ Derreferenciación de métodos

```
SELECT d.nombre, SUM(e.ptr->salario)
FROM Departamentos d, UNNEST(d.emps) AS e(ptr)
GROUP BY d.nombre
```

```
SELECT e.nombre, e.salario
FROM Empleados e
```

```
SELECT e.nombre, DEREF(e.oid).salario
FROM Empleados e
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

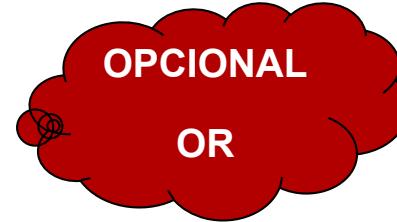
## ■ Herencia entre tipos estructurados

### ▷ Características

#### - Herencia simple

- Un valor solo tiene un tipo más específico (subtipos disjuntos)

puedes declarar subtipos de un tipo



OPCIONAL  
OR

```
CREATE TYPE rectangulo AS (
    x1 FLOAT, y1 FLOAT, x2 FLOAT, y2 FLOAT)
NOT FINAL
INSTANCE METHOD area() RETURNS FLOAT;
```

```
CREATE TYPE cuadrado
UNDER rectangulo (tamano FLOAT)
NOT FINAL
OVERRIDING METHOD area() RETURNS FLOAT;
```

```
CREATE TABLE rectangulos AS (
    id_rect INTEGER PRIMARY KEY,
    rec rectangulo)
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Herencia entre tipos estructurados

- ▷ Sobrecarga de operadores se pueden hacer cosas como esta

```
CREATE TYPE geom AS (
    mbr rectangulo)
NOT INSTANTIABLE NOT FINAL;
```

```
CREATE TYPE poligono UNDER geom (
    coords FLOAT ARRAY)
INSTANTIABLE NOT FINAL
INSTANCE METHOD area() RETURNS FLOAT;
```

```
CREATE TYPE rectangulo UNDER geom (
    x1 FLOAT, y1 FLOAT, x2 FLOAT, y2 FLOAT)
INSTANTIABLE NOT FINAL
OVERRIDING METHOD area() RETURNS FLOAT;
```

```
CREATE METHOD area() FOR rectangulo
BEGIN
    RETURN (SELF.x2 - SELF.x1) *
        (SELF.y2 - SELF.y1);
END
```

```
CREATE TYPE cuadrado
UNDER rectangulo (tamano FLOAT)
INSTANTIABLE NOT FINAL
OVERRIDING METHOD area() RETURNS FLOAT;
```

```
CREATE METHOD area()
FOR cuadrado
BEGIN
    RETURN tamano * tamano;
END
```

Introducción

Estr. de datos  
complejas

Referencias

**Herencia**

XML

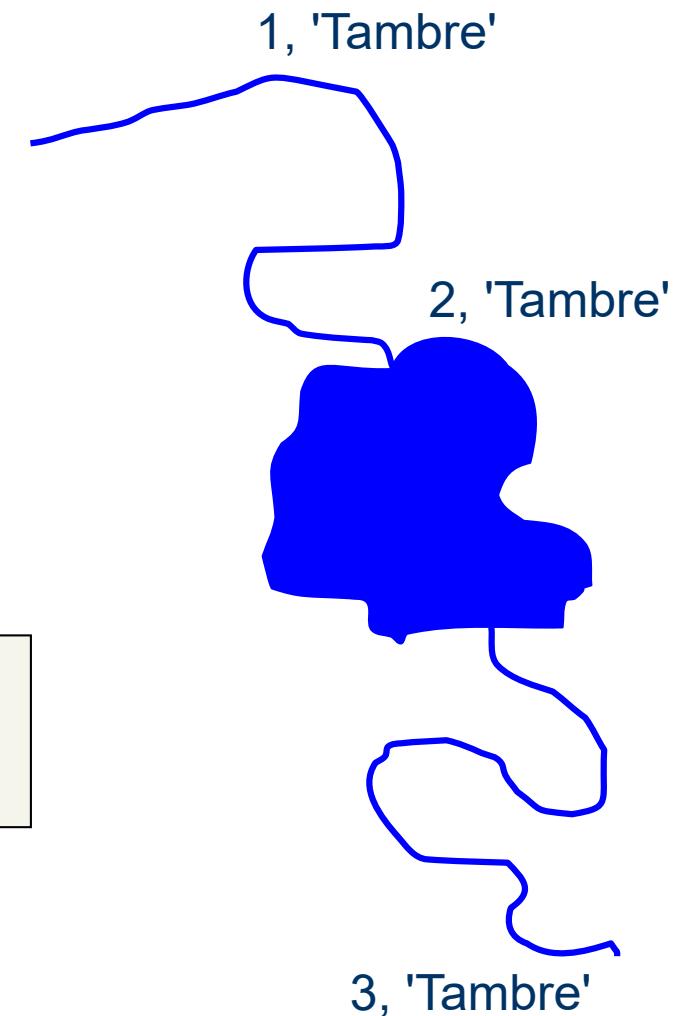
JSON

## ■ Herencia entre tipos estructurados

- ▷ Uso de un tipo concreto

```
CREATE TABLE rios AS (
    id_rio INTEGER PRIMARY KEY,
    nombre VARCHAR(50),
    geom geo)
```

```
SELECT nombre, SUM(TREAT(geom AS linea).longitud)
FROM rios
WHERE geom IS OF (linea)
GROUP BY nombre
```



Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Herencia entre tipos estructurados

### ▷ Herencia entre tablas

#### - Motivación



**RELACIONAL**

**EMPLEADOS**

id_emp	nombre	salario	dep
1	Juan	1500	3
2	Elisa	2100	5

**DEPARTAMENTOS**

id_dep	nombre	presup
3	Ventas	345466
5	Producción	216500

**ADMINISTRADORES**

id_emp	Sistema
2	Linux

**PROGRAMADORES**

id_emp	Lenguaje
1	Pascal

**PROG-PROY**

id_emp	id_proy
1	23

**PROYECTOS**

id_proy	duracion	presup	dep
23	2	4643	3
34	3	2234	3

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Herencia entre tipos estructurados

### ▷ Herencia entre tablas

#### – Definición de datos

```
CREATE TYPE empleado AS (
    nombre VARCHAR(50),
    salario DECIMAL(6, 2),
    dep REF(departamento)
) INSTANTIABLE NOT FINAL
```

```
CREATE TYPE administrador UNDER
empleado (
    sistema VARCHAR(20))
INSTANTIABLE NOT FINAL
```

```
CREATE TYPE programador
UNDER empleado (
    lenguaje VARCHAR(20),
    proys REF(proyecto) MULTISET)
INSTANTIABLE NOT FINAL
```

```
CREATE TYPE departamento AS (
    nombre VARCHAR(20),
    presup DECIMAL(9, 2),
    emps REF(empleado) MULTISET,
    proys REF(proyecto) MULTISET)
INSTANTIABLE NOT FINAL
```

no se usa mucho en la practica

```
CREATE TYPE proyecto AS (
    duracion INTEGER,
    presup DECIMAL(9, 2),
    dep REF(departamento),
    progs REF(programador) MULTISET)
INSTANTIABLE NOT FINAL
```

```
CREATE TABLE EMPLEADOS OF empleado
(REF IS id_emp SYSTEM GENERATED,
dep WITH OPTIONS SCOPE DEPARTAMENTOS);
```

```
CREATE TABLE ADMINISTRADORES OF administrador
UNDER EMPLEADOS;
```

```
CREATE TABLE PROGRAMADORES OF programador
UNDER EMPLEADOS
(proys WITH OPTIONS SCOPE PROYECTOS);
```

```
CREATE TABLE DEPARTAMENTOS OF departamento
(REF IS id_dep SYSTEM GENERATED,
emps WITH OPTIONS SCOPE EMPLEADOS,
proys WITH OPTIONS SCOPE PROYECTOS);
```

```
CREATE TABLE PROYECTOS OF proyecto
(REF IS id_proy SYSTEM GENERATED,
dep WITH OPTIONS SCOPE DEPARTAMENTOS,
progs WITH OPTIONS SCOPE PROGRAMADORES);
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Herencia entre tipos estructurados

### ▷ Herencia entre tablas

#### - Consultas

- Al seleccionar en subtablas, se realiza automáticamente el **join** con la supertabla.

```
SELECT *  
FROM Empleados
```

```
SELECT p.nombre, p.lenguaje  
FROM Programadores
```

```
SELECT *  
FROM ONLY (Empleados)
```

```
SELECT *  
FROM Empleados  
WHERE DEREF(oid) IS OF (Empleado)
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON



## ■ Herencia entre tipos estructurados

### ▷ Herencia entre tablas

utilizar herencia tiene consecuencias en estas tres cosas:

- **Inserciones, modificaciones y borrados**

- Inserción

- Se permiten inserciones en las supertablas (**herencia opcional**)
- Inserción en la subtabla inserta automáticamente en la supertabla

- Borrado

- Siempre se realizan borrados **en cascada de forma automática**

se borra de todos lados donde este

- Modificación

- Modificación en la supertabla se puede hacer a través de la subtabla.

```
INSERT INTO PROGRAMADORES
(nombre, salario, lenguaje)
VALUES ('Felipe', 2450, 'Java')
```

```
DELETE FROM EMPLEADOS
WHERE nombre = 'Felipe'
```

```
DELETE FROM PROGRAMADORES
WHERE nombre = 'Felipe'
```

```
UPDATE ADMINISTRADORES
SET salario = salario + salario*0.25
WHERE dept->nombre = 'Redes'
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Extensible Markup Language (XML)

- ▷ Metalenguaje: Reglas para definir lenguajes (cada uno con su vocabulario)
- ▷ Combina datos con etiquetas (Metadatos) mezcla texto y metadatos
- ▷ Para humanos y máquinas

Declaración  
(Opcional)

Comentario

```
<?xml version="1.0"?>
<!-- Esta es una representación XML de la tabla de empleados --&gt;
&lt;Empleados&gt;
  &lt;Empleado&gt;
    &lt;Nombre&gt;Alberto&lt;/Nombre&gt;
    &lt;DNI&gt;34233456-D&lt;/DNI&gt;
    &lt;Edad&gt;35&lt;/Edad&gt;
    &lt;Sueldo Moneda ="Euro"&gt;1200&lt;/Sueldo&gt;
  &lt;/Empleado&gt;      1200 en el dato, y lo que hay entre &lt;&gt; son los metadatos del mismo
  &lt;Empleado&gt;
    &lt;Nombre&gt;Inés&lt;/Nombre&gt;
    &lt;DNI&gt;31245659-D&lt;/DNI&gt;
    &lt;Edad&gt;29&lt;/Edad&gt;
    &lt;Sueldo Moneda ="Peseta"&gt;180000&lt;/Sueldo&gt;
  &lt;Empleado&gt;
&lt;/Empleados&gt;</pre>
```

Atributo

Elemento

Introducción

Estr. de datos complejas

Referencias

Herencia

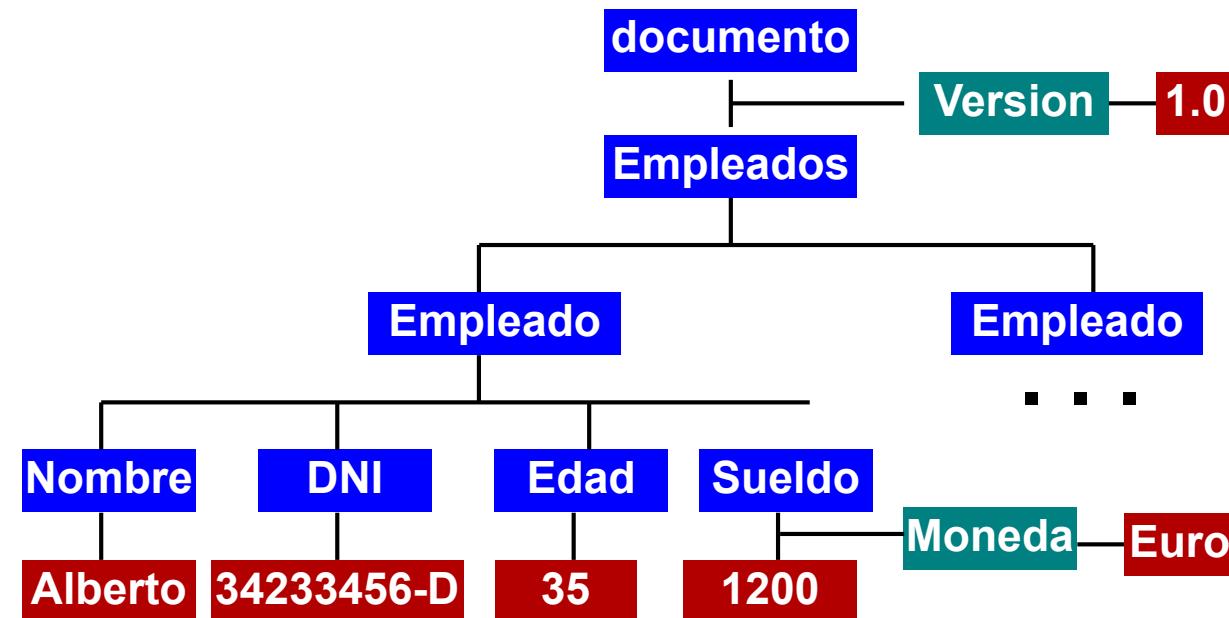
XML

JSON

## ■ Extensible Markup Language (XML)

### ▷ Estructura **Jerárquica**

- \_ Raíz o nodo documento
- \_ Elemento raíz
- \_ Secuencia de hijos
- \_ Hojas
  - caracteres, atributos, comentarios, etc.



Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ Extensible Markup Language (XML)

- ▷ Document Type Definition (DTD) se puede (o no) definir esquema
  - Documentos bien formados: 1 raíz, anidamiento correcto, etc.
  - Documentos válidos: bien formados + cumplen un DTD

DTD define el  
vocabulario del  
lenguaje

Orden importa

```
<?xml version="1.0"?>
<!-- Esta es una representación
     XML de la tabla de empleados
--&gt;
&lt;Empleados&gt;
  &lt;Empleado&gt;
    &lt;Nombre&gt;Alberto&lt;/Nombre&gt;
    &lt;DNI&gt;34233456-D&lt;/DNI&gt;
    &lt;Edad&gt;35&lt;/Edad&gt;
    &lt;Sueldo Moneda ="Euro"&gt;
      1200
    &lt;/Sueldo&gt;
  &lt;/Empleado&gt;
  &lt;Empleado&gt;
    &lt;Nombre&gt;Inés&lt;/Nombre&gt;
    &lt;DNI&gt;31245659-D&lt;/DNI&gt;
    &lt;Edad&gt;29&lt;/Edad&gt;
    &lt;Sueldo Moneda ="Peseta"&gt;
      180000
    &lt;/Sueldo&gt;
  &lt;/Empleado&gt;
&lt;/Empleados&gt;</pre>

```

```
<!ELEMENT Empleados (Nota?, Empleado*)>
<!ELEMENT Empleado (Nombre, DNI, Edad, Sueldo)>
<!ELEMENT Nombre (#PCDATA)>
<!ELEMENT DNI (#PCDATA)>
<!ELEMENT Edad (#PCDATA)>
<!ELEMENT Sueldo (#PCDATA)>
<!ATTLIST Sueldo
  Moneda (Euro | Peseta) #REQUIRED>
```

Parsed Character Data

**Cardinalidad :**

- ? Opcional
- \* Cero o más
- + uno o más
- Por defecto, Uno

**Atributos:**

Por defecto opcionales

<!ATTLIST Sueldo Moneda>

Valor por defecto

<!ATTLIST Sueldo
 Moneda
 (Euro | Peseta) Euro>

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON

## ■ XML Schema

- ▷ Es un lenguaje XML (no como el DTD)
- ▷ Mucho más expresivo

<http://www.w3.org/2001/XMLSchema>

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Schema para Empleados-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              targetNamespace="http://www.usc.es/ribdd/empleados"
              xmlns="http://www.usc.es/ribdd/empleados"
              elementFormDefault="qualified">

  <xsd:element name="Empleados">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Empleado"
                     type="tipoEmpleado"
                     minOccurs="0"
                     maxOccurs = "unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="tipoEmpleado">
    <xsd:sequence>
      <xsd:element name="Nombre" type="xsd:string"/>
      <xsd:element name="DNI" type="tipoDNI"/>
      <xsd:element name="Edad" type="xsd:integer"/>
      <xsd:element name="Sueldo" type="tipoSueldo"/>
    </xsd:sequence>
  </xsd:complexType>
```

complexType  
element simpleContent  
Schema sequence

<http://www.empleados.es>

```
<xsd:simpleType name="tipoDNI">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{8}-[a-z]{1}"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="tipoSueldo">
  <xsd:simpleContent>
    <xsd:extension base="xsd:decimal">
      <xsd:attribute name="Moneda"
                     type="tipoMoneda"
                     use="required"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:simpleType Name="tipoMoneda">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Euro"/>
    <xsd:enumeration value="Peseta"/>
  </xsd:restriction>
</xsd:simpleType>
```

Empleados  
Nota DNI Empleado  
Suelo Moneda  
Edad

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON



## ■ Lenguajes de búsqueda y consulta

▷ XPath no es un lenguaje cerrado

- Expresiones formadas por conjuntos de pasos
  - /paso/paso/.../paso
- Cada paso tres componentes: **axisName::nodetest[predicate]**
  - axisName: dirección de navegación (child, parent, descendant, attribute, etc.)
    - child es la opción por defecto
  - nodetest: tipo de nodo y etiqueta del nodo
    - child::\* (cualquier hijo), attribute::\* (cualquier atributo), child::text() (hijos de tipo texto)
    - / (nodo raíz), // (cualquier nodo), . (nodo actual), .. (nodo padre), @ (atributo)
  - Predicate: Número de posición o expresión de tipo booleano
- Ejemplos
  - /Empleados/Empleado[1]/Nombre/text()
  - /Empleados/Empleado[Edad>34]/Nombre
  - //Empleado[@Sueldo>120]

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

## ■ Lenguajes de búsqueda y consulta

- ▷ XQuery para consultas parecido a SQL

```
for $variable1 at $pos in doc("localizacion del documento") /xpath  
let $variable2 := valor  
where condición  
order by expresión, expresión descending, expresión ascending  
return expresión
```

JSON



```
<ul>  
{  
for $e in doc("empleados.xml") /Empleados/Empleado  
where $e/Edad/data() > 58  
order by $e/@Sueldo descending  
return <li>{$e/Nombre/text()}</li>  
}  
</ul>
```

Introducción

Estr. de datos  
complejas

Referencias

Herencia

**XML**

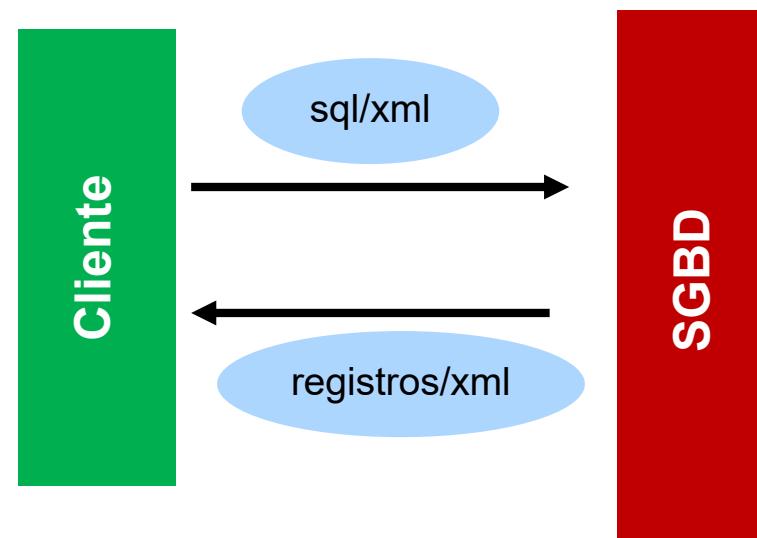
JSON

## ■ Soporte en sistemas de bases de datos

- ▷ Bases de datos XML nativas (XPath / XQuery)
- ▷ Soporte XML en ISO SQL
  - Tipo de datos XML
  - Métodos para crear XML a partir de datos relacionales
    - Creación de elementos
    - Concatenación de partes de documentos
    - Agregación de elementos
    - Etc.
  - Métodos para acceder a partes de un documento XML (XPath, XQuery)

el como se anide el modelo implica que habra consultas más rápidas y más lentas!!!! (esto respeto al objeto-relacional)

el modelo que tiene detras XML es jerarquico



## Departamento

<b>id</b>	<b>Nombre</b>	<b>Empleados</b>
d1	Ventas	<Empleados...
d2	Publicidad	<Empleados...
d3	Infraestructuras	<Empleados...

Introducción

Estr. de datos  
complejas

Referencias

Herencia

XML

JSON



- **JavaScript Object Notation (JSON)** lenguaje jerarquico similar a xml
  - ▷ Formato abierto (texto) de intercambio de datos
  - ▷ Para humanos y máquinas en la app web de cliente se usa JSON, por eso se devuelve JSON ahora al no tener que estar transformando, puede agilizar el movimiento de datos
  - ▷ Subconjunto de JavaScript Programming Language Standard ECMA-262
  - ▷ Estructuras
    - Colección de pares nombre/valor (**Objeto**)
    - Lista ordenada de valores (**Array**)
  - ▷ Sintaxis muy simple
    - <https://www.json.org/json-es.html>
- **JSON Schema**
  - ▷ <https://json-schema.org/>
- **Soporte en sistemas de bases de datos**
  - ▷ Representación y modelo de datos para las bases de datos NoSQL de tipo documental más conocidas (MongoDB, CouchDB, ...)
  - ▷ Soporte en ISO SQL similar al proporcionado para XML

JSON esta en el estandar de SQL

# Bases de datos Objeto-Relacionales

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

# Bases de datos Distribuidas

no se utilizan mucho porque son caras

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

- **Introducción**
  - ▷ **Bases de datos homogéneas y heterogéneas**
- **Almacenamiento distribuido**
- **Transacciones distribuidas**
- **Protocolos de compromiso**
- **Control de concurrencia**
- **Disponibilidad**
- **Procesamiento distribuido de consultas**
- **Bases de datos distribuidas heterogéneas**

## Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Bases de datos **Distribuidas Vs Bases de datos Paralelas**

tienen que ver con paralelizar las consultas, para ello, ...

- ▷ Paralelas: Procesadores pueden estar muy acoplados, dentro del mismo SGBDs
- ▷ Distribuidas: Nodos débilmente acoplados (no comparte componentes hardware)

cada nodo no comparte nada más que la red (habitualmente), todos juntos se comportan como un único gestor  
“se paralleliza el acceso a disco”

## ■ Distribución de los datos

- ▷ Necesaria para mejorar (**Volumen, Velocidad**)

– Rendimiento

cuando se mejora una se cae la otra (como norma general)

se mejoran

– Disponibilidad que este funcionando siempre

- consistencia: si tengo varias copias, que sean iguales

- ▷ Principal causa de las dificultades de su implementación

si tengo muchas copias, si se cae una tengo otra disponible para consultar, pero cuantas mas copias tenga mas posibilidad de inconsistencia

## Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

### ■ Bases de datos Homogéneas Vs Bases de datos Heterogéneas

existen dos tipos

#### ▷ Bases de datos Homogéneas

- Sistema Gestor de Bases de datos **idéntico** en cada nodo
- Cada nodo es consciente de la existencia de los demás
- Todos los nodos juntos se comportan como un único sistema, y con un esquema de datos único que **se distribuye**.  
transparencia

En general asumiremos el caso de BDs homogéneas

#### ▷ Bases de datos Heterogéneas (Variedad) sirve para atacar la variedad. No son tan tipicas

- Cada nodo puede tener un software de acceso a datos distinto
- Cada nodo puede tener un esquema distinto en sus datos.
- Los nodos pueden no conocer la existencia de los demás
- Los nodo deben de poder operar con total independencia para atender a sus usuarios locales
- Se proporciona algún tipo limitado de cooperación entre los nodos
  - Ejemplo: Consulta integrada
    - Dificultad en el procesamiento de consultas debido a la existencia de esquemas distintos.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Dos soluciones para almacenar una relación de forma distribuida

### ▷ Replicación

- Varias copias de la misma tabla en varios nodos

### ▷ Fragmentación

- Dividir la tabla en pedazos y almacenar cada pedazo en un nodo.

### ▷ Se puede (suele) combinar la replicación y la fragmentación

- Cada fragmento se puede replicar en varios nodos

puede ser vertical u horizontal. Se puede almacenar unas filas en un nodo y otras en otro, o columnas. Ahora existe tambien el sharding, que fragmentación pero solo por filas !!

# Almacenamiento Distribuido

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Replicación de Datos

se recomienda minimo 3 copias

- ▷ Cada relación almacenada en dos o más nodos.

replicacion me da mayor disponibilidad, aumenta el paralelismo

- ▷ Ventajas y Desventajas

– Disponibilidad ↑ (en lectura)

crea problemas de inconsistencia (si lo pides

antes de que se actualicen todas, ya que puede darte la lectura de una copia aun no actualizada o genera

problemas de disponibilidad (en escritura) (esto ultimo si le pides inconsistencia, ya que espera a que se actualicen todas las tablas

y, si falla una, da error)

▪ Aumenta. Si un nodo falla, los mismos datos pueden obtenerse de otro.

Paralelismo ↑

▪ Aumenta. Varias transacciones de lectura pueden acceder a la misma tabla en paralelo en distinto nodos.

▪ Aumenta también la probabilidad de tener los datos en local (mismo nodo donde se ejecuta la transacción)

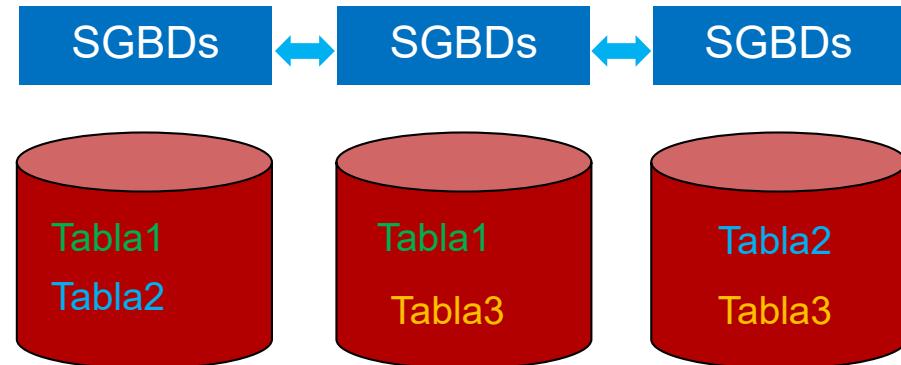
buena para leer pero mala para escribir: hay que actualizar todas las copias

Sobrecarga en las modificaciones ↑

- ▷ Buena para las lecturas, mala para las actualizaciones

es importante ver si mi aplicacion se beneficia o no

- ▷ Elegir una réplica como **primaria** simplifica la gestión.



Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Fragmentación de Datos

- ▷ Fragmentación horizontal sharding
  - Cada fila se envía a una partición.
  - La tabla se reconstruye usando la Unión
- ▷ Fragmentación vertical
  - Cada columna a una tabla, de manera que se pueda reconstruir usando un Join Natural
- ▷ Se pueden combinar

para que el sistema realmente escale, habría que aplicar la fragmentación horizontal, ya que las tablas aumentan por filas

**horizontal**

SGBDs

DNI	Nome	Salario	Coste_hora
23456238	Alfredo	35000	45
25368964	Sofía	43000	60
78878965	Elena	40500	55

SGBDs

DNI	Nome	Salario	Coste_hora
58325647	Ricardo	29500	30
78532564	Ernesto	41000	56

DNI	Nome	Salario	Coste_hora
23456238	Alfredo	35000	45
25368964	Sofía	43000	60
58325647	Ricardo	29500	30
78878965	Elena	40500	55
78532564	Ernesto	41000	56

**Vertical**

SGBDs

DNI	Nome	Salario
23456238	Alfredo	35000
25368964	Sofía	43000
58325647	Ricardo	29500

SGBDs

DNI	Coste_hora
23456238	45
25368964	60
58325647	30
78878965	55
78532564	56

Bases de datos distribuidas

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ **Transparencia** lo importante es tener transparencia

- ▷ **De fragmentación y de replicación**
  - Los usuarios no necesitan saber como ha sido particionada o replicada una tabla para poder trabajar con ella.
- ▷ **De localización**
  - Los usuario no necesitan saber donde están físicamente almacenados los datos. El SGBDs debería de poder localizar los datos a través de sus identificadores.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

- Tipos de transacciones que el sistema debe soportar
  - ▷ Transacciones locales
    - Acceden y modifican datos solo en la base de datos local (la que recibe la transacción)
    - ACID se garantiza con las técnicas de bases de datos centralizadas.  
aprticionamiento importante → sharding
  - ▷ Transacciones globales
    - Acceden y modifican datos en varias bases de datos
    - Garantizar ACID es mucho más complicado
      - Posibles fallos en cada base de datos o en las comunicaciones

aqui se puede caer la red en vez de  
las maquinas, lo que causa mas  
problema que que se caigan las  
maquinas

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

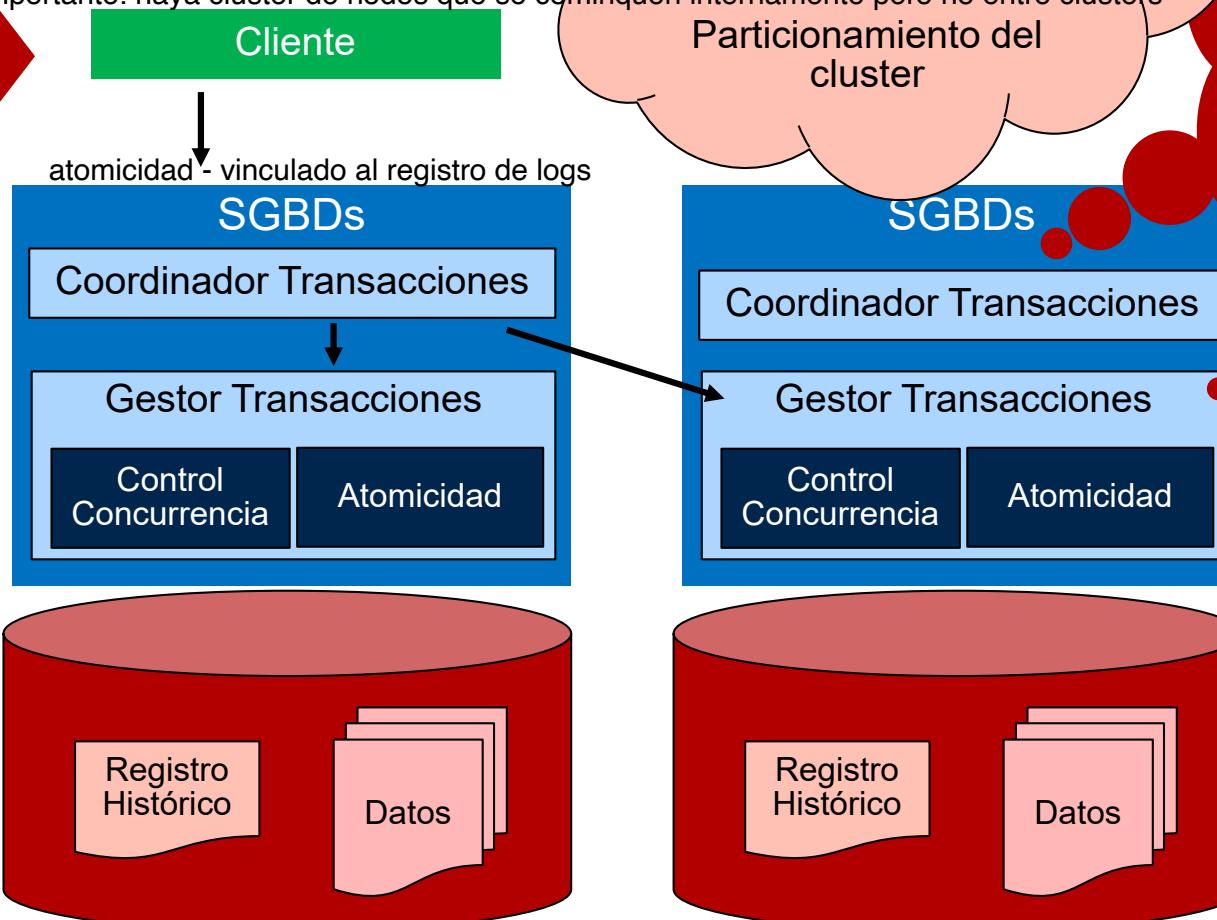
Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Estructura del sistema

importante: haya cluster de nodos que se comunique internamente pero no entre clusters



### Nuevos modos de fallo

(Fallos de nodos,  
pérdida de mensajes,  
Comunicaciones, etc.)

### Particionamiento del cluster

1.- Inicia la transacción

2.- Divide la transacción en sub-transacciones y las distribuye entre los SGBDs

3.- Coordina la terminación (compromiso o aborto)

Gestiona el registro histórico y el control de concurrencia

Se necesitan nuevas técnicas

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

lo que hace el sistema para terminar una transacción: decide si se aborta o no

## ■ Compromiso en arquitecturas centralizadas

- ▷ Se almacena un registro <Comprometida T> para la transacción T en el Registro Histórico
- ▷ El Registro Histórico se almacena en **almacenamiento estable**
  - \_ Implementado con varias copias para asegurar la durabilidad



## ■ Compromiso en arquitecturas distribuidas

- ▷ Todos los SGBDs deben de coordinarse para decidir si la transacción se compromete o se aborta
  - \_ No es aceptable un compromiso en algunos nodos y una cancelación en otros.
- ▷ Coordinador de Transacciones debe de ejecutar **un protocolo de compromiso**

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Protocolo de compromiso en dos fases

- ▷ La transacción **T** se inicia en un Nodo (**N**). **C** es el coordinador de ese nodo **N**.  
esto NO implica que finalice con éxito, todavía puede fallar, ya que aún no se ha volcado a disco
- ▷ Cuando **T** finaliza su ejecución, todos los nodos involucrados en su ejecución informan a **C** que **T** ha finalizado su ejecución. **C** inicia el protocolo.  
solo se han ejecutado las instrucciones, pero todavía no se ha comprometido  
Puede fallar aún.
- ▷ **Fase 1**
  - **C** añade el registro <**preparar T**> al registro histórico (en almacenamiento estable)
  - **C** envía el mensaje (**preparar T**) a todos los nodos involucrados.
  - Cada gestor de transacciones de cada nodo decide si quiere comprometer o abortar **T**.
    - Abortar: Almacena <**no T**> en el registro histórico y envía (**abortar T**) al coordinador
    - Comprometer: Almacena <**listo T**> en el registro histórico y envía (**listo T**) al coordinador

Introducción

Almacenam.

Transacciones

Compromiso



Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Protocolo de compromiso en dos fases

### ▷ Fase 2

- **C** decide si **T** se compromete o se aborta, después de esperar un tiempo por las respuestas de los nodo involucrados.
- **C** añade <**comprometer T**> o <**abortar T**> al registro histórico
  - Después de almacenar este registro, **T** está **comprometida o abortada**.  
aun no ha informado a otros nodos
    - Independientemente de lo que ocurra después.
- **C** envía mensajes (**comprometer T**) o (**abortar T**) a los nodos involucrados
- Cada nodo almacena <**comprometer T**> o <**abortar T**> en el registro histórico

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Protocolo de compromiso en dos fases

### ▷ Gestión de los fallos

#### – Fallos de un nodo participante

- Detección de los fallos por parte del coordinador
  - Antes de que el nodo envíe (**listo T**), se asume que **T** abortará
  - Despues de recibir (**listo T**), se asume que **T** se comprometerá.
- Despues de reiniciar despues del fallo, el nodo que falla examina el registro histórico
  - Si contiene **<comprometer T>**, ejecuta un **<rehacer T>**
  - Si contiene **<abortar T>**, ejecuta un **<deshacer T>**
  - Si contiene **<listo T>**, consulta al coordinador, o si el coordinador no responde a otros nodos.
  - En cualquier otro caso, ejecuta **<deshacer T>**

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Protocolo de compromiso en dos fases

### ▷ Gestión de los fallos

con el protocolo simple se sacrifica consistencia !!

#### – Fallos en el coordinador

- Si el coordinador **C** falla en medio de la ejecución del protocolo para **T**, los nodos participantes han de decidir lo que se hace con **T**.
  - En algunos casos no van a poder, y deberán esperar a que el coordinador **C** se recupere
- Si algún nodo tiene en el registro histórico **<comprometer T>**, entonces **T** debe ser comprometida
- Si algún nodo tiene en su registro histórico **<abortar T>**, entonces **T** debe de ser abortada
- Si algún nodo no tiene **<listo T>** en el registro histórico, debería de abortarse también.
- En los demás casos, todos los nodos tienen **<listo T>**, pero nada más. No se puede conocer la decisión del coordinador hasta que este se reinicie. **T** estará bloqueada hasta que **C** se recupere.

para solucionar esto, surge el protocolo en tres fases

Principal  
problema de  
este protocolo

Introducción

Almacenam.

Transacciones

**Compromiso**



Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Protocolo de compromiso en dos fases

### ▷ Gestión de los fallos

- **Particionamiento de la red** (nodos divididos en dos o mas particiones que no pueden comunicarse entre si)
  - Si todos los nodos participantes y **C** están en la misma partición, el fallo no tiene efecto sobre el protocolo de compromiso de **T**.
  - Si algún nodo involucrado o **C** está aislado de los demás, los nodos de una partición (y el coordinador) creen que los demás han fallado.
    - Los nodos que no tienen al coordinador en su partición ejecutan el protocolo para tratar el fallo del coordinador.
    - Los nodos que tienen al coordinador en su partición, y el propio coordinador, siguen con el protocolo asumiendo fallo de los demás nodos.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Protocolo de compromiso en dos fases

### ▷ Recuperación y control de concurrencia

- Proceso de recuperación debe de tratar los **casos dudosos**
  - <listo T> en el registro histórico, pero no está <comprometer T> o <abortar T>
- No se puede continuar con el procesamiento de transacciones hasta resolver estos casos dudosos
  - Proceso que puede ser lento o muy lento
  - Puede afectar a otras transacciones concurrentes con T
  - Necesidad de soluciones específicas para estos casos, que van a depender del protocolo de control de concurrencia utilizado.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

- Los protocolos de control de concurrencia utilizados necesitan adaptarse para funcionar en bases de datos distribuidas
  - ▷ Deben tener en cuenta la existencia de varias réplicas de cada elemento de datos.
  - ▷ Si el nodo de alguna réplica falla, ya no se pueden procesar modificaciones del elemento de datos.
    - Baja la disponibilidad
  - ▷ **Reto**
    - Continuar con el procesamiento de transacciones incluso si algunos de los nodos dejan de funcionar.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Replicación con bajos niveles de consistencia

### ▷ Replicación Maestro-esclavo

- Modificaciones en el nodo primario
  - Se propagan automáticamente hacia los secundarios
- Lectura desde cualquier nodo
- En las lecturas en nodos secundarios, los datos pueden no estar actualizados, pero sí deberían ser consistente (*transaction-consistent snapshot* de los datos del primario).  
contesta antes de actualizar
  - No podemos tener una versión de los datos en medio de una transacción
- Propagación de las modificaciones de primario: Inmediatamente o periódicamente.
- Se adapta muy bien a configuraciones con oficinas centrales y sucursales.
- Muy útil cuando tenemos consultas largas que no queremos que afecten al rendimiento de las transacciones.
  - Propagar los cambios por las noches por ejemplo.

Como un Data Warehouse, pero sobre los datos operacionales directamente

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Replicación con bajos niveles de consistencia

### ▷ Replicación Multimáestro

- Se permiten modificaciones en cualquier réplica
  - Modificaciones propagadas automáticamente a las demás réplicas.
- Uso de **compromiso en dos fases** para realizar la modificación de las réplicas
- Alternativa: uso de **propagación perezosa (Lazy propagation)**, en lugar de actualizar las réplicas como parte de la propia transacción.
  - Permite el funcionamiento del procesamiento de transacciones incluso si algunos nodos fallan.
    - Aumenta **Disponibilidad** 
    - Baja la **Consistencia** 
  - Dos aproximaciones
    - Propagar modificaciones al primario directamente y al resto de replicas de forma perezosa.
    - Realizar propagación perezosa desde cada réplica a todas las demás. **Causa más problemas de concurrencia que la anterior.**

- **Alta disponibilidad:** El SGBDs debe funcionar de forma ininterrumpida o casi.
  - **Robustez:** Habilidad de continuar funcionando incluso con fallos.
    - ▷ Detectar fallos
    - ▷ Reconfigurar el sistema para seguir funcionando
    - ▷ Recuperar los componentes que fallaron (procesadores, discos, comunicaciones, etc.)
  - Distintos tipos de fallos necesitan soluciones distintas.
    - ▷ No es posible distinguir entre fallos de nodos y particionamientos de red.
  - Ejemplos de reconfiguraciones
    - ▷ Transacciones activas en nodos que fallan deben de ser abortadas. Cuando el nodo reinicia, debe asegurarse que tiene el último valor de cada réplica de cada elemento de datos.
    - ▷ Si un nodo que falla, el catálogo debe de ser informado para indicarle que sus réplicas ya no están disponibles.
    - ▷ Si falla un componente principal (coordinador, serv. nombres, etc.), se debe de elegir otro nodo para que asuma ese rol.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad



Pro. Consultas

BDs Hetero.

- Comparativa entre **Sistema Remoto de Copia de Seguridad** y **Sistema Distribuido**
  - ▷ **Copia de Seguridad:** Solo se copian datos y registro histórico
    - Menor coste 
  - ▷ **Sistema distribuido:** Sistemas de control de concurrencia y recuperación deben de funcionar en todos los nodos del sistema.
    - Mayor disponibilidad 
- Selección del coordinador
  - ▷ Opción 1: Mantener un nodo backup listo para asumir el rol de coordinador.
    - Alta disponibilidad
    - Sobrecarga producida por la ejecución doble en coordinador y nodo backup
  - ▷ Opción 2: Ejecutar un **algoritmo de elección** para que otro nodo asuma el rol de coordinador

Veremos  
también en  
NoSQL

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Compromiso entre Consistencia y Disponibilidad •••

- ▷ Típicamente, **mayoría** de los nodos con réplicas deben participar para que una modificación se realice la mitad mas uno

- Si la red se partitiona en más de dos partes, cada partición podría no tener la mayoría suficiente de nodos para seguir.

- ▷ **Teorema CAP:** Un sistema solo puede tener dos de las siguientes

- **Consistencia:** Mismo resultado que una ejecución secuencial en un solo nodo
- **Disponibilidad:** Un nodo accesible, debe de responder a operaciones de lectura y escritura
- **Tolerancia al particionamiento:** El sistema debe de seguir funcionando si hay particionamiento de red.

en general asumimos que asumimos la tercera !!!! y buscamos compromiso entre las otras dos, porque perderíamos

- ▷ En un sistema distribuido, el particionamiento de red es inevitable, y la tolerancia al mismo es necesaria (seguir funcionando si hay particionamiento) mucho disponibilidad si hay que parar el sistema cada vez que falla un nodo

- Debemos sacrificar o consistencia o disponibilidad

- ▷ Si permitimos modificaciones incluso si alguna réplica no es accesible, entonces tendremos una base de datos inconsistente (sube disponibilidad y baja la consistencia)

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

- Aspectos a tener en cuenta para estimar el coste de una consulta
  - ▷ Acceso a disco (como en sistemas centralizados)
  - ▷ Coste de comunicaciones
  - ▷ Ganancia de rendimiento por ejecución paralela.
- Debido a las distintas combinaciones posibles de fragmentación (horizontal y/o vertical) y replicación, la optimización de consultas es mucho más complicada.
- En concreto, la optimización de las operaciones de Join es compleja y las diferencias entre unas estrategias u otras van a tener mucho más impacto en el rendimiento.
  - ▷ Mejorando mucho el rendimiento por la ejecución paralela
  - ▷ Empeorando mucho el rendimiento por la necesidad de mover datos entre nodos a través de la red.

BDs Hetero.

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

TODO lo anterior es asumiendo que la base de datos es homogénea (cada nodo tiene el mismo software instalado y es consciente de la existencia de los otros).

## Sistema multidatabase

▷ Capa de software encima de los SGBDs existentes

▷ Sistemas locales pueden tener

- Distintos modelos y lenguajes de definición y consulta de datos

- Ejemplo: Relacional con SQL, XML con XQUERY, etc.

- Distintos mecanismos de control de concurrencia y recuperación.

hay que dar la ilusión al usuario  
**Ilusión de integración** de datos a nivel lógico, sin necesitar integración física.

## Problemas para integrar todo de forma física (en el mismo SGBDs distribuido)

▷ Dificultades técnicas: Necesidad de una **gran inversión**. Nuevo SGBDs y migrar todas las aplicaciones.

▷ Dificultades de organización: Dificultad de integrar físicamente todos los datos de varias organizaciones. En un sistema multidatabase, los sistemas locales mantienen un **alto grado de autonomía** y sus propias aplicaciones.

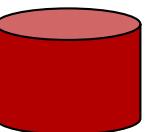
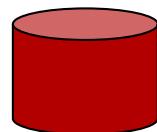
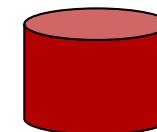
esto permite consultar fuentes en tiempo real, un datawarehouse no

### Sistema Multidatabase

SGBD1

SGBD2

SGBD3



Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Vista unificada de los datos

- ▷ El sistema multidatabase debe utilizar un **modelo de datos común**.
  - Por ejemplo: Relacional con SQL
    - Ejecución de consultas SQL sobre fuentes no relacionales
      - Ejemplo: **PostgreSQL Foreign Data Wrappers**
- ▷ Necesidad de proporcionar un **esquema conceptual común**
  - Problemas con la heterogeneidad semántica
    - Columnas con nombres iguales y significados distintos, etc.
    - Tipos de datos no soportados por algunos sistemas
      - Transformación de tipos puede ser complicada
    - Distintas unidades de medida en los datos de algunos atributos
    - Diferencias en la semántica de los propios datos y no solo en los metadatos
      - Ejemplo: En un nodo el país puede llamarse "Greece" y en otro "Ellada".



Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

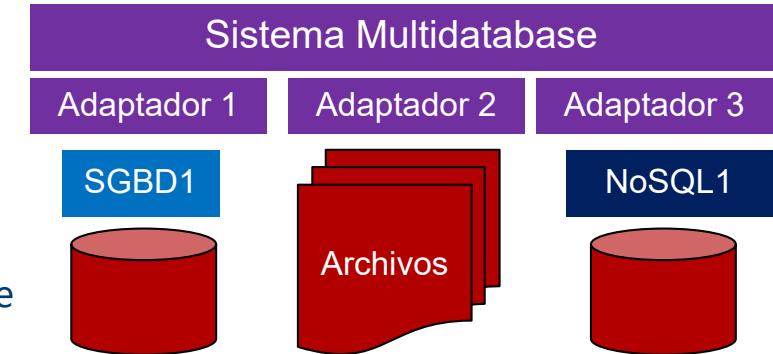
## ■ Procesamiento de consultas

### ▷ Problemas

- Consulta global ha de traducirse a consultas sobre los esquemas locales. Los resultados de cada subsistema han de traducirse al modelo común y unirse.
  - Tarea que se simplifica si se usan **Adaptadores (Wrappers)**
    - Vista global de datos locales. Traducen consultas y resultados.
- **Capacidades de consulta** de los distintos subsistemas pueden ser distintas.
  - Ejemplo: Un subsistema puede no permitir hacer joins, con lo que deben hacerse en la capa del sistema multidatabase
- Necesidad de **procesar los resultados** de cada sitio para unirlos con los demás (eliminación de duplicados, etc.)
- **Optimización global** de las consultas es muy compleja
  - Dificultad de conocer el coste de la ejecución de planes concreto sen subsistemas concretos.
  - Solución: Optimización local + heurísticas a nivel global.

### ▷ Sistemas Mediador

- Solo proporcionan capacidades de consulta globales, y no transacciones.



### Notaciones alternativas

- Sistemas multidatabase
- Sistemas mediador
- Bases de datos Virtuales

Introducción

Almacenam.

Transacciones

Compromiso

Concurrencia

Disponibilidad

Pro. Consultas

BDs Hetero.

## ■ Gestión de transacciones

- ▷ Tipos de transacciones
  - Locales: Se ejecutan en cada sistema de bases de datos
  - Globales: Se ejecutan bajo el control del sistema multidatabase
- ▷ Al mantener la autonomía de cada subsistema, el sistema multidatabase no puede saber que transacciones se están ejecutando en cada subsistema
  - Sincronización de estas ejecuciones no es posible (control de concurrencia)
- ▷ Simplifica imponer restricciones como que las transacciones globales solo puedan ser de lectura



# Bases de datos Distribuidas

Capítulo 19: Bases de datos distribuidas. A. Silberschatz,  
H.F. Korth, S. Sudarshan, Database System Concepts, 6th  
Edition, McGraw-Hill, 2014

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

# Bases de datos Paralelas

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

- **Introducción**
- **Paralelismo de Entrada/Salida**
- **Paralelismo Interquery**
- **Paralelismo Intraquery**
- **Paralelismo Intraoperation**
- **Paralelismo Interoperation**
- **Optimización**
- **Diseño de sistemas paralelos**
- **Paralelismo en sistemas multinúcleo**

## Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

## Motivación

- ▷ Incremento en el volumen de transacciones a procesar en un sistema OLTP
  - On Line Transaction Processing (OLTP) tienen mucha carga de lectura insercion y borrado, atienden a los procesos de la organizacion. Procesos del dia a dia de los trabajadores: el problema viene de hacer muchas por segundo, no por una transaccion muy voluminosa
- ▷ Enormes volúmenes de datos a procesar para el apoyo a la toma de decisiones
  - On Line Analytical Procesing (OLAP) gran volumen de datos almacenados (normalmente historico) para hacer analitica para poder aprender y predecir para apoyar la toma de decisiones
- ▷ Facilidad para paralelizar operadores de procesamiento de conjuntos (Tablas)
- ▷ Bajada del precio de hardware
- ▷ Irrupción de las arquitecturas multinúcleo

no siempre mejora una y la otra, a veces es facil mejorar el throughput pero no tanto el response time (por ejemplo, leer una fila

## Medidas de rendimiento

- ▷ **Response Time:** Tiempo de respuesta ejecución aislada de una consulta
- ▷ **Throughput:** Número de consultas ejecutadas por unidad de tiempo.

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

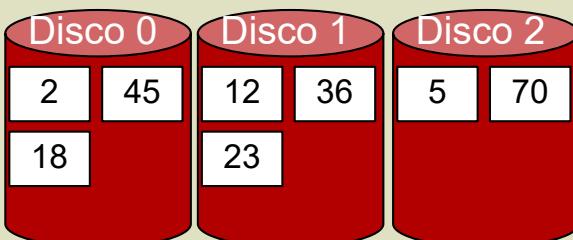
Diseño Sistem.

Multinúcleo

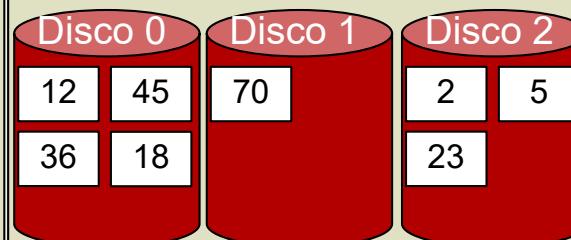
- Paralelizar el acceso al disco      recordar que el disco es lento !!
  - ▷ Particionamiento horizontal de las tablas en varios discos <sup>sharding</sup>  
esto es importante, hay que tener varios discos
- Estrategias (n discos numerados 0...n-1)
  - ▷ la que mejor reparte los datos estan mejor repartidos
  - ▷ **Round-robin:** Tupla número  $i$  se almacena en el disco  $i \bmod n$
  - ▷ **Particionamiento hash:** Aplica función de hash sobre algunos atributo para generar un valor entre 0 y n-1, que determina el disco.
    - la función de hash NO es variable
      - Intenta buscar una distribución aleatoria y uniforme de los datos, basada en una clave de búsqueda.
    - con el round-robin no sabemos donde esta una cierta clave, con hashing si
  - ▷ **Particionamiento por rango:** Valores contiguos de una clave (claves) de búsqueda van al mismo disco. Usa un vector de bordes de partición.  
este permite búsquedas por rango, entre un mínimo y un máximo, sin tener que ir a todos los discos

Clave (k): 2, 12, 5, 45, 36, 70, 18, 23

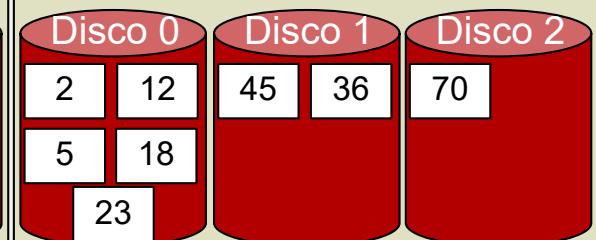
**Round-robin**     $i \bmod 3$



**Hashing**     $k \bmod 3$



**Rango**    Bordes: (33, 66)



Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

## ■ Comparativa de las estrategias

- ▷ Tipos de acceso a los datos
  - Escanear una relación completa (**scan**)
  - Consultas de punto (**point query**): Condiciones del tipo *Atributo = valor* ( $A = v$ )
  - Consultas de rango (**range query**): Condiciones del tipo  $v1 < A < v2$
- ▷ Eficiencia en función del tipo de consulta
  - Round-robin Solo es eficiente en **scan**.
  - Hash
    - Eficiente en **point query** por la clave de particionamiento.  
por liberar los demás discos el rendimiento no mejora !!! (tardo lo mismo en leer tres a la vez que uno)
      - Libera todos los discos menos uno (mejora el “throughput” aunque no mejore el “Response time”)  
a mi no me sirve, pero a otros usuarios a lo mejor si
    - Eficiente en **scan** si la función de hash es buena (distribuye uniformemente)
  - Rango
    - Eficiente en **point y range query** sobre la clave de particionamiento (mejora el throughput)
    - **Sesgo (skew)** de ejecución: muchas tuplas en el rango de búsqueda y pocos discos.

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

## ■ Comparativa de las estrategias

- ▷ La estrategia de particionamiento afecta a los algoritmos que pueden utilizarse para algunas operaciones
  - Ejemplo: Algoritmos para la operación de Join, y necesidad de mover tuplas entre nodos.
- ▷ En general, los sistemas eligen entre **Hash** o **Rango** para particionar las tablas
  - Algunos sistemas permiten elegir al usuario y otros ya solo proporcionan una opción.
- ▷ Las relaciones pequeñas en general es mejor no particionarlas
  - Tabla con **M bloques** de disco y sistema con **N discos**
  - Ideal, dividir en **min(M, N) particiones**. Nunca más particiones que el número de bloques.

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

## ■ Tratamiento del sesgo (Skew)

- ▷ Muchas tuplas en algunas particiones y muy pocas en otras
  - Los discos con muchas tuplas se convierten en **cuellos de botella** durante la ejecución.
  - Puede ocurrir cuando no usamos round-robin
- ▷ **Tipos de sesgo**
  - Sesgo de valor de atributo (**attribute-value skew**): Muchas tuplas con mismo valor
    - esto seria mi culpa de elegir mal la clave
  - Sesgo de particionamiento (**partition skew**)
    - Particiones mal balanceadas incluso sin attribute-value skew.
    - Más habitual en particionamiento por rango y menos en hashing
    - Impacto en rendimiento se nota mucho más cuando aumenta el paralelismo
- ▷ Vector de particionamiento por rango balanceado
  - Puede construirse **ordenando** los datos por la clave de particionamiento
    - Problema: Sobrecarga de E/S para realizar la ordenación
  - Puede construirse generando y manteniendo un **histograma** (construcción vía muestreo) para saber mas o menos cuantas filas se leerán al hacer un where para ver si uso un indice o no

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

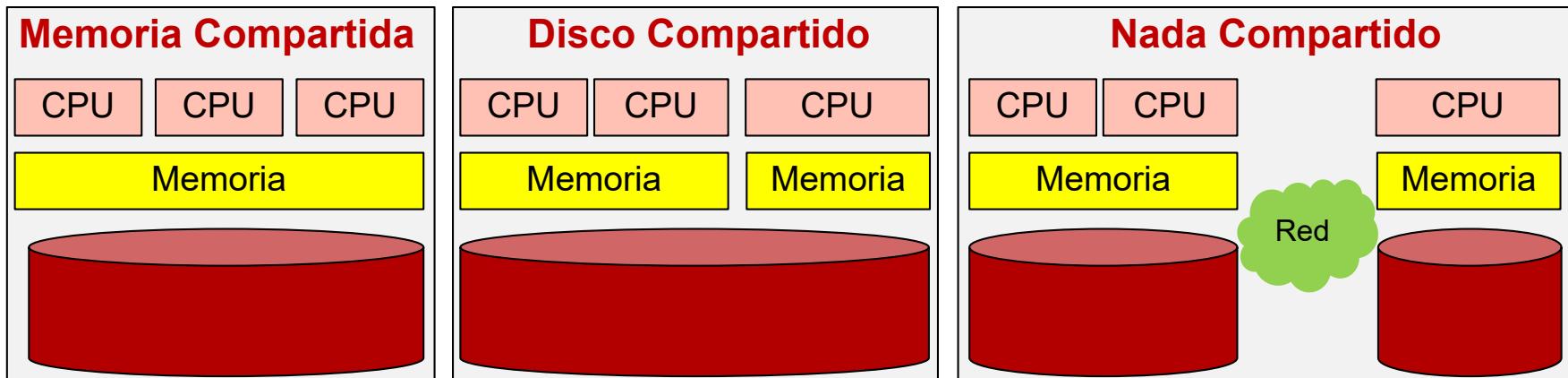
Interoperation

Optimización

Diseño Sistem.

Multinúcleo

- Ejecución en paralelo de varias consultas. del mismo o distintos usuarios
- Puede mejorar el **Throughput** pero no el **Response time**.
- Implementación
  - ▷ Fácil de implementar en arquitecturas de **memoria compartida**
  - ▷ Más complejo en **disco compartido** o **nada compartido (shared nothing)**
    - Coordinar la gestión de concurrencia y el registro de log entre varias máquinas
    - Problema de **coherencia de caché**
      - Asegurarse que los datos que hay en memoria principal siguen siendo consistentes, si hay otras máquinas que puedan haberlos modificado.



Introducción

Entrada/Salida

Interquery

Intraquery



Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

- Un único usuario lanza una consulta
- **Ejecución en paralelo de una sola consulta en varios procesadores / discos, etc.**
- Importante en consultas con tiempos de ejecución largos
- Se necesita      cada operación es un nodo de nuestro árbol
  - ▷ Ejecutar una operación en paralelo (**intraoperation**) este es el que da la escalabilidad
    - Particionar los datos y ejecutar en paralelo sobre cada partición
    - Puede escalar mucho en conjuntos de datos grandes (muchas particiones)
  - ▷ Ejecutar varias operaciones independientes en paralelo (**interoperation**)
    - Uso de **pipelines**
    - Escala poco ya que una consulta no suele tener muchas operaciones

Introducción

Entrada/Salida

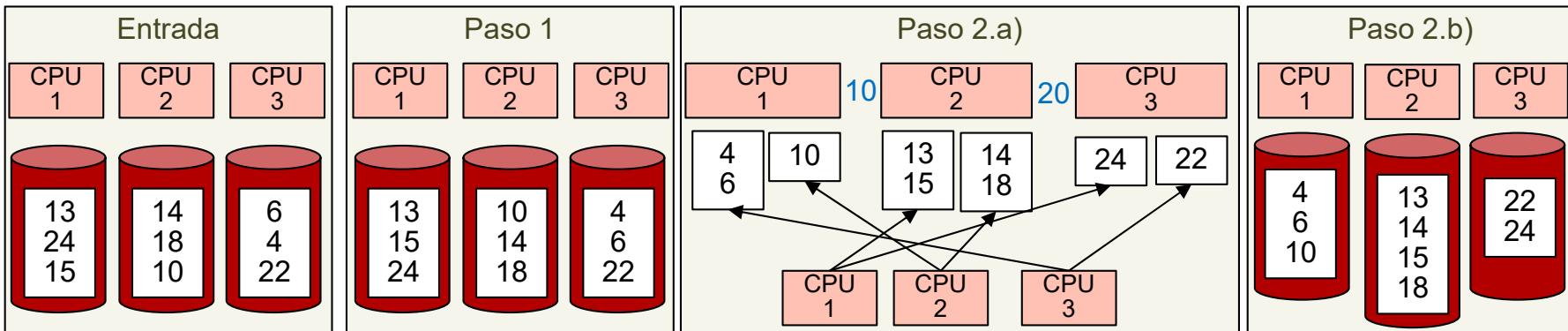
Interquery

Intraquery

**Intraoperation**

- Veremos un par de ejemplos      QUE NO HAY QUE SABER NO LO VA A PREGUNTAR
- Ordenación (Parallel External Sort-Merge)
  - ▷ Relación particionada de alguna forma entre los discos
  - 1. Cada procesador ordena los datos de un determinado disco (partición)
  - 2. Se mezclan las particiones ordenadas para obtener el resultado ordenado
    - a) Reparticionar por rango las particiones ordenadas entre los procesadores. Cada procesador recibe las tuplas de forma ordenada.
    - b) Cada procesador mezcla los streams ordenados que recibe de entrada para generar un stream de salida ordenado
    - c) El sistema concatena los streams de los procesadores para generar la salida.
  - ▷ Sesgo de ejecución en la recepción de datos en paso 2.b). Solución: enviar primer bloque de cada partición, luego segundo, etc.

tiempo de respuesta: lo que tarde en ordenar el mas grande



Introducción

Entrada/Salida

Interquery

Intraquery

**Intraoperation**

Interoperation

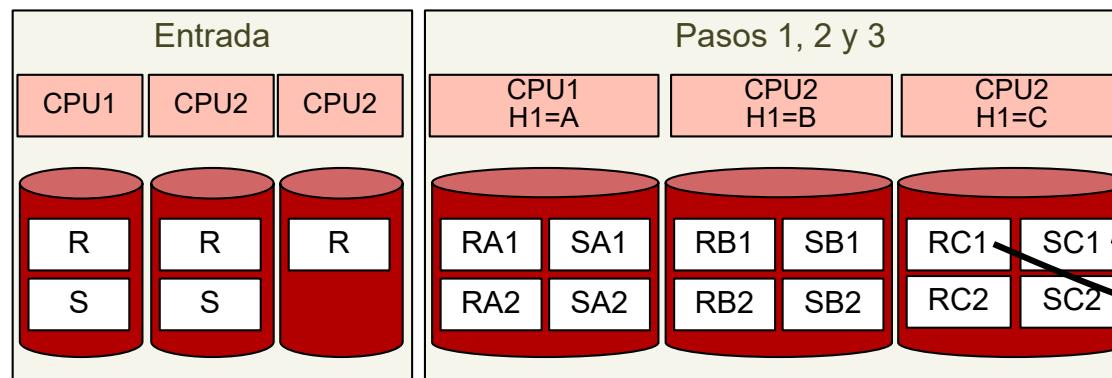
Optimización

Diseño Sistem.

Multinúcleo

## ■ Join (Partitioned Parallel Hash Join)

- ▷ Relaciones R y S (S más pequeña que R) particionadas de alguna forma entre los discos
- 1. Utilizar una función hash H1 para distribuir la relación S entre los procesadores.
- 2. Utilizar una segunda función hash H2 para particionar S dentro del disco de cada procesador. Cada partición de S debe de caber en memoria.
- 3. Utilizar la función hash H1 para distribuir la relación R entre los procesadores, y la función H2 para particionar R dentro de cada procesador.
- 4. En cada procesador, para cada número de partición de H2 se crea un índice hash (usando una tercera función hash H3) en memoria para la partición de S, y para cada tupla de R en la misma partición se buscan tuplas de S en el índice para generar tuplas del resultado.



H3(joinattrs)	tuplas S
x	tsx1, tsx2, ...
y	tsy1, tsy2, ...
z	tsz1, tsz2, ...
w	tsw1, tsw2, ...
<b>tupla R</b>	
tr1	
tr2	
tr3	
tr4	
tr5	

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation



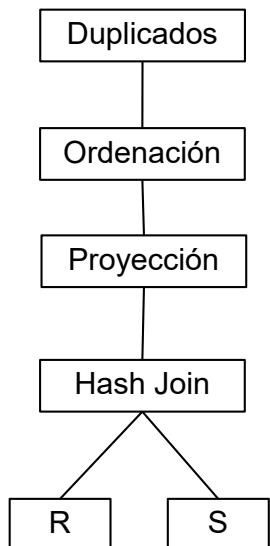
Optimización

Diseño Sistem.

Multinúcleo

## ■ Evaluación de expresiones

- ▷ **Materialización** hay dos formas de pasar datos de una expresión a otra. Eso lo tenemos por ejemplo en la ordenación. Hasta que no esté todo ordenado, no se puede empezar la limpieza de duplicados
  - El resultado del operador se almacena en una tabla temporal
  - Hasta que termina la ejecución de una operación no se puede iniciar la siguiente
- ▷ **Pipelining (tuberías)**
  - Cada tupla generada por un operador se envía al siguiente operador antes de empezar a procesar la siguiente tupla
  - Si todo el árbol se evalúa con pipelining, se pueden generar resultados en la salida muy pronto
  - **Implementación**
    - **Guiado por demanda** por demanda, no puede haber paralelismo
      - Cada operador intenta generar la siguiente tupla solo cuando su operador cliente se la solicita con una llamada tipo **next()**
    - **Guiado por el productor**
      - Antes de que se las soliciten, cada operador pide las tuplas que necesite en su entrada y empieza a generar salidas en un buffer.
      - Permite que cada operador se ejecute en un hilo independiente de forma paralela



Introducción

Entrada/Salida

Interquery

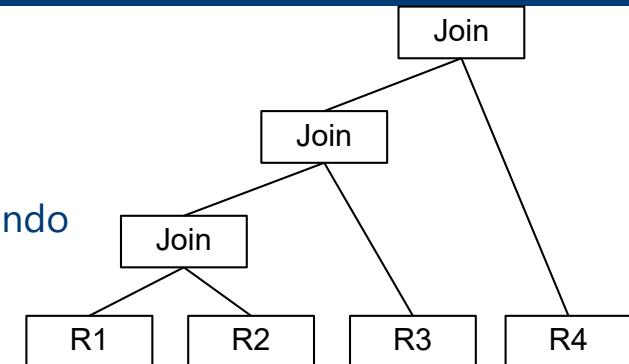
Intraquery

Intraoperation

Interoperation

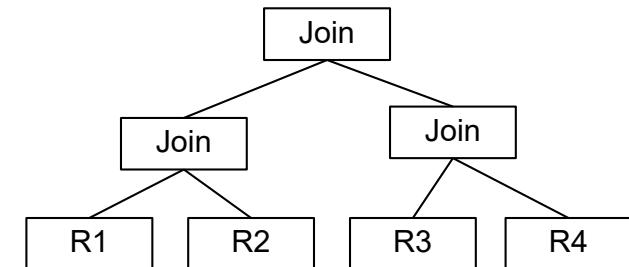
## ■ Paralelismo de pipelining

- ▷ Ejemplo: Join de 4 tablas
  - Mientras el primer join procesa una tupla, el segundo puede estar procesando un resultado anterior.
  - Útil cuando tenemos pocos procesadores, ya que una consulta no va a tener un gran número de operadores que puedan combinarse con pipelining
  - No sirve para operadores que requieren materialización (Ordenación por ejemplo)
  - Si el coste de un operador es mucho mayor que los demás, éste se convierte en un cuello de botella y el beneficio no es grande.
  - Más importante el no tener que escribir resultados intermedios que lo que aporta en paralelización



## ■ Paralelismo independiente

- ▷ Dos operaciones independientes se pueden ejecutar en paralelo
- ▷ Mismo Join de 4 tablas
- ▷ También útil cuando hay pocos procesadores



Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

- Optimizador: Componente fundamental en un SGBDs relacional
  - ▷ Busca el mejor plan de ejecución para cada consulta
- Con la paralelización el trabajo del optimizador se hace **mucho más complejo**
  - ▷ Muchas más combinaciones para hacer planes distintos.
  - ▷ Como paralelizar cada operación (Cuantos procesadores utilizar)
  - ▷ Como ejecutar cada operación (Pipelining, independientes, secuenciales)
- ¿Cómo definir los **recursos** de cada tipo que debería usar cada operación?
  - ▷ Algunas operaciones es mejor no ejecutarlas en paralelo
- Evitar **pipelines largos** (las últimas operaciones pueden tener que esperar mucho, y consumen recursos)
- Uso de **heurísticas** para no tener que generar todos los planes posibles
  - ▷ Ejemplo de Teradata: No usa pipelining paralelo y las operaciones se paralelizan con todos los procesadores
  - ▷ Optimizar en secuencial y después paralelizar las operaciones.

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

**Diseño Sistem.**

- **Disponibilidad:** Cuestiones a tener en cuenta
  - ▷ Tolerancia a fallos de algunos procesadores o discos
  - ▷ Reorganizaciones de datos en tiempo de ejecución y cambios de esquema
- **Probabilidad de fallo** de un sistema paralelo es mayor
  - ▷ Un procesador, un fallo cada 5 años. 100 procesadores fallarían cada 18 días
  - ▷ Replicar datos en al menos 2 procesadores
- Modificaciones del esquema pueden tardar mucho con grandes volúmenes de datos (añadir atributos, crear índices, etc.)
  - ▷ Se deben poder ejecutar on-line (sin parar la ejecución de transacciones)
- Ejemplos de sistemas:
  - ▷ Netezza (IBM)
  - ▷ DATAAllegra (Microsoft)
  - ▷ GreenPlum (Open Source)
  - ▷ Aster Data
  - ▷ CITUS Data (prácticas)

Multinúcleo

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

- Todos los SGBDs actuales se ejecutan en una plataforma paralela
- Paralelismo vs Velocidad cruda
  - ▷ Progreso exponencial de la velocidad de los procesadores
  - ▷ **Eficiencia energética:** Alta velocidad basada en más consumo energético
    - Duración de la batería, consumo eléctrico, dissipación de calor, etc.
  - ▷ Solución: colocar varios procesadores (núcleos) en cada chip
    - Procesador multinúcleo
- Memoria caché y procesos multihilo
  - ▷ Memoria principal pasa a ser un cuello de botella
    - Solución: Incluir varios niveles de memoria cache (L1, L2, etc.)
  - ▷ Necesidad de extender la jerarquía de memoria asumida por el SGBDs
    - SGBDs controla tráfico entre memoria y disco. Tráfico entre cachés asumido por hardware
  - ▷ Utilizar varios hilos para minimizar impacto de los fallos de caché
    - Si un hilo falla en el acceso a caché, otro hilo asume el control mientras se cargan los datos del primero (como en la paralelización de la entrada/salida a disco).

Introducción

Entrada/Salida

Interquery

Intraquery

Intraoperation

Interoperation

Optimización

Diseño Sistem.

Multinúcleo

- Adaptando el diseño de los SGBDs para arquitecturas modernas
  - ▷ Uso eficiente de arquitecturas modernas (multinúcleo) es un reto
  - ▷ Cantidad de datos necesarios en cache aumenta con el número de hilos
    - Aumentan los fallos de cache
      - Incluso un núcleo con varios hilos en ejecución puede tener que esperar por el acceso a memoria principal
  - ▷ Control de concurrencia
    - Restricciones en el acceso a los datos en concurrencia
      - Esperas por datos o retrocesos de transacciones potencialmente problemáticas
    - Para disminuir los conflictos, puede ser necesario aumentar los datos en caché (más fallos de cache)
  - ▷ Componentes del SGBDs compartidos por las transacciones
    - Gestor de bloqueos, gestor del buffer, gestor del registro histórico, etc.
    - Todos son potenciales cuellos de botella
  - ▷ Muchas transacciones pueden no aprovechar bien el hardware
  - ▷ Área activa de investigación

# Bases de datos Paralelas

Capítulo 18: Bases de datos paralelas. A. Silberschatz, H.F. Korth, S. Sudarshan, Database System Concepts, 6th Edition, McGraw-Hill, 2014

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209  
**Telf:** 881816463  
**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)  
**Skype:** jrviqueira  
**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

# Bases de datos NoSQL: Modelado

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

- **Motivación: ¿Por qué NoSQL?**
- **Modelos de datos**
- **Bases de datos sin esquema**
- **Modelando para el acceso a datos**

## Motivación

Modelos Datos

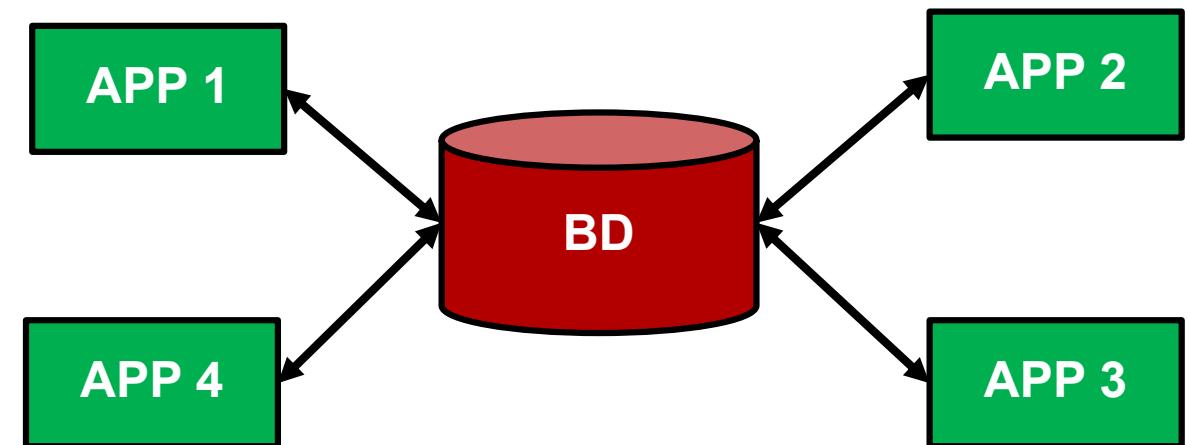
Sin Esquema

Acceso Datos

### ■ Bases de datos Relacionales

- ▷ Dominantes durante muchos años (siguen siéndolo)
- ▷ Filtrado eficiente de datos persistentes
- ▷ Control en el acceso concurrente: Consistencia
- ▷ Integración de aplicaciones
  - \_ Base de datos compartida
  - \_ Colaboración y coordinación a través de los datos persistentes
- ▷ Estandarización
  - \_ Modelo relacional
  - \_ Lenguaje SQL

muy importante la estandarización, para no tener que recodificar aplicaciones al cambiar la BD (oracle, ibm, etc)



# Motivación: ¿Por qué NoSQL?

Motivación

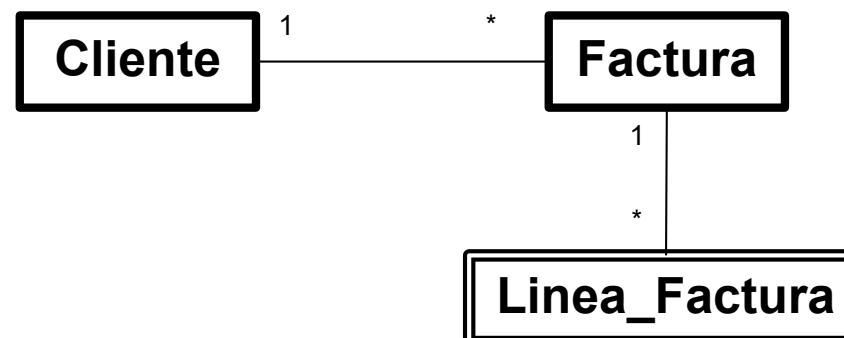
Modelos Datos

Sin Esquema

Acceso Datos

## ■ Impedancia entre modelos Memoria/Disco

- ▷ Problema
  - **Disco:** tuplas simples (1<sup>a</sup> Forma normal)
  - **Memoria:** Estructuras complejas (anidadas)
- ▷ Bases de datos orientadas a objetos
  - No llegaron a triunfar
- ▷ Mapeado objeto-relacional
  - No acaban de resolver el problema
  - Crean otros problemas
    - Eficiencia cuando tratamos de no usar el SGBDs



FACTURA		
[Nombre de la empresa]		
[Calle]	N.º DE FACTURA	FECHA
[Ciudad, provincia y código postal]	2034	21/02/2018
Teléfono: (000) 000-0000		
FACTURAR A		
[Nombre]	ID. DEL CLIENTE	TÉRMINOS
[Nombre de la empresa]	564	Pago contra entrega
[Calle]		
[Ciudad, provincia y código postal]		
[Teléfono]		
[Dirección de correo electrónico]		
<hr/>		
DESCRIPCIÓN	CANT.	PRECIO UNITARIO
Tarifa del servicio	1	200,00
Mano de obra: 5 horas a 75 € la hora	5	375,00
Descuento de cliente nuevo	-	50,00
	-	-
	-	-
	-	-
	-	-
	-	-
	-	-
	-	-
<hr/>		
Gracias por su confianza	SUBTOTAL	525,00
	TIPO IMPOSITIVO	4,250%
	IMPUESTOS	22,31
	TOTAL	547,31 €

If you have any doubts about this invoice, please contact us at [Nombre, teléfono, correo electrónico@dirección.com]

# Motivación: ¿Por qué NoSQL?

## Motivación

Modelos Datos

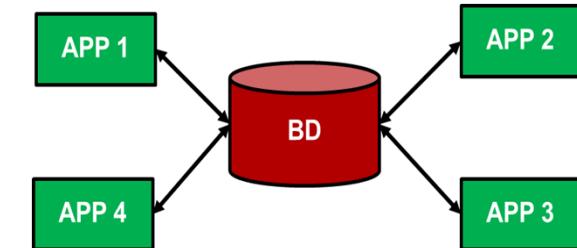
Sin Esquema

Acceso Datos

### ■ Integración de aplicaciones a través de una BD centralizada

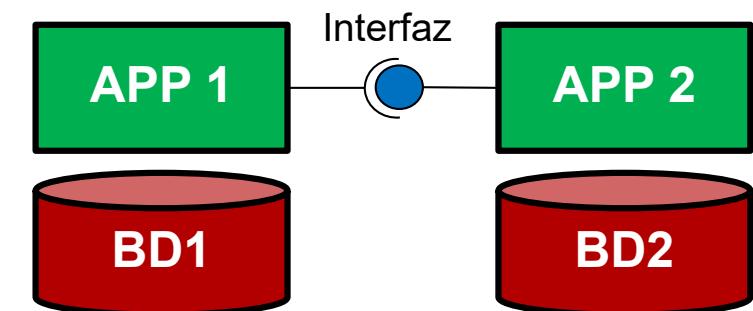
#### ▷ Problemas

- Base de datos compleja
- Cambios en la BD necesitan coordinación entre aplicaciones
  - Ejemplo: Índice necesario para una crea problemas de eficiencia en otra
- Son necesarios mecanismos de verificación de consistencia en la BD



#### ▷ Alternativa: Cada aplicación tiene su BD

- El equipo de cada app. conoce, mantiene y evoluciona su BD
- Verificación de integridad en cada app.
- Integración a través de **interfaces**
  - Servicios web
  - Uso de XML y JSON
- Distintos modelos de BD en distintas aplicaciones
- Se pueden mover responsabilidades de la BD a la app.
- Incluso en estas configuraciones, las BD relacionalles siguen siendo la opción dominante



## Motivación

Modelos Datos

Sin Esquema

Acceso Datos

### ■ Necesidad del uso de clusters

- ▷ Incremento de la escala de algunas aplicaciones
  - Monitorización de la actividad sitios web
  - Grandes conjuntos de datos
  - Muchos usuarios
- ▷ Necesidad de más recursos (Opciones)
  - Máquinas más potentes y caras (escalabilidad limitada)
  - Cluster de muchas máquinas pequeñas
    - Mayor Resiliencia y Fiabilidad
    - Menor Precio
- ▷ BD **relacionales** no diseñadas para clusters
  - BD sobre sistema de archivos distribuido
    - Único punto de fallo
  - Varios SGBDs para almacenar distintas partes de los datos
    - La fragmentación debe controlarla la aplicación.
- ▷ Aparecen **alternativas**: Big Table (Google), Dynamo (Amazon)
- ▷ Esta si parece una amenaza para la hegemonía de las BD **relacionales**



## Motivación

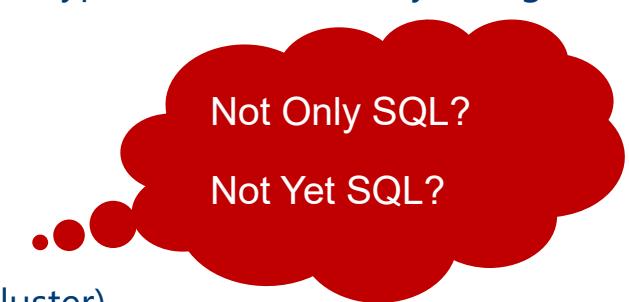
Modelos Datos

Sin Esquema

Acceso Datos

### ■ Aparición de las BD NoSQL

- ▷ Falta una definición clara del término NoSQL
  - Inicialmente: "Bases de datos no relacionales distribuidas y de código abierto"
    - Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB y MongoDB
- ▷ Características
  - No usan SQL (mucho debate sobre esto)
    - Lenguajes muy parecidos
    - ¿Si alguna acaba implementando SQL?
  - Funcionan sobre arquitecturas distribuidas (Cluster)
    - Impacto en el modelo de datos
    - Impacto en la solución de consistencia (problemas entre ACID y clusters)
      - Rango de opciones para distribución y consistencia
    - Excepción: Bases de datos de grafos
  - Necesidades de aplicaciones del siglo 21 (web, etc.)
  - Operan sin esquema
    - En BD relacionales acaban apareciendo campos como "customField6"



Not Only SQL?

Not Yet SQL?

## ■ Aparición de las BD NoSQL

- ▷ Nuevo abanico de opciones para el almacenamiento de datos, sin restringir la incorporación de nuevos tipos
  - Relacionales siguen siendo el tipo más utilizado (pero no el único)
- ▷ Uso como BD de aplicación y no como BD de integración
- ▷ Razones principales para considerar NoSQL
  - Tamaño y rendimiento que hace necesario el uso de un cluster
  - Productividad en el desarrollo de aplicaciones
    - Interacción con los datos más natural (cercano al entorno de aplicación)

Motivación

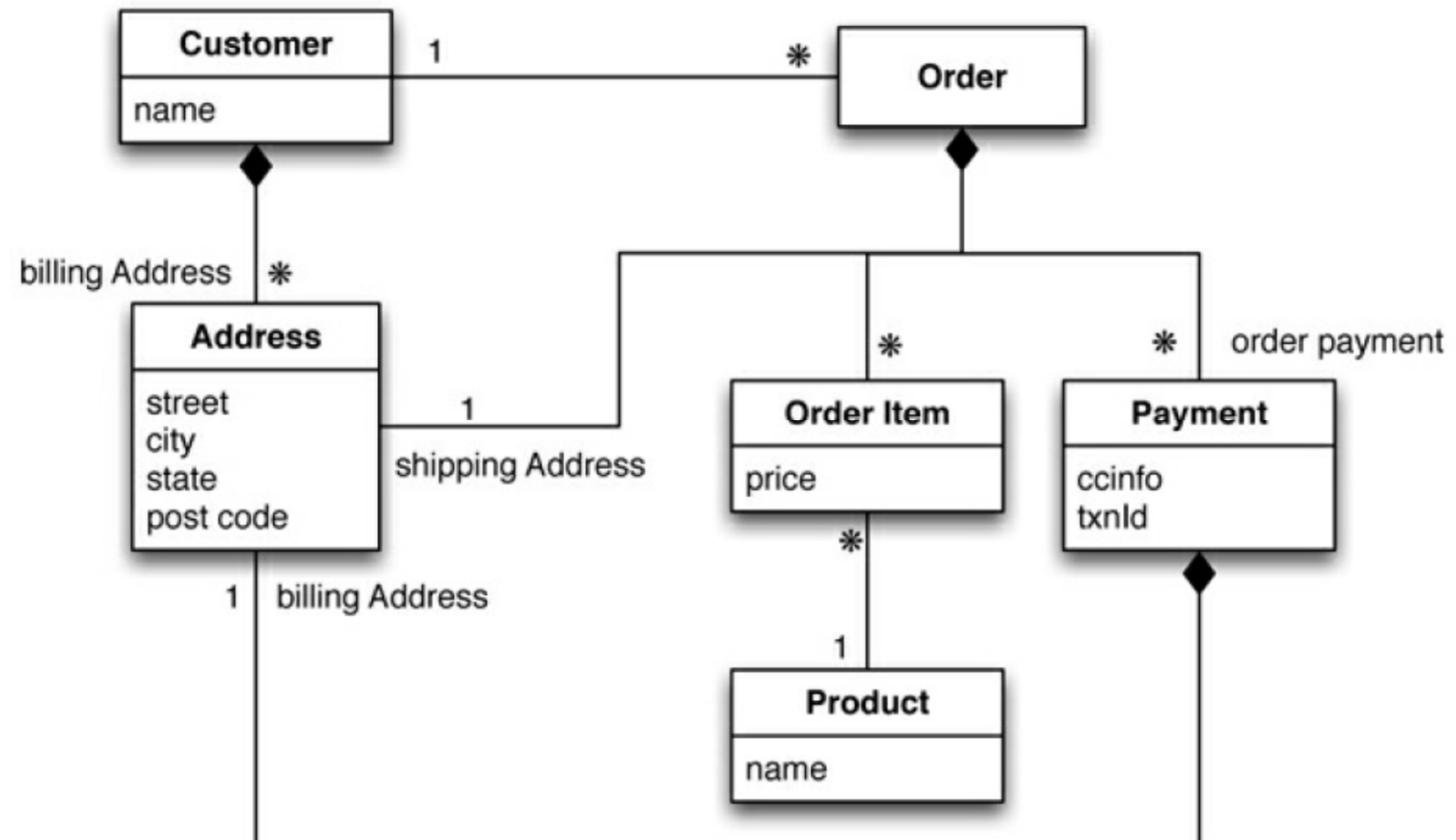
Modelos Datos

Sin Esquema

Acceso Datos

## ■ Agregados

El modelo de datos aparece, entre otros motivos, por el de agragados



Motivación

Modelos Datos

Sin Esquema

Acceso Datos

## ■ Agregados

agrega muchas estruturas

- ▷ Modelo relacional no permite anidar estructuras

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

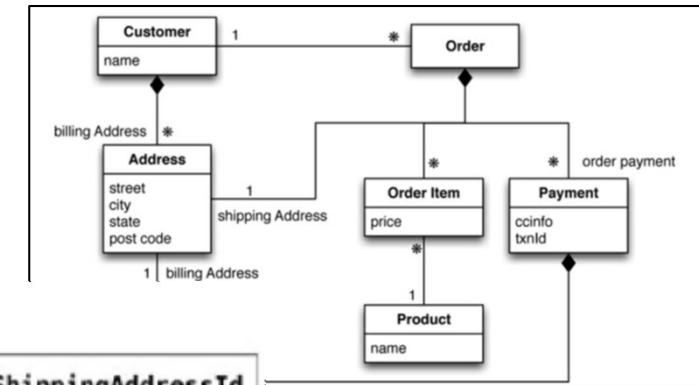
Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressId
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft



Motivación

Modelos Datos

Sin Esquema

Acceso Datos

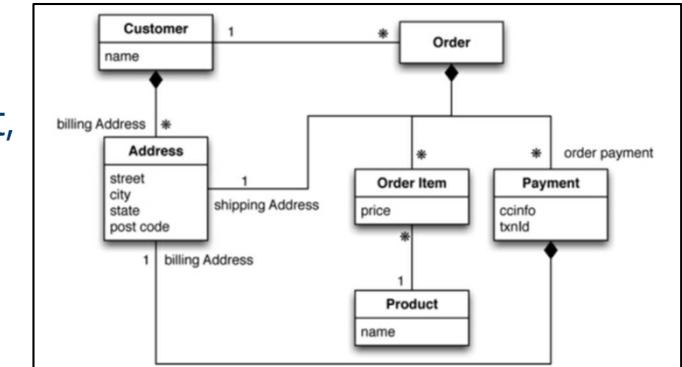
## ■ Agregados

- ▷ NoSQL: Key-Value, Column-Family, Document, Graph

```
// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}

// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment": [
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}
}
```

la gran diferencia es la utilizacion de agregados !!!!



JSON Document

Motivación

Modelos Datos

Sin Esquema

Acceso Datos

## ■ Agregados

- ▷ **NoSQL:** Key-Value, Column-Family, Document, Graph

- **Diseñar estructuras pensando en como se van a acceder**
  - Desnormalización para minimizar accesos
- **Consecuencias**
  - La agregación puede beneficiar unas consultas y perjudicar otras
  - Trabajo con un cluster
    - Determinar datos que deben de ir juntos para minimizar accesos a varios nodos
  - **BD NoSQL en general no soportan ACID en transacciones que se expanden por varios agregados**

```
// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}

// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment": [
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}
```

Motivación

Modelos Datos

Sin Esquema

Acceso Datos

## ■ Modelos Key-Value y Document

### ▷ Pares (Clave, Agregado)

#### - Key-Value

HBASE es un poco mas que esto cuando le pides una tira de bits, te devuelve la tabla de bits que corresponde permiten busquedas por rango tambien en la clave

- Agregado es opaco para la base de datos (BLOB)
- No imponen restricciones sobre el contenido del agregado
- Solo se puede consultar por clave

#### - Document

ejemplo MongoDB

- La base de datos entiende la estructura compleja del agregado
- Definen las estructuras posibles para el agregado (Ejemplo: JSON)
- Se puede eprmite ver y consultar dentro del agregado
  - consultar sobre los campos del agregado
  - recuperar solo parte del agregado
  - etc.

- Puede haber soluciones difíciles de clasificar por tener características de ambas

Ninguna de estas tiene esquema, la key-value no tiene ni estrutcura

carcaterísticas de los modelos. Hay cuatro grandes tipos:

Motivación

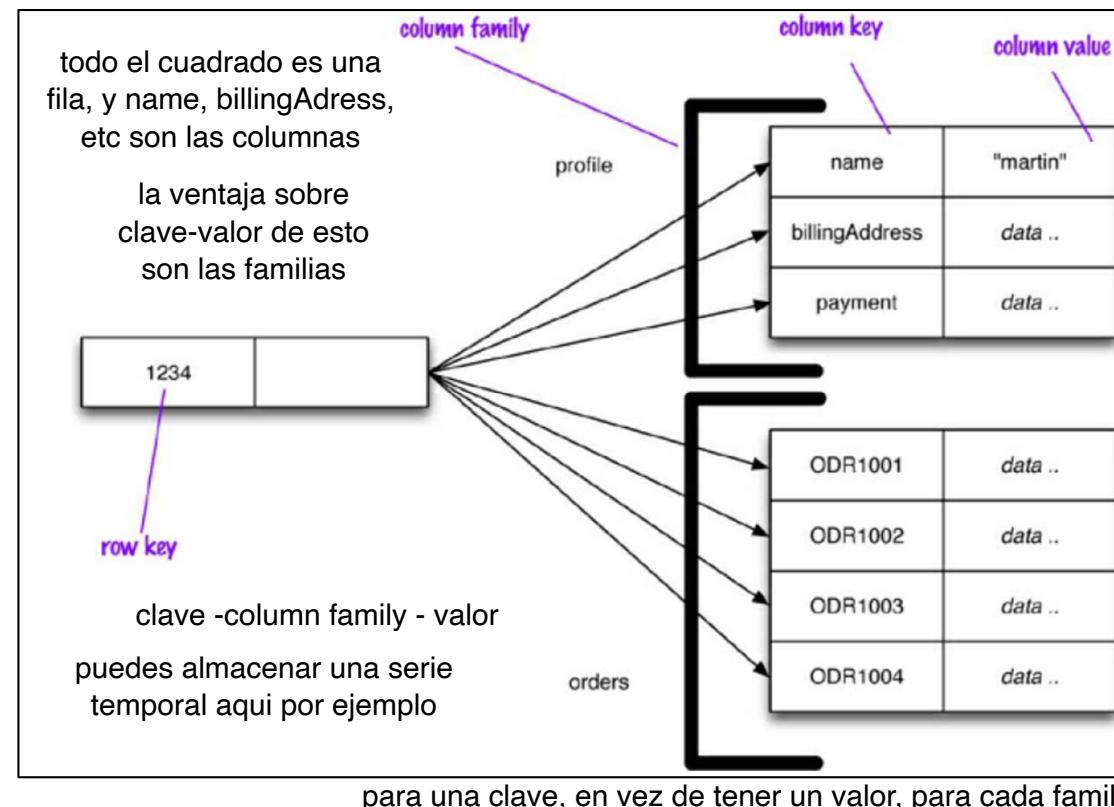
Modelos Datos

Sin Esquema

Acceso Datos

## ■ Modelo Column-Family

- ▷ Google Big Table      NO CONFUDIR CON ALMACENAMIENTO COLUMNAR
- ▷ Estructura tabular con columnas dispersas y sin esquema (**Column-Stores**).
- ▷ **Mapeo en dos niveles**    NO es una tabla
- ▷ Influencia para HBase y Cassandra
- ▷ **Column Families**: Se almacenan juntas.



el esquema es el nombre de las familias, lo único que se declara. En el ejemplo, profile y orders



EJEMPLO CASANDRA Y HBase.

Motivación

Modelos Datos

Sin Esquema

Acceso Datos

## ■ **Modelo Column-Family**

- ▷ Organización de los datos
  - Por filas
    - Cada fila es un agregado. Cada column family un bloque de datos dentro del agregado
  - Por columnas
    - Cada column family define un tipo de registro
- ▷ **Cassandra** usan una notación un poco distinta
  - Una fila solo puede pertenecer a una column family
  - Una column family puede tener **supercolumns** (con columnas anidadas)
    - Concepto equivalente a las column families de **Big Table** o **HBase**
- ▷ Una column family puede tener columnas distintas en cada fila (**sin esquema**)
  - Un nuevo pedido supone insertar una nueva columna en la column family "orders" de la fila correspondiente al cliente
- ▷ **Cassandra**
  - **Skinny row**: pocas columnas. Mismas columnas en la mayoría de filas (ej: profile)
  - **Wide row**: Muchas columnas (miles). Filas con columnas muy variadas. Modelan una lista. Orden para las columnas. Recuperar todos los pedidos, o un rango de pedidos en un solo acceso.

Motivación

Modelos Datos

Sin Esquema

Acceso Datos

## ■ Relaciones

- ▷ Acceso a datos relacionados de distintas formas
  - **Ejemplo: Cliente - Pedido** relación cliente-pedido
    - Obtener los pedidos de un cliente
    - Obtener los clientes de los pedidos de un producto concreto
  - La relación se almacena con una referencia a una clave. BD no conoce la existencia de la relación (es solo un dato más)
- ▷ Alternativa: **Hacer visibles las relaciones** para la base de datos
  - Dividir los datos y habilitar relaciones entre ellos
  - Modificar varios agregados en una sola transacción.
    - BD NoSQL no lo permiten normalmente
    - BD relacional proporciona ACID
  - BD relacional no muy eficiente si hay que realizar **muchos joins** para seguir muchas relaciones
    - Consultas difíciles de expresar
    - Bajo rendimiento

en estos casos de muchos join, muchas tablas en el from, hay que pensar si la relacional es la tecnología que realmente hace falta —>

**Motivación**  
**BD  
de  
Grafos**

Motivación

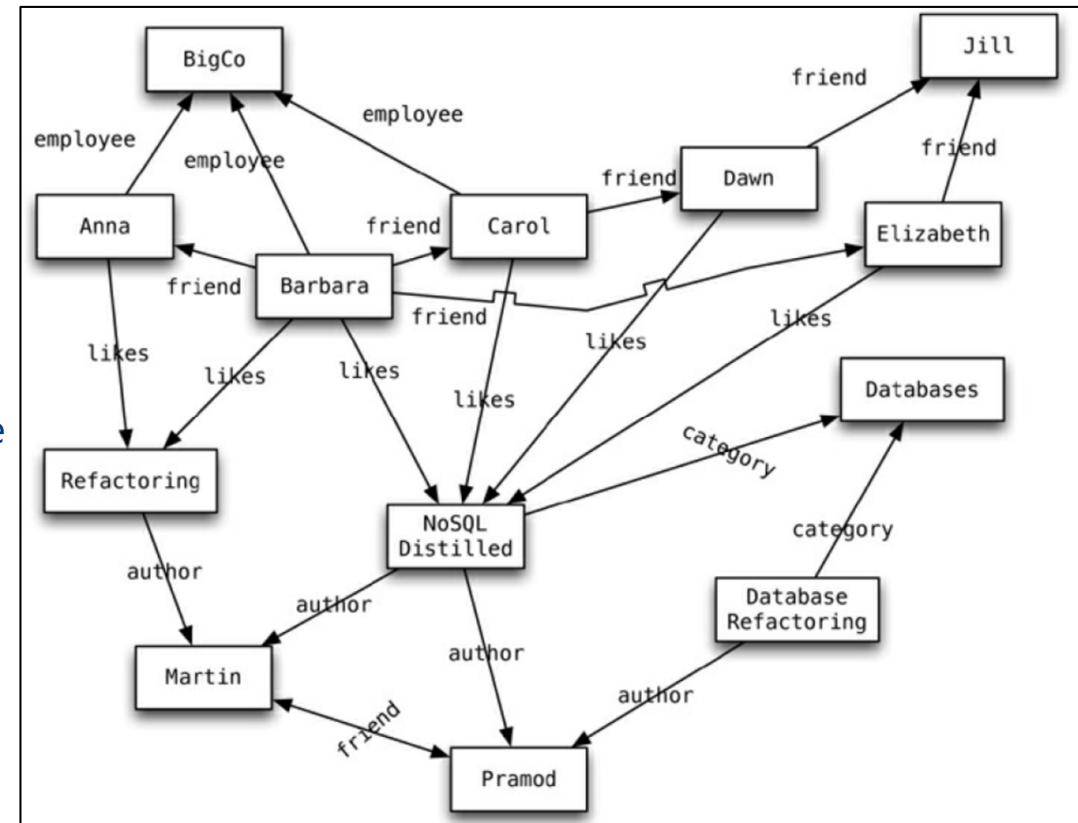
Modelos Datos

Sin Esquema

Acceso Datos

## ■ Bases de datos de Grafos

- ▷ Caso especial de NoSQL
  - No motivado por agregados y clusters
- ▷ Motivación
  - Registros muy simples
  - Muchas relaciones entre los datos
  - Ejemplo
    - Libros de bases de datos escritos por alguien que le guste a alguno de mis amigos
- ▷ Aplicaciones
  - Redes sociales, preferencias de productos, etc.
- ▷ Modelo compuesto por Nodos y Arcos
  - Posiblemente con datos en nodos e incluso en arcos
  - Neo4J: Objetos java como propiedades de los nodos y arcos (sin esquema)
- ▷ Navegación rápida de las relaciones. Inserción puede ser lenta.



al consulta suele ser entrar en un nodo y expandirse a buscar vecinos

los arcos son los enlaces

Motivación

Modelos Datos

**Sin Esquema**



Acceso Datos

- Característica común a todas las BD NoSQL
  - ▷ No tener esquema
- No es necesaria la previa definición de un esquema para almacenar datos
  - ▷ **Clave-valor:** Cualquier valor se puede insertar para una clave
  - ▷ **Document:** No hay restricciones sobre el contenido de cada documento
  - ▷ **Column-family:** Cualquier dato se puede almacenar sobre un columna de una fila
  - ▷ **Grafos:** La propiedades de nodos y arcos son libres
- **Ventajas de trabajar sin esquema**
  - ▷ No tener que hacer asunciones a priori
    - según van llegando los datos podemos insertar cualquier cosa
  - ▷ Facilidad para incorporar cambios en los datos
  - ▷ Facilidad para trabajar con datos no uniformes
  - ▷ Problemas de tener esquema
    - Inflexible en la inserción de datos      aparecen y desaparecen campos
      - Muchas columnas con valores nulos
      - Columnas con semántica poco definida
        - Ej: "columna3", "informacion", "comentarios", ... campo de texto donde va cualquier cosa

Motivación

Modelos Datos

**Sin Esquema**



Acceso Datos

## ■ Problemas de trabajar sin esquema

- ▷ Las aplicaciones necesitan normalmente cierto formato y semántica en los datos.  
hay que minimizar los literales para que el código se puede cambiar y mejorar de forma fácil
  - Nombres de las columnas, tipos de datos, etc.
  - Casi cualquier programa que escribimos confía en la existencia de cierto esquema implícito en los datos
- ▷ Tener el esquema codificado en las aplicaciones puede dar problemas
  - Tener que analizar código para entender los datos
  - La BD no puede utilizar el esquema para mejorar la eficiencia
    - Column-family tiene cierto esquema, por eso puede ser más eficiente
  - No se pueden implementar restricciones de integridad en la base de datos
- ▷ Los esquemas tienen valor y su rechazo **puede ser incluso alarmante**
- ▷ Una BD sin esquema realmente desplaza el problema del esquema a la aplicación
  - Puede ser peor si las aplicaciones se realizan por personas distintas



Motivación

Modelos Datos

**Sin Esquema**

Acceso Datos

## ■ ¿Soluciones?

- ▷ Opción 1: Integrar todo el acceso a la base de datos en una única aplicación que proporciona servicios web para las demás
- ▷ Opción 2: Delimitar diferentes partes de cada agregado para el acceso de aplicaciones diferentes

## ■ Esquemas en bases de datos relacionales

- ▷ Se critica su falta de flexibilidad
  - Pero se pueden modificar los esquemas
    - Añadir y quitar columnas por ejemplo

## ■ Problemas al almacenar datos de formas nuevas en BD NoSQL.

- ▷ Las aplicaciones tienen que funcionar con datos viejos y nuevos

pera modificar, añades una columna y convives con dos columnas, una con nulos hacia arriba y otra con nulos hacia abajo

Motivación

Modelos Datos

**Sin Esquema**



Acceso Datos

## ■ Cambios en el esquema

- ▷ Importancia en metodologías ágiles poder cambiar fácilmente de esquema
- ▷ En NoSQL se pueden hacer cambios rápido, pero hay que tener cuidado con las migraciones de esquema
- ▷ **Cambios de esquema en BD Relacionales**
  - Un cambio de esquema puede ser un proyecto en si mismo
  - Necesidad de crear scripts para la migración de datos de un esquema a otro
  - Proyectos nuevos
    - Almacenar los cambios con los scripts de migración de datos
    - Ejemplo: **DBDeploy** para manipular cambios en la BD
      - Guardar una tabla con todas la versiones del esquema
      - Herramientas para mantener el historial de versiones junto con las versiones de las aplicaciones
  - Proyectos legacy
    - Extraer el esquema de la BD
    - Proceder como con los proyectos nuevos
    - Mantener compatibilidad hacia atrás
      - Fase de transición en la que ambos esquemas funcionan
      - Código tipo **Scaffolding** (disparadores, vistas, etc.)

Motivación

Modelos Datos

**Sin Esquema**

## ■ Cambios en el esquema

### ▷ Cambios de esquema en BD NoSQL

- Se intenta no tener que realizar estos cambios de esquema el esquema esta en la aplicacion !
- Engañoso pensar que las BD NoSQL no tienen esquema
  - El esquema está en la aplicación
    - Debe parsear los datos obtenidos de la base de datos
  - Si no cambiamos la aplicación el error de esquema que daría la BD lo da la aplicación
    - Debemos cambiar el código que lee y el que escribe
- Migración incremental
  - Migrar todos los datos al nuevo esquema puede ser muy costoso
  - Alternativa      transicionar a partir de la propia aplicación,  
                      lee de ambos y siempre guarda en el nuevo
    - Leer de ambos esquemas (período de transición)
    - Guardar en el más reciente
  - Algunos datos no llegan a migrarse nunca
- Migración en BD de grafos
  - Alternativa a migrar todos los tipos de arco en la base de datos
    - Definir nuevos arcos con el nuevo esquema



Acceso Datos

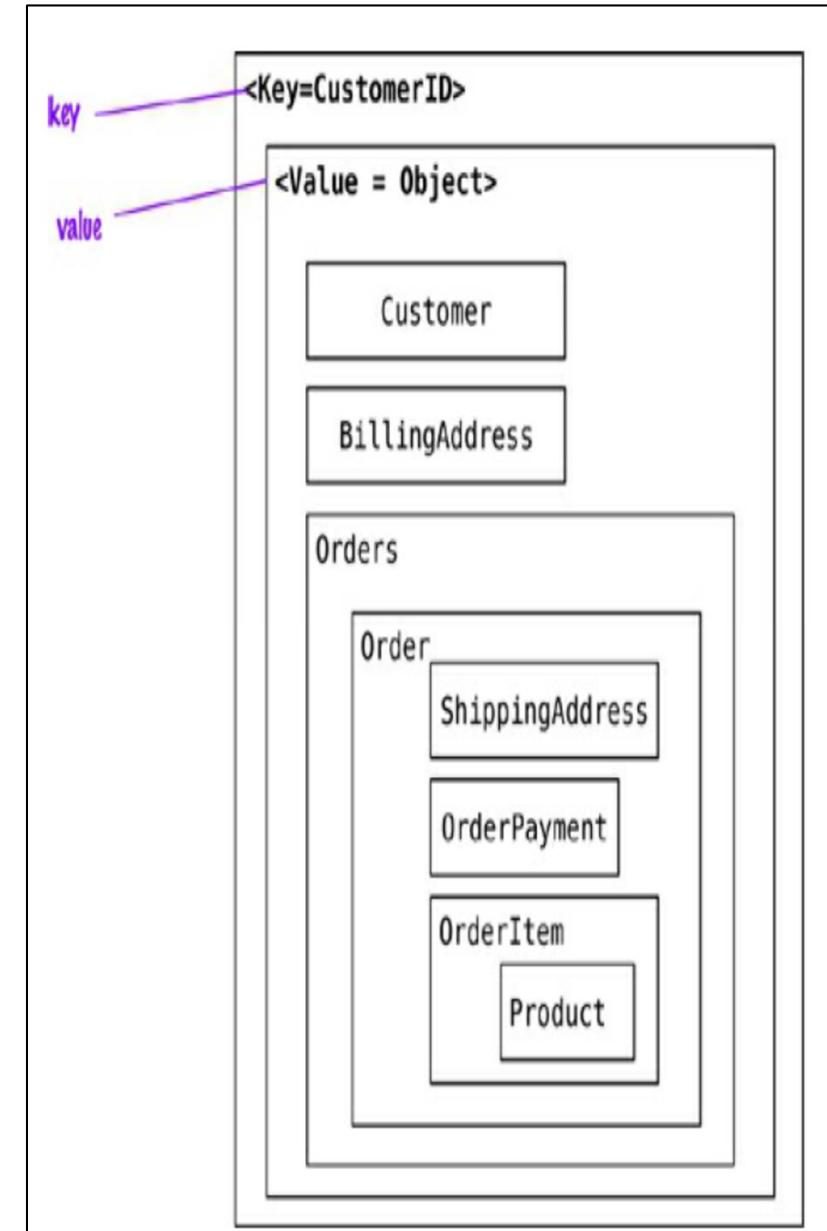
Motivación

Modelos Datos

Sin Esquema

Acceso Datos

- Modelado de los agregados
  - ▷ Tener en cuenta como va a ser el acceso
    - Ejemplo: Leer los datos de los productos necesita procesar todos los clientes



# Modelando para el acceso a datos

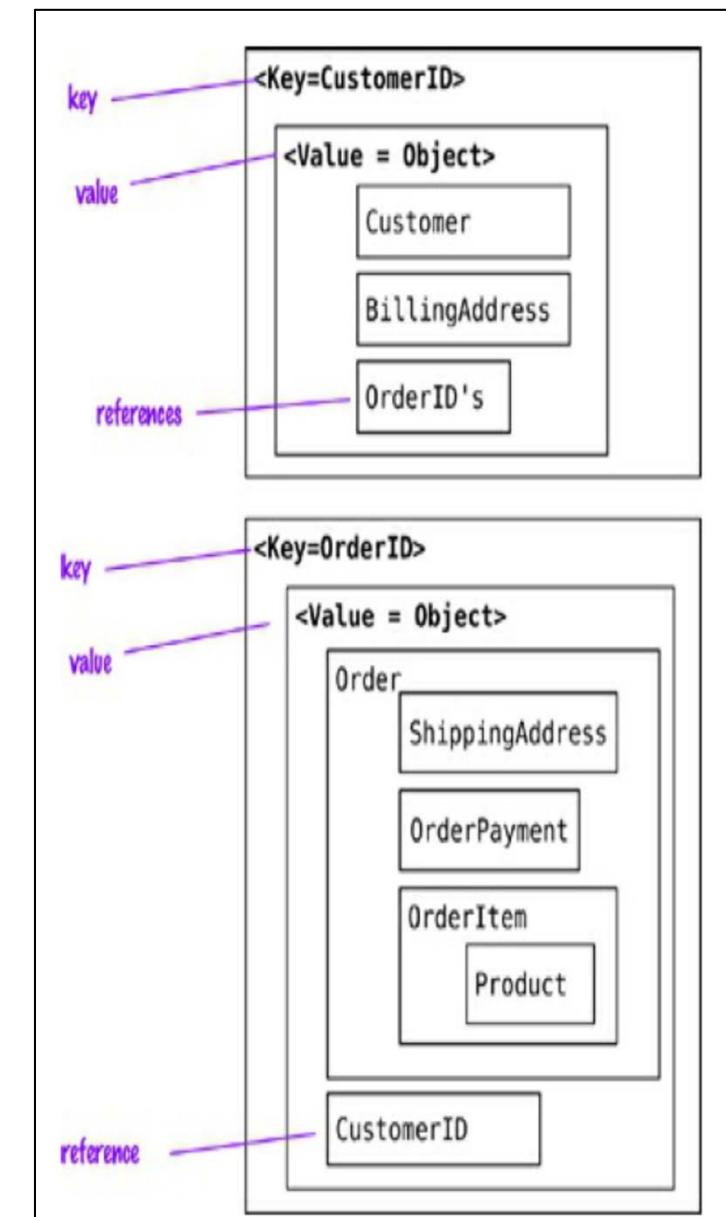
Motivación

Modelos Datos

Sin Esquema

Acceso Datos

- Modelado de los agregados
  - ▷ Tener en cuenta como va a ser el acceso
    - Ejemplo: Leer los datos de los productos necesita procesar todos los clientes
- Alternativa: Dividir el agregado y usar referencias
  - ▷ BD de documentos no necesitan almacenar las referencias en ambas direcciones
- Calcular agregados para realizar analítica concreta
  - ▷ Ejemplo: Almacenar en qué pedidos se encuentra cada producto
  - ▷ BD documentos permiten realizar la consulta sin calcular el agregado, pero no es eficiente
- Column-families
  - ▷ Lecturas eficientes. Denormalizar en las escrituras



# Bases de datos NoSQL: Modelado

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnoloxías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

# Bases de Datos NoSQL: Distribución

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

- **Introducción**
- **Particionamiento (Sharding)**
- **Replicación**
- **Procesamiento (Map-Reduce)**

## ■ Principal interés del uso de tecnologías NoSQL

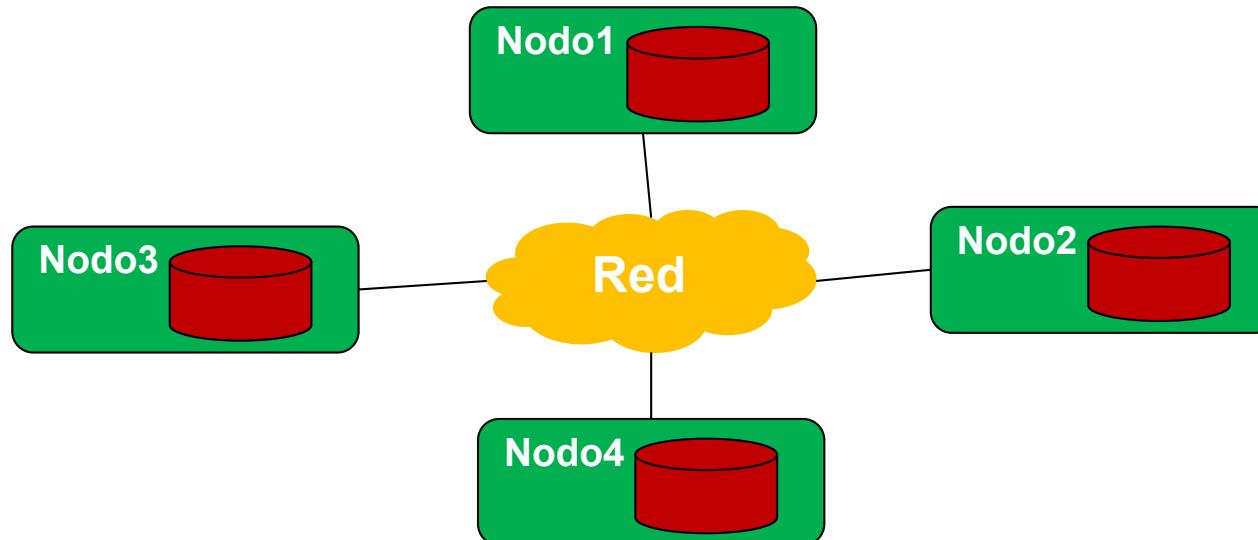
- ▷ Funcionar aprovechando los recursos de un **cluster** de computación
- ▷ Uso del **agregado** como unidad natural de distribución de datos.

## ■ Ventajas

- ▷ Capacidad para gestionar mayor **volumen** de datos
- ▷ Incrementar el tráfico de operaciones de lectura y escritura (**rendimiento**)
- ▷ Incrementar la **disponibilidad** (tolerancia a fallos)

## ■ Desventaja

- ▷ Sistema **más complejo**. Usar solo en caso necesario



## ■ Formas de distribución de los datos

### ▷ Replicación

- Varias copias del mismo dato en distintos nodos
- Arquitecturas
  - Maestro-esclavo
  - Peer-to-peer

### ▷ Particionamiento (Sharding)

- Repartir los datos entre los nodos
- ▷ Técnicas ortogonales: Se pueden combinar

## ■ Técnicas (de más sencillo a más complejo)

- ▷ Un solo servidor
- ▷ Replicación Maestro-Esclavo
- ▷ Particionamiento
- ▷ Replicación peer-to-peer

Introducción

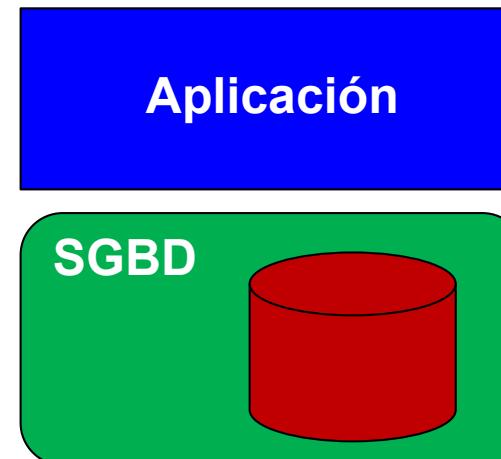
Sharding

Replicación

Map-reduce

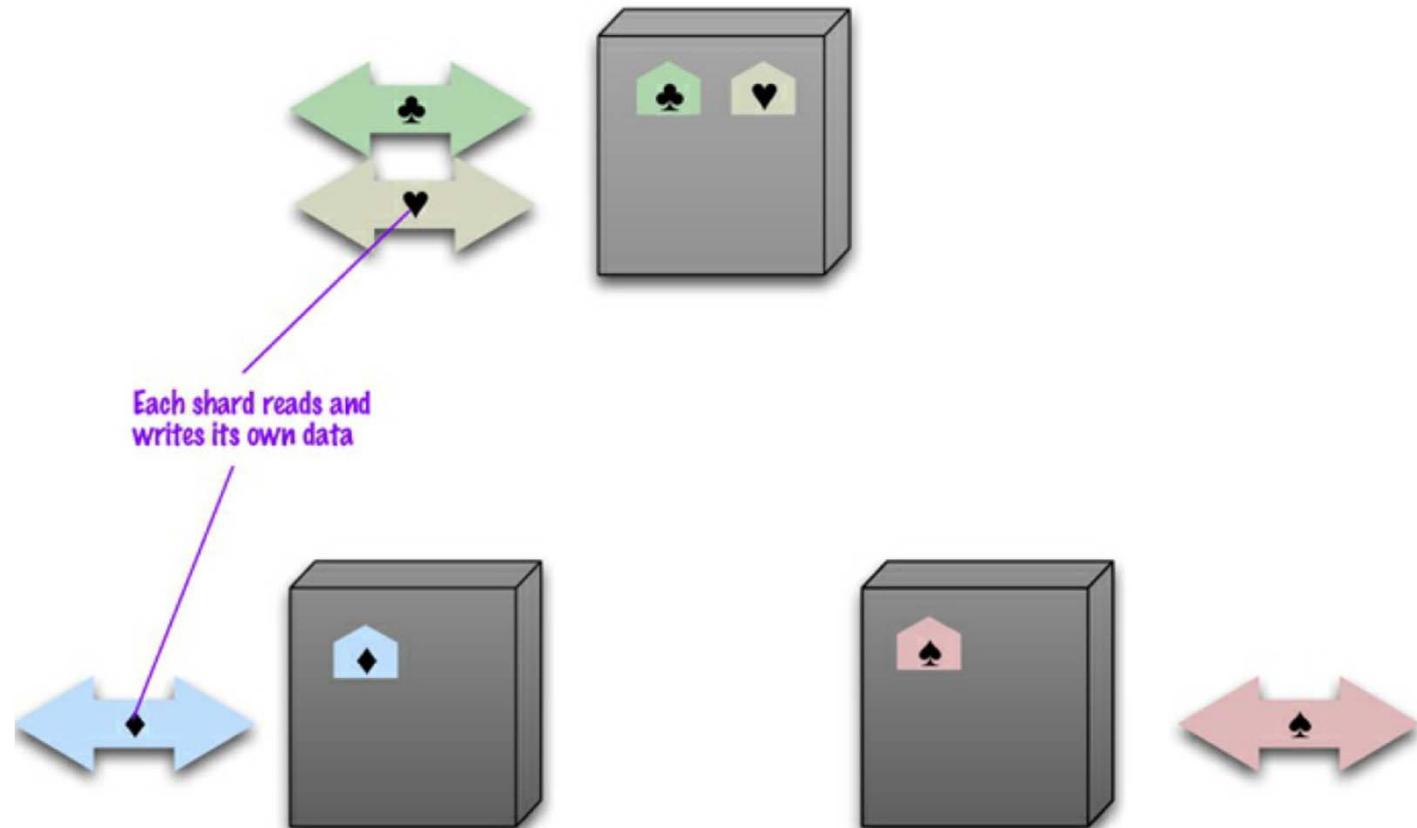
## ■ Un solo servidor

- ▷ No existe la distribución de datos
- ▷ Preferida por ser la **más simple**
- ▷ El uso de **NoSQL** se justificaría solo por cuestiones relacionadas con el **modelo de datos**
  - Ej: Las **BD de Grafos** se suelen utilizar con esta arquitectura
  - Datos agregados que se procesan en el nivel de aplicación y se recuperan juntos



## ■ Varios usuarios accediendo a partes distintas de la base de datos

- ▷ Colocar partes distintas de los datos en nodos distintos
- ▷ Caso ideal: Cada usuario acaba accediendo a un nodo distinto
- ▷ **Escalabilidad horizontal**



Introducción

Sharding

Replicación

Map-reduce



## ■ Localidad de los datos

- ▷ Intentar que los datos que se acceden juntos estén almacenados en el mismo nodo y cerca en el disco
- ▷ ¿Cómo?
  - Colocar datos que se acceden juntos dentro del mismo **agregado**
  - Utilizar el **agregado como unidad de datos para la distribución**  
join precalculado se parece mucho a tener un agregado
- ▷ Incluso mantener juntos agregados que suelen accederse juntos
  - Cuando se conoce el orden más típico de acceso

## ■ Tener los datos lo más cerca posible de donde son utilizados con más frecuencia

- ▷ Ejemplo: Los pedidos de Boston almacenados en el servidor de la costa este
- ▷ **MongoDB:** Uso de Zonas en el Sharding

## ■ Mantener todos los nodos con una carga de datos similar

- ▷ Utilizar **distribuciones uniformes de los datos**
  - Misma probabilidad para todos los nodos

Introducción

Sharding

Replicación

Map-reduce

## ■ Implementación del Sharding en el nivel de aplicación

- ▷ Aplicaciones con código mucho más complejo
- ▷ Necesidad de cambiar la aplicación al reorganizar datos

## ■ Muchas BD NoSQL hacen una gestión automática del Sharding

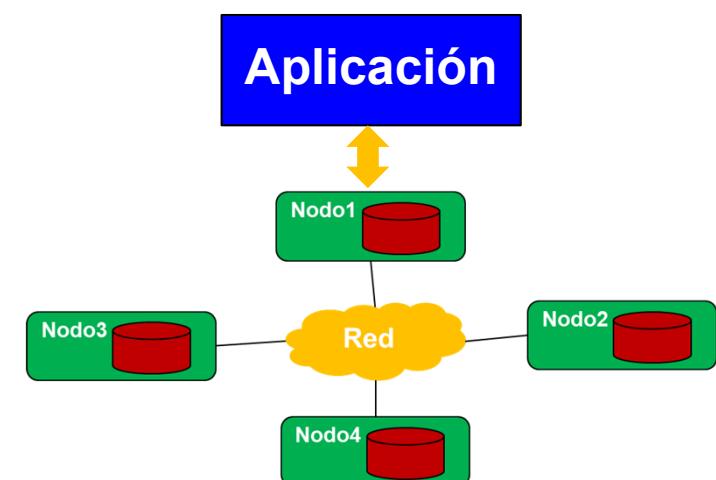
- ▷ Asumen la responsabilidad del particionamiento
- ▷ Simplifican la labor de la aplicación

## ■ Rendimiento

- ▷ Particionamiento mejora lecturas y escrituras
- ▷ Replicación puede mejorar lecturas, pero puede empeorar escrituras

## ■ Fiabilidad

- ▷ Sharding no mejora la disponibilidad
- ▷ Puede haber fallos parciales
- ▷ Aumenta la probabilidad de fallo.



Introducción

Sharding

Replicación

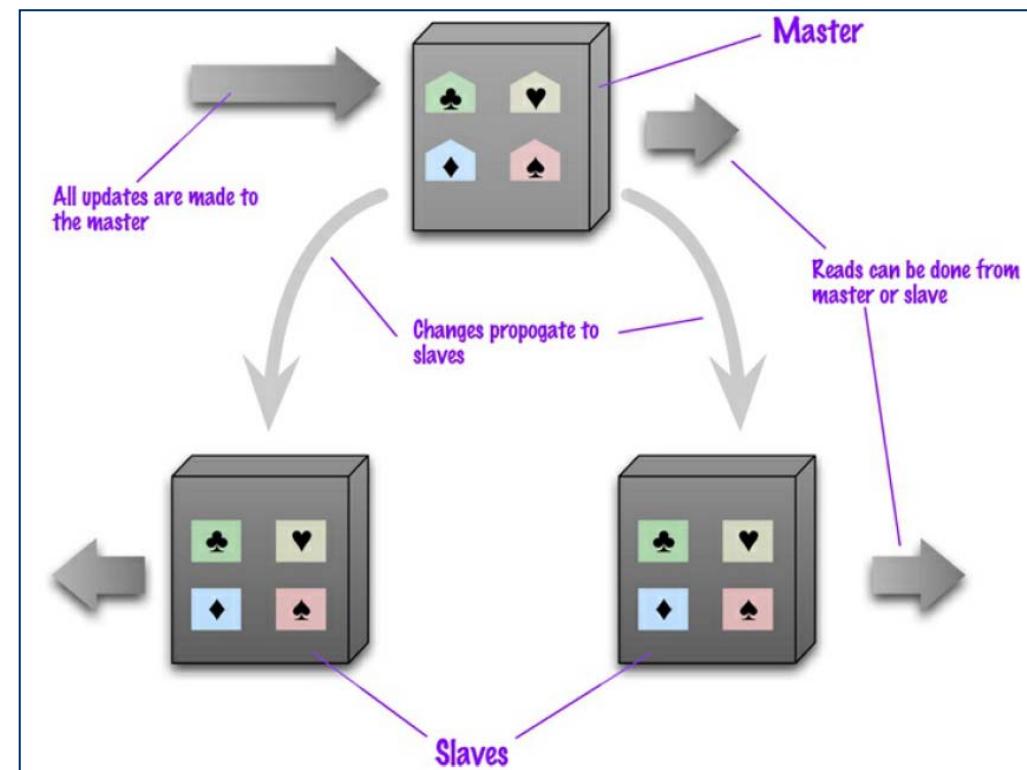


Map-reduce

## ■ Replicación maestro-esclavo

cuanto más sincrono, mas consistencia pero bajando rendimiento y disponibilidad

- ▷ Datos replicados en varios nodos.
- ▷ Un nodo elegido como **maestro** ( o **primario**)
  - \_ Autoridad como fuente de los datos y responsable de su actualización
- ▷ Resto de nodos: **esclavos** (o **secundarios**)
- ▷ Proceso de replicación **sincroniza** los esclavos con el maestro



Introducción

Sharding

Replicación

Map-reduce

## ■ Replicación maestro-esclavo

- ▷ Buena solución cuando la aplicación es intensiva en lecturas (sincronización)
- ▷ **Ventajas**
  - Alta disponibilidad para lecturas (**IMPORTANTE**)
    - Si el maestro falla los esclavos pueden atender peticiones de lectura
  - Sustitución del maestro ante un fallo
    - Si tenemos replicación completa del maestro podemos hacer recuperación en caliente
    - sustitución del maestro manual o automática
- ▷ **Problemas**
  - Maestro **cuello de botella en modificaciones**. Problemas con muchas escrituras
  - **Baja disponibilidad en escrituras.**
  - Problemas de **Consistencia**
    - Lecturas de esclavos distintos pueden dar resultados distintos para el mismo dato
    - Un cliente podría no conseguir leer una actualización recién escrita por el mismo
      - Escritura del maestro y lectura del esclavo antes de sincronizar.



Introducción

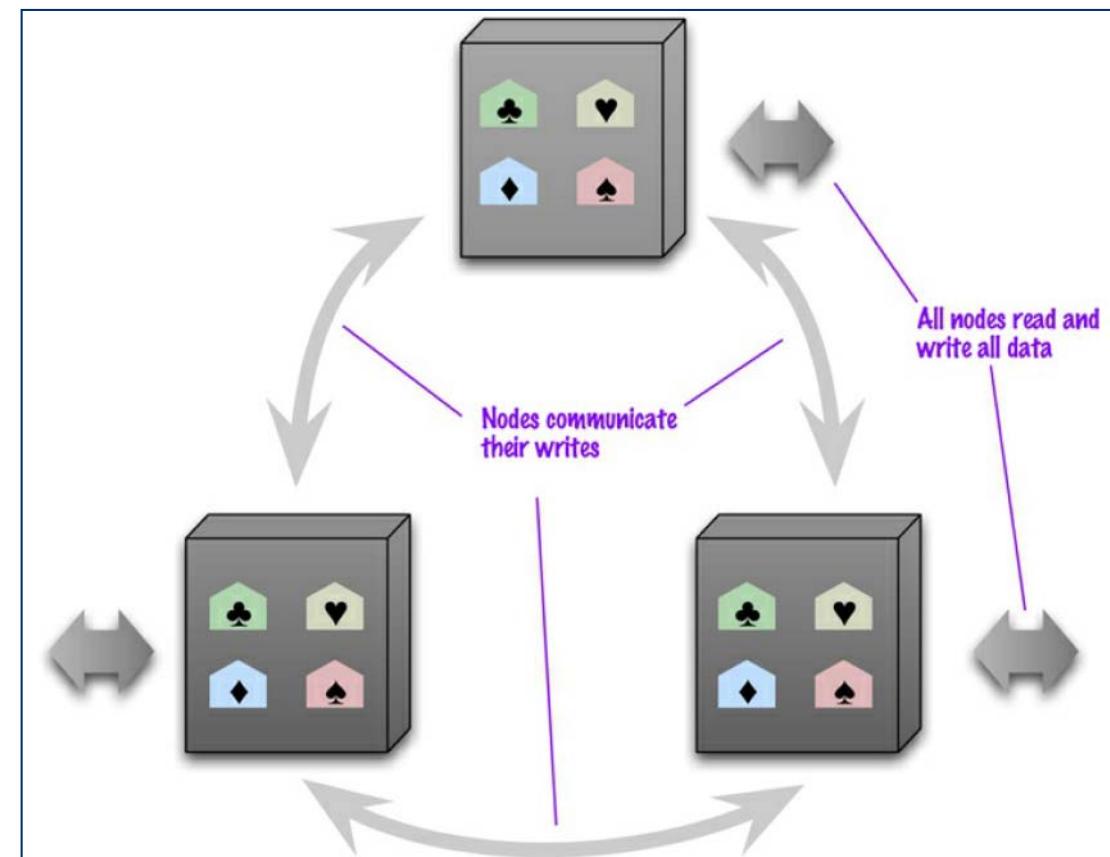
Sharding

Replicación

Map-reduce

## ■ Replicación peer-to-peer

- ▷ **Motivación:** problemas de la arquitectura maestro-esclavo
  - Escalabilidad en escrituras: maestro cuello de botella.
  - Baja disponibilidad: maestro punto único de fallo.
- ▷ **Solución:** Eliminar la distinción entre maestro y esclavos



Introducción

Sharding

Replicación

Map-reduce

## ■ Replicación peer-to-peer

### ▷ Problemas

- Consistencia
  - Dos usuarios modificando el mismo dato al mismo tiempo en dos nodos distintos (conflicto write-write)
    - **Grave!!:** Las inconsistencias read-write crean problemas transitorios, pero las write-write, crean problemas que perduran.
  - Se verán soluciones más detalladas en la parte de consistencia

### ▷ Soluciones generales

- Coordinar las replicas durante las escrituras
  - Coste de red adicional para la coordinación
  - Con actualizar la **mayoría** de forma coordinada sería suficiente en la maestro-escalvo actualizas un nodo, aqui hay que ir a soluciones como esta
- Asumir las inconsistencias e intentar arreglarlas combinando réplicas



Compromiso

Consistencia vs Disponibilidad

Introducción

Sharding

Replicación

Map-reduce

## ■ Combinación de particionamiento y replicación

- ▷ Con replicación **maestro-esclavo**
  - Un maestro único para cada partición
  - Dependiendo de la configuración
    - Elegir maestro y esclavos a nivel de cluster
    - Elegir maestro y esclavos para cada partición
- ▷ Con replicación **peer-to-peer**
  - Muy común en soluciones NoSQL con modelos de tipo **column-family**
  - Replicación con factor 3
    - Cada partición tiene tres copias en tres nodos
    - Si un nodo falla, las particiones de ese nodo se distribuyen entre los demás.

¿MongoDB?

Introducción

Sharding

Replicación

Map-reduce



## ■ Cambios en la forma de procesar

- ▷ Arquitectura centralizada
  - Procesamiento en el cliente
    - (+) Más flexible. (-) Mover datos del servidor al cliente
  - Procesamiento en el servidor
    - (-) Menos cómodo para programar. (+) Más eficiente
- ▷ Arquitectura distribuida
  - Muchas máquinas entre las que repartir la carga de procesamiento
  - Intentar minimizar el tráfico de datos entre nodos

## ■ Patrón Map-Reduce

- ▷ Forma de programar el procesamiento para minimizar tráfico entre nodos
- ▷ Ejemplo de plataforma: **Apache Hadoop**
- ▷ Varias bases de datos NoSQL tienen su propia implementación
- ▷ Basado en operaciones Map y Reduce de lenguajes de programación funcional

Introducción

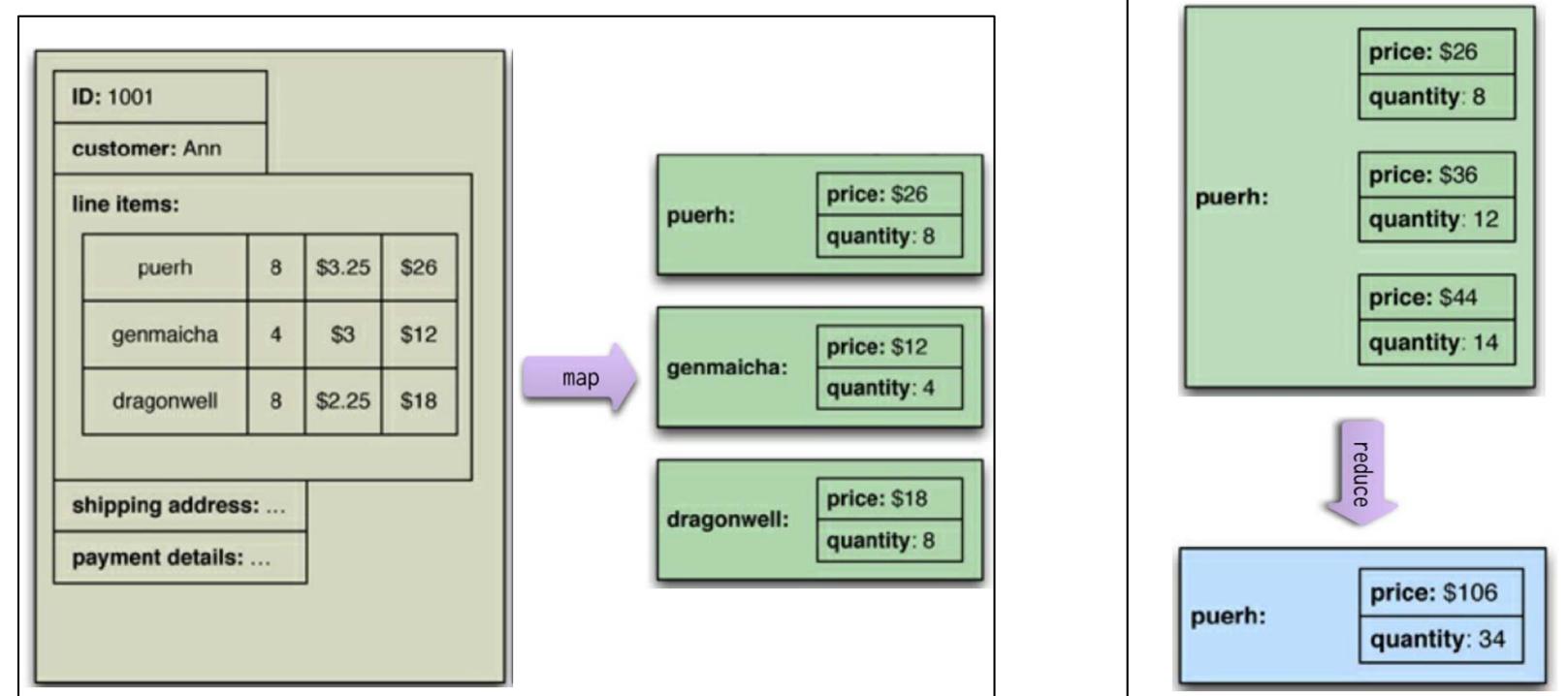
Sharding

Replicación

Map-reduce

## ■ Map-Reduce básico

- ▷ Ejemplo: Obtener los ingresos por cada producto en los últimos 7 días.
  - Necesitamos procesar en todos los nodos
- ▷ **Map:** Para cada agregado genera un conjunto de pares clave-valor.
- ▷ **Reduce:** Sobre el resultado del Map, agrega valores de elementos con la misma clave.



Introducción

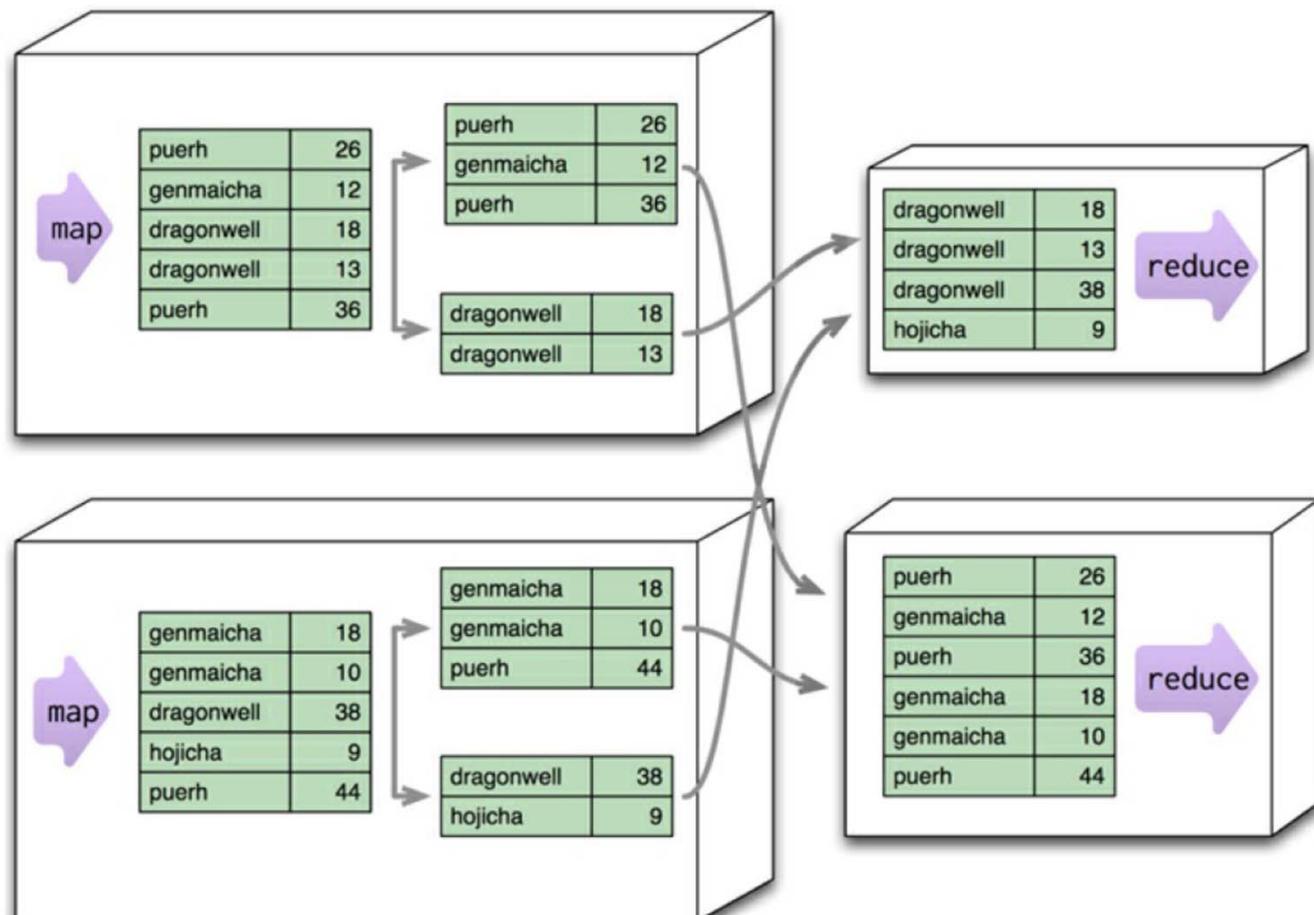
Sharding

Replicación

Map-reduce

## ■ Paralelización de la ejecución de la operación reduce

- ▷ Las operaciones **Map** en cada agregado son independientes con lo que se pueden paralelizar fácilmente
- ▷ Paralelizar **reduce** necesita reparticionar la salida del map (**Shuffling**)



Introducción

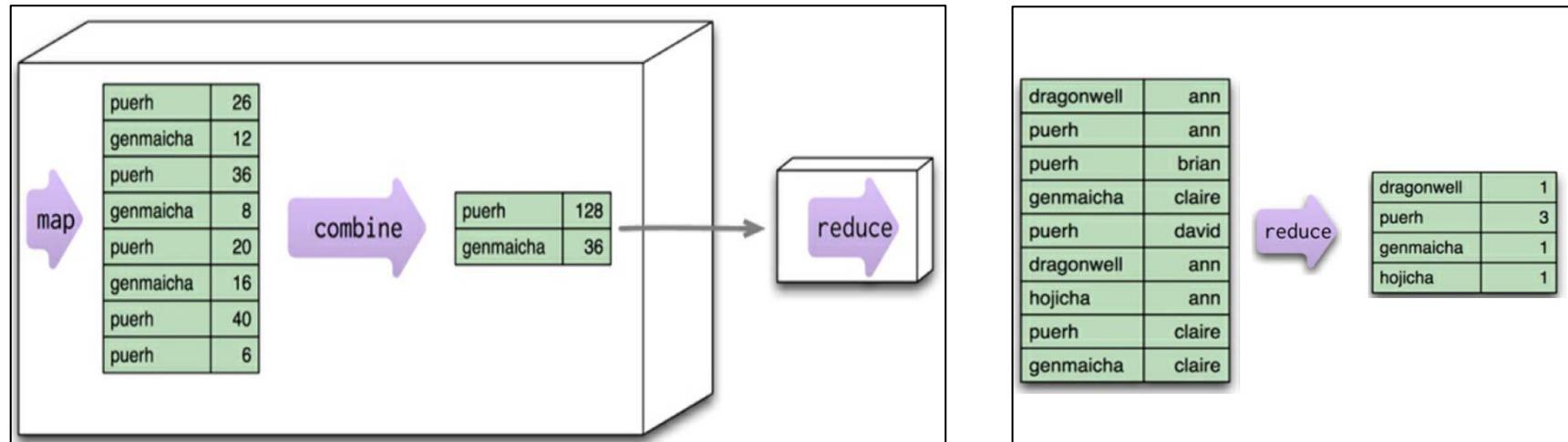
Sharding

Replicación

Map-reduce

## ■ Paralelización de la ejecución de la operación reduce

- ▷ Minimizar el movimiento de datos entre el map y el reduce
- ▷ Reducer combinable
  - Misma forma en la entrada que en la salida
  - Se puede aplicar en cada nodo antes de enviar los datos.
  - Se envían datos ya parcialmente agregados
  - Se podría empezar el reduce incluso antes de terminar el map
- ▷ No todas las operaciones reduce son combinables
  - Ejemplo: Una función que cuenta. Combina sumando y no contando
- ▷ Algunos sistemas solo admiten reducers combinables
  - Si no lo son hay que separar el procesamiento en pasos (pipelines)



Introducción

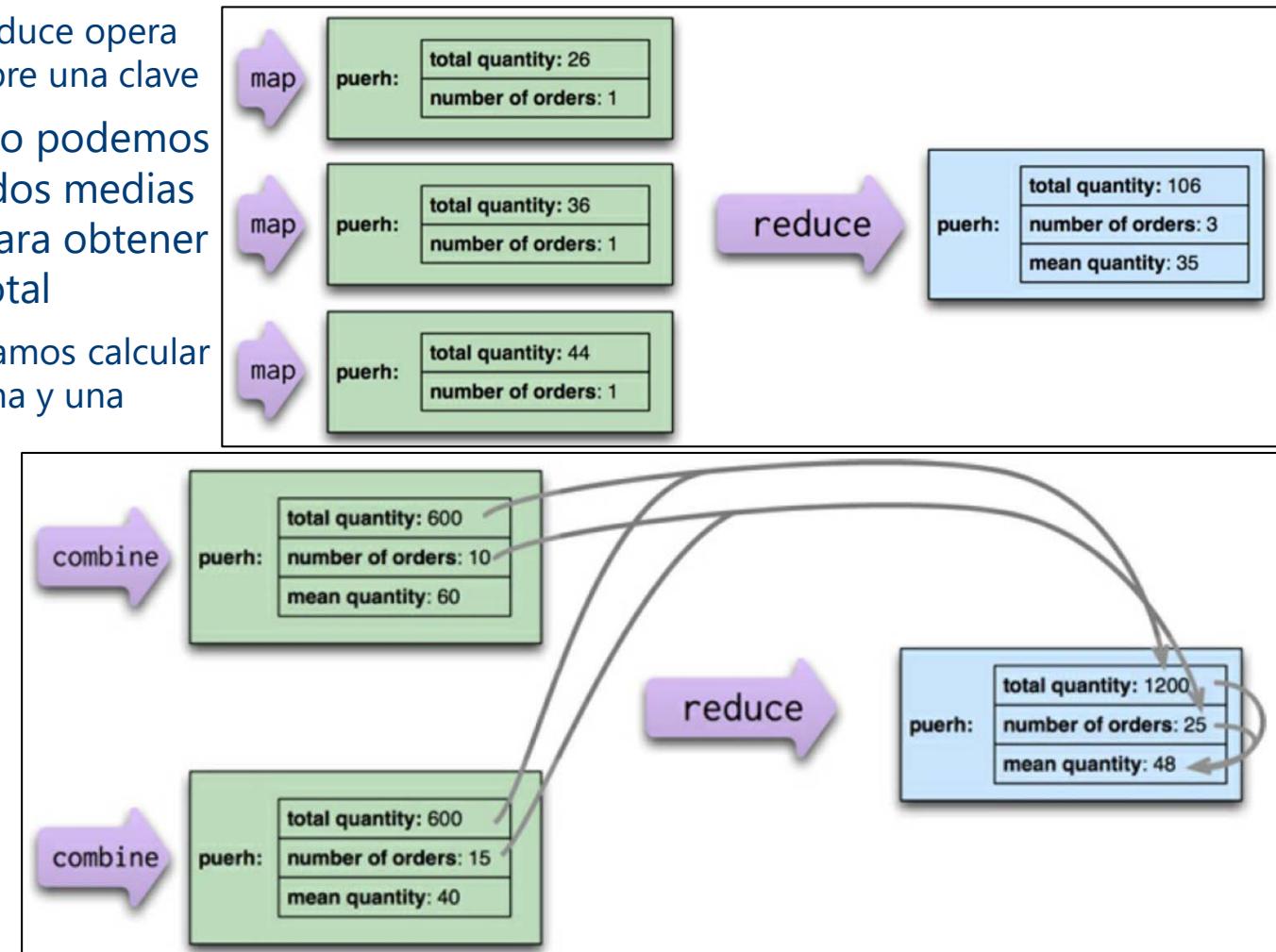
Sharding

Replicación

Map-reduce

## ■ Componiendo cálculos Map-Reduce

- ▷ **Restricciones** en los cálculos
  - Tarea map opera solo sobre un agregado
  - Tarea reduce opera solo sobre una clave
- ▷ **Ejemplo:** No podemos combinar dos medias parciales para obtener la media total
  - Necesitamos calcular una suma y una cuenta



Introducción

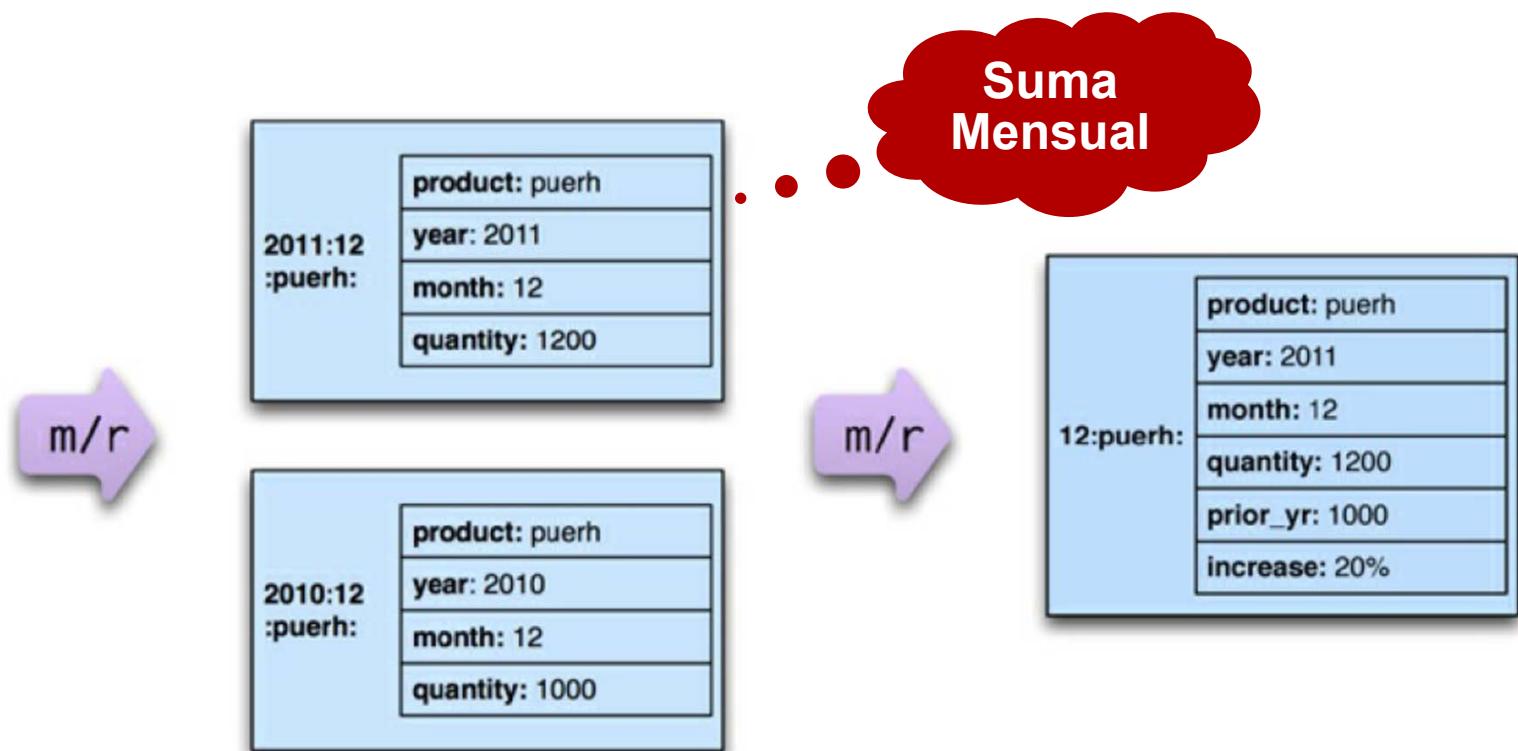
Sharding

Replicación

Map-reduce

## ■ Componiendo cálculos Map-Reduce

- ▷ En casos más complejos necesitaremos descomponer el cálculo en varias etapas Map-Reduce
- ▷ Ejemplo
  - Comparar las ventas de cada mes de 2011 con el mismo mes del año anterior



Introducción

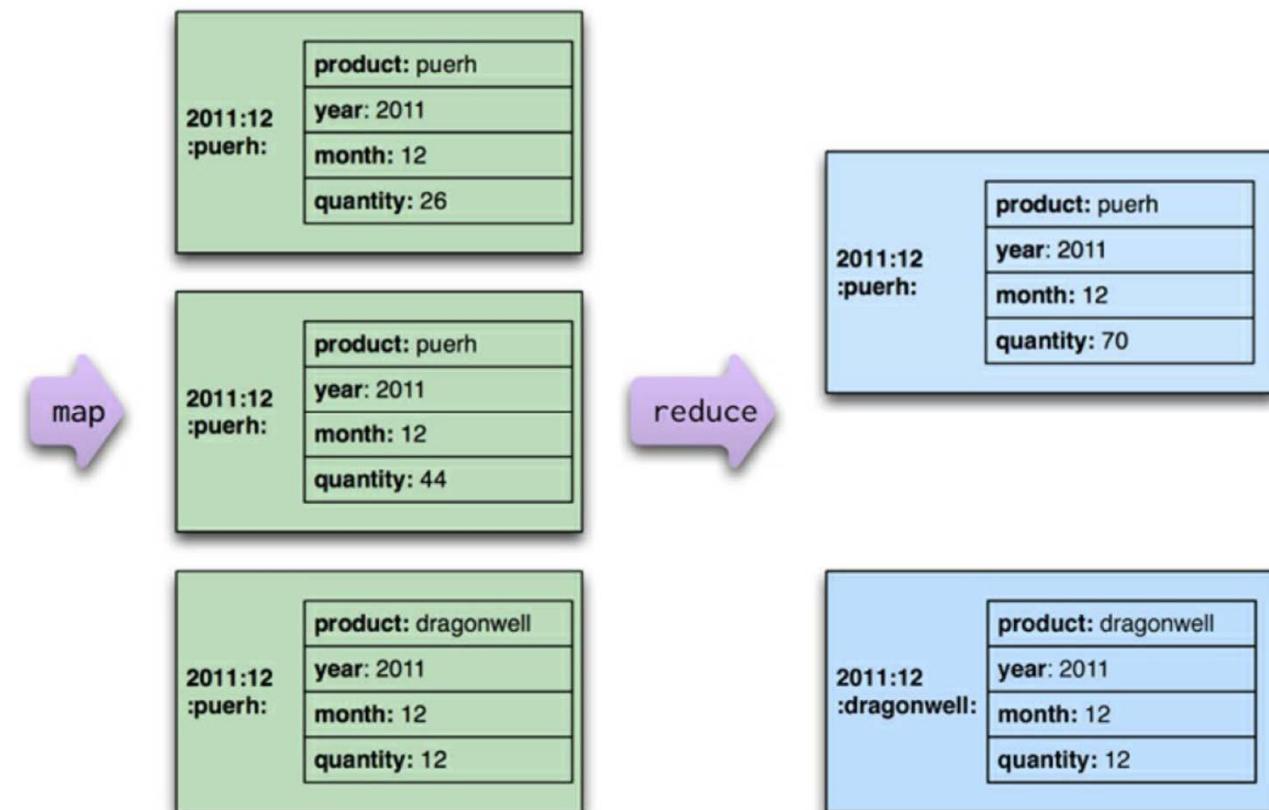
Sharding

Replicación

Map-reduce

## ■ Componiendo cálculos Map-Reduce

- ▷ En casos más complejos necesitaremos descomponer el cálculo en varias etapas Map-Reduce
- ▷ Ejemplo
  - Comparar las ventas de cada mes de 2011 con el mismo mes del año anterior
    - Calculo de la suma mensual (**Map-Reduce**)



Introducción

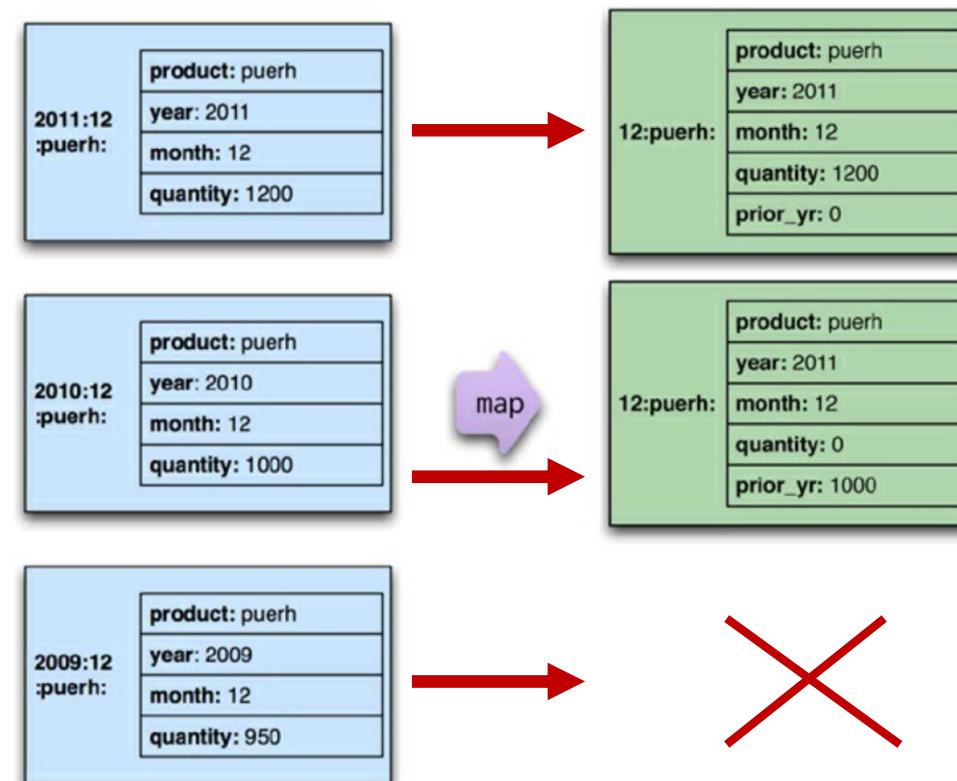
Sharding

Replicación

Map-reduce

## ■ Componiendo cálculos Map-Reduce

- ▷ En casos más complejos necesitaremos descomponer el cálculo en varias etapas Map-Reduce
- ▷ Ejemplo
  - Comparar las ventas de cada mes de 2011 con el mismo mes del año anterior
    - Cálculo de cantidad de año 2011 y cantidad del año previo (**Map**)



Introducción

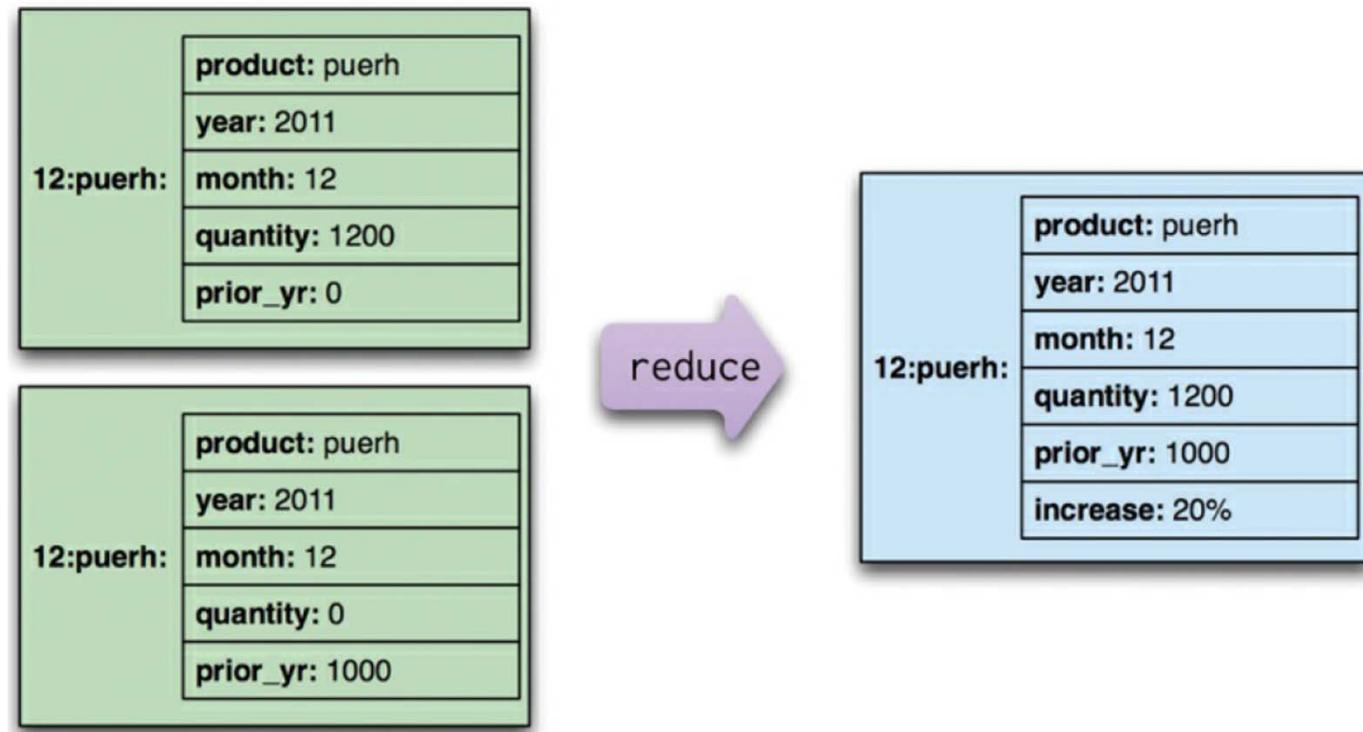
Sharding

Replicación

Map-reduce

## ■ Componiendo cálculos Map-Reduce

- ▷ En casos más complejos necesitaremos descomponer el cálculo en varias etapas Map-Reduce
- ▷ Ejemplo
  - Comparar las ventas de cada mes de 2011 con el mismo mes del año anterior
    - Resultado final (**Reduce**)



Introducción

Sharding

Replicación

Map-reduce

## ■ Componiendo cálculos Map-Reduce

- ▷ Lenguajes especializados en este tipo de programación (ecosistema Hadoop)
  - Pig
  - Hive (Sintaxis similar a SQL)
- ▷ Map-Reduce es un paradigma importante fuera de NoSQL, cuando trabajamos con archivos en sistemas distribuidos

## ■ Map-Reduce Incremental

- ▷ A veces los datos llegan a través de un flujo continuo
  - Empezar cada cálculo desde el principio con todos los datos puede no ser viable
- ▷ Operación Map es fácil de ejecutar de forma incremental
  - Cada tarea Map es independiente de las demás
- ▷ Tareas Reduce dependen de muchos resultados de diferentes Map.
  - Tareas reduce que se aplican a resultado Map que no cambiaron no necesitan ejecutarse de nuevo
  - Reducer combinable y cambios aditivos: solo hay que aplicar reduce sobre cambios.
    - Caso contrario: Usar redes de dependencias para recalcular solo las partes necesarias, dependientes de los cambios en la entrada.

# Bases de Datos NoSQL: Distribución

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

# Bases de Datos NoSQL: Consistencia

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024

- **Introducción**
- **Consistencia en modificaciones**
- **Consistencia en lecturas**
- **Relajando la consistencia**
- **Relajando durabilidad**
- **Quorums**
- **Versiones**

## Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Base de Datos relacionales proporcionan alta consistencia (**ACID**)

- ▷ **Atomicidad:** Se ejecutan todas las instrucciones o ninguna
  - Registro histórico
- ▷ **Consistencia:** En escenario de uso aislado
  - Usuario garantiza la consistencia
  - SGBDs proporciona ayudas para la definición y verificación de restricciones
    - Claves primarias y foráneas, otras restricciones (check), disparadores, etc.
- ▷ **Aislamiento:** En un escenario de varios usuarios en concurrencia
  - SGBDs garantiza la ejecución aislada
    - Ejemplo: **Aislamiento de instantáneas**
- ▷ **Durabilidad:** Cuando una transacción termina, sus efectos perduran
  - Registro histórico + copias

## ■ Bases de datos NoSQL

- ▷ **¿Que clase de consistencia necesita la aplicación?**
  - Consistencia eventual. Teorema CAP

# Consistencia en las modificaciones

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

- Dos usuarios intentan modificar el mismo elemento en paralelo

- ▷ Problema de la modificación perdida
- ▷ Conflicto de tipo write-write

Efecto haber ejecutado T2 se pierde

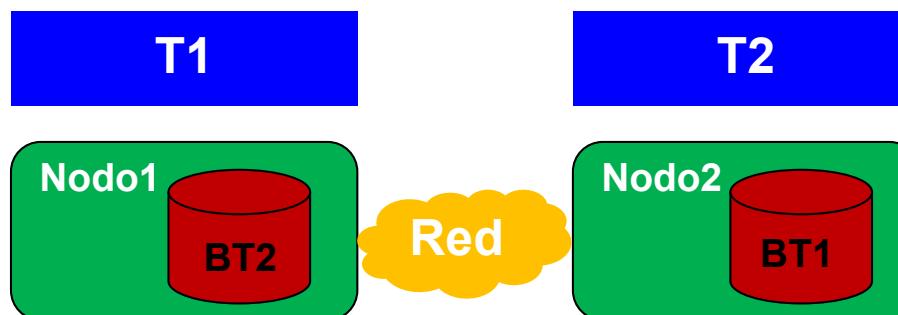
T1	T2
leer(B); escribir(B);	escribir(B);

## ■ Estrategias optimistas y pesimistas

- ▷ **Pesimistas:** Realizan acciones para que no se den casos de inconsistencias
  - Ejemplos: Protocolos basados en **bloqueos** y en **marcas de tiempo**
- ▷ **Optimistas:** Permiten la ejecución normal de la transacción y actúan en el momento del compromiso solo si se detectan problemas
  - Ejemplos: Protocolos basados en **validación** y en **versiones**

## ■ Ejemplo con replicación peer-to-peer

- ▷ Dos réplicas pueden aplicar las mismas modificaciones en distinto orden



T1	T2
leer(B, nodo1); escribir(B, nodo1); sincro(B, nodo2);	escribir(B, nodo2); sincro(B, nodo1)

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

- Control de concurrencia en distribuido
  - ▷ Consistencia secuencial
    - Aplicar las operaciones en el mismo orden en todos los nodos
- Forma optimista de gestionar conflictos de tipo write-write
  - ▷ Almacenar ambos cambios
  - ▷ Indicar que existe un conflicto
  - ▷ Intentar mezclar ambas versiones para obtener una versión consistente
    - Soluciones del ámbito del control de versiones
    - Soluciones muy específicas del ámbito de aplicación
- **Compromiso entre consistencia y eficiencia**
  - ▷ Aproximaciones pesimistas
    - generalmente poco eficientes
      - poco concurrentes
    - pueden tener interbloqueos
- Mayoría de modelos de distribución usan solo una copia para modificar
  - ▷ Simplifica las soluciones que eviten conflictos write-write
  - ▷ Excepción: **Replicación peer-to-peer**

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Conflictos read-write

### ▷ Consistencia lógica

- Asegurarse de que varios elementos de datos tienen consistencia cuando se tratan de forma conjunta
  - A + B en el ejemplo

## ■ Las **Bases de datos de Grafos** suelen dar soporte para ACID

## ■ Bases de datos con agregados

- ▷ Modificaciones atómicas dentro del mismo agregado
- ▷ Transacciones con varios agregados

- Ventana de inconsistencia
  - Tiempo durante el cual se pueden producir lecturas inconsistentes
  - Suelen ser períodos muy cortos (menos de 1 segundo)

T1	T2
leer(A); A:= A – 50; escribir(A);	leer(A) leer(B) visualizar(A+B)
leer(B); B:=B+50; escribir(B);	

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

- Cuando se introduce **replicación** aparecen nuevos potenciales problemas de falta de consistencia

- ▷ Regla general
  - + **redundancia -> - consistencia**

## ■ Ejemplo

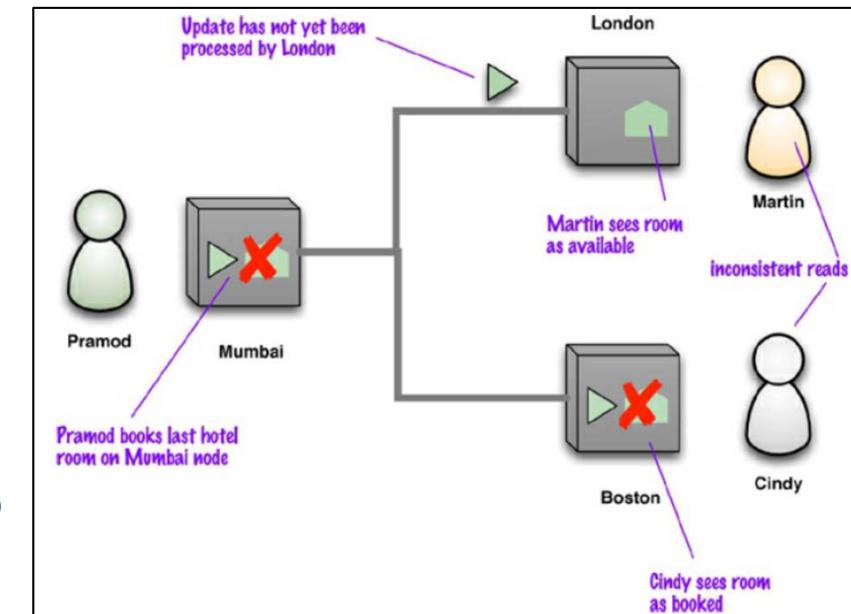
- ▷ Usuario reserva la última habitación de un hotel y se actualiza este hecho en una de las réplicas
- ▷ Dos nuevos usuarios pueden ver dos estados distintos para la habitación, en función de que réplica consulten

## ■ Consistencia de replicación

- ▷ Asegurar que el mismo dato se lee igual desde todas las réplicas
- ▷ Situación temporal.
  - **Consistencia eventual**
- ▷ Problema similar en las memorias cache

## ■ Ajustar el nivel de consistencia deseado a la aplicación

- ▷ Mayoría de operaciones pueden realizarse con niveles bajos



Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Problemas de consistencia con el mismo usuario

- ▷ Ejemplo: Publicar comentarios en un Blog
  - Al refrescar la lista un comentario publicado anteriormente puede no estar
- ▷ Consistencia de tipo **read-your-write**
- ▷ Solución: **Consistencia de sesión**
  - Read-your-write durante la sesión de usuario

## ■ Implementación de la consistencia de sesión

- ▷ **Sticky session:** La sesión se vincula a un nodo
  - Al mantener read-your-write en el nodo se mantiene en la sesión
  - Problema: Reduce la capacidad del balanceador de carga
- ▷ **Version stamps:** Uso de marcas de versiones
  - Asegurar que cada operación actúa sobre última versión
    - En caso contrario esperar para poder responder
- ▷ **Problema con Sticky session y replicación maestro-esclavo**
  - Se puede leer de los esclavos, pero se modifica en el maestro siempre
  - **Solución 1:** Modificar en el esclavo y hacerlo responsable de actualizar el maestro
  - **Solución 2:** Mover la sesión al maestro mientras se realiza la modificación
    - La sesión vuelve al esclavo cuando se actualizan los cambios en el esclavo

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Consistencia en la interacción con el usuario

- ▷ Ejemplo: Reserva de la habitación del hotel
  - Dos intentos de reserva de forma concurrente
  - Cuando se pulsa el botón de reservar, el estado puede haber cambiado ya y no estar disponible
  - **Solución:** dividir la transacción en dos partes
    - Parte 1: Interacción con el usuario
    - Parte 2: Finalización de la transacción
      - Ejecución atómica en una transacción de BD
      - Verificación de la consistencia de los datos

Introducción

Modificaciones

Lecturas

**Relajando  
Consistencia**



Relajando  
Durabilidad

Quorums

Versiones

- A veces es necesario sacrificar la consistencia
  - ▷ Para mejorar otras propiedades
    - Solución de compromiso
    - Dependiente de la aplicación
- También en sistemas centralizados
  - ▷ Recordar niveles de consistencia en SQL
    - **Secuenciable**: Ejecución secuencial (casi siempre...)
    - **Lectura repetible**: Lectura de datos comprometidos y lectura repetible.
    - **Lectura comprometida**: Lectura de datos comprometidos.
    - **Lectura no comprometida**: Lectura de datos no comprometidos.
- Muchos sistemas evitan el uso de transacciones completamente
  - ▷ MySQL muy popular cuando no tenía gestión de transacciones
    - Aplicaciones web
  - ▷ Sitios web muy grandes con necesidad de sharding (eBay)
  - ▷ Necesidad de interacción con sistemas remotos

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia



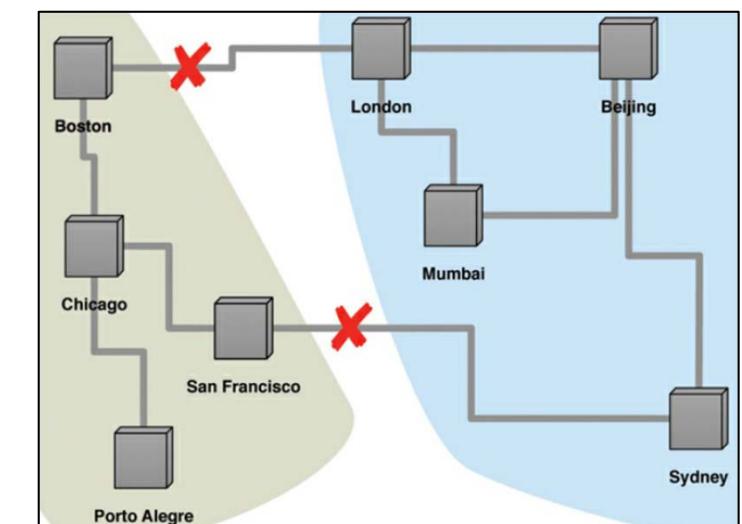
Relajando  
Durabilidad

Quorums

Versiones

## ■ Teorema CAP

- ▷ Propuesto por Eric Brewer en el 2000 (conjetura de Brewer). Prueba formal por Seth Gilber y Nancy Lynch en 2002.
- ▷ **Teorema**
  - Un sistema solo puede tener dos de las tres siguientes características
    - Consistencia
    - Disponibilidad (Availability)
      - Significado aquí un poco distinto al habitual
      - Si podemos comunicar con un nodo del cluster, entonces podemos leer y escribir datos.
    - Tolerancia al particionamiento
      - El cluster puede sobrevivir a fallos de comunicación entre los nodos que lo separan en varias partes.



Introducción

Modificaciones

Lecturas

**Relajando  
Consistencia**



Relajando  
Durabilidad

Quorums

Versiones

## ■ Teorema CAP

- ▷ Sistema centralizado
  - Consistencia y Disponibilidad (CA)
  - No tolerancia al particionamiento: una sola máquina no puede particionar.
  - Si el nodo está levantado, entonces está disponible
- ▷ Cluster
  - Un cluster puede particionarse en dos pedazos no conectados
  - Es posible tener un cluster CA
    - Si hay una partición del cluster, debemos pararlo por completo para que no pueda usarse
      - Definición habitual de disponibilidad significaría falta de disponibilidad
      - En el contexto de CAP **disponibilidad** significa: **Cada petición recibida por un nodo que no falla debe ser respondida.**
    - Detener todos los nodos cuando hay una partición es muy costoso
      - Incluso detectar las particiones a tiempo es un problema en sí mismo

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia



Relajando  
Durabilidad

Quorums

Versiones

## ■ Teorema CAP

### ▷ Como se interpreta el teorema en la práctica

- Si un sistema pueden sufrir particiones (sistema distribuido), hay un **compromiso** entre la **consistencia** y la **disponibilidad**.
- **Ejemplo**
  - Dos intentos de reserva de la misma habitación concurrentes en sistema con replicación peer-to-peer
  - Nodo1 y Nodo2 necesitan comunicarse para serializar ambas peticiones
    - **Incrementa la consistencia**
    - **Baja disponibilidad:** Si la red falla, ninguno de los nodos puede responder
  - Mejorar la disponibilidad: **Elegir un maestro** y dirigir todas las modificaciones al maestro
  - Mejorar más la disponibilidad: Aceptar peticiones incluso si la red falla.
    - Perdemos consistencia: ambos podrían reservar la última habitación
    - Podríamos admitir esto: overbooking, mantener habitaciones libres para estos casos

U1



U2



Introducción

Modificaciones

Lecturas

Relajando  
Consistencia



Relajando  
Durabilidad

Quorums

Versiones

## ■ Teorema CAP

### ▷ Como se interpreta el teorema en la práctica

- Si un sistema pueden sufrir particiones (sistema distribuido), hay un **compromiso** entre la **consistencia** y la **disponibilidad**.
- **Ejemplo: Carro de la compra (Amazon Dynamo)**
  - Siempre se permite escribir en el carro de la compra: incluso si falla la red.
  - Un proceso final puede unir todos los carros generados en uno
    - Si hay problemas el usuario toma el control para verificar los carros antes de realizar el pedido

### ▷ Conclusión

- Programadores buscan **consistencia**. Pero se puede relajar para conseguir **disponibilidad** y mejorar **eficiencia**
  - Normalmente se necesita **conocimiento del dominio de aplicación**
- **Consistencia de lectura**
  - Importancia del dato para la aplicación: inversión bolsa vs noticia en blog
    - Tolerancia a las lecturas viejas
    - Duración de la ventana de inconsistencia

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia



Relajando  
Durabilidad

Quorums

Versiones

## ■ Teorema CAP

### ▷ NoSQL: Propiedades BASE

- Propiedades (sin definición clara): Basically Available, Soft State, Eventual Consistency
- Compromiso entre ACID y BASE (soluciones intermedias)

### ▷ Compromiso entre **consistencia** y **latencia** (más claro)

- Mejoramos consistencia introduciendo más nodos en la interacción
  - **Ejemplos:**
    - Si al **leer**, leemos de 5 nodos estaremos más seguros de la consistencia de la lectura que si leemos de solo 2
    - Si al **escribir**, no devolvemos el control hasta escribir en 5 nodos tendremos una escritura más consistente que si escribimos en solo 2 y confiamos en que el resto se escribirán de forma asíncrona más tarde.
  - **Empeoramos eficiencia:** Cada nuevo nodo incrementa el tiempo de respuesta
- **Disponibilidad como límite de la latencia** que podemos tolerar en la aplicación
  - Esta idea relaciona **disponibilidad** con **eficiencia**
  - Cuando la latencia es muy alta, nos rendimos y asumimos que la operación no puede terminar.

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Atomicidad: clave para la consistencia

- ▷ Definición de unidades atómicas de ejecución (**Transacciones**)

## ■ ¿Se puede relajar la durabilidad?

- ▷ Perder las actualizaciones de algunas transacciones terminadas con éxito
- ▷ **Compromiso entre durabilidad y rendimiento**
  - Pérdida de datos en memoria si hay un fallo en el sistema
  - Perder información de sesión en una web puede no ser muy importante

## ■ Especificar en cada escritura si necesitamos durabilidad o no

- ▷ Podríamos perder algunos datos insertados por un sensor

## ■ Durabilidad de replicación

- ▷ Ejemplo de problema por falta de durabilidad de replicación
  - Maestro falla. Esclavo busca otro maestro. Algunos datos del maestro caído no se habían replicado a un esclavo. Esclavo sigue recibiendo cambios a través del nuevo maestro. Maestro antiguo se recupera. Podemos encontrar conflictos
- ▷ Esperar a tener las réplicas generadas para comprometer transacción **mejora la durabilidad de replicación.**
  - **Empeora rendimiento** de las escrituras
  - **Disminuye la disponibilidad** (aumenta la probabilidad de fallo de la escritura)

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

- Soluciones reales para la consistencia o durabilidad
  - ▷ Búsqueda del nivel deseado en la aplicación. **¿Cuantos nodos implicar?**
    - Si tenemos tres réplicas, ¿Cuántas debemos involucrar para obtener una respuesta?
      - ¿Llega con la mayoría?, dos en este caso
  - ▷ Si tenemos escrituras en conflicto, solo una puede haber realizado la mayoría de escrituras
  - ▷ **Quorum de escritura**
    - $W > N/2$ 
      - **W**: Nodos que participan en la escritura
      - **N**: Nodos totales involucrados en la replicación (**factor de replicación**)
  - ▷ **Quorum de lectura**
    - ¿Cuántos nodos tenemos que consultar para asegurar que tenemos la copia más reciente?
      - $N = 3, W = 2$ . Necesitamos leer dos nodos para asegurar que leemos el último dato.  $R = 2$ .
      - $N = 3, W = 1$ . Necesitamos leer tres nodos.  $R = 3$ .
    - $R + W > N$

Asumiendo Replicación peer-to-peer

Maestro-esclavo llegaría con  
interactura con el maestro

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

- No confundir el factor de replicación con el número de nodos del cluster
  - ▷ Podemos estar realizando 3 copias de cada dato en un cluster de 12 nodos
- Factor de replicación 3 es muy común
  - ▷ Permite un fallo de un nodo mientras no se realiza una nueva réplica
- Ajustar W y R a las necesidades de la aplicación
  - ▷ Lecturas consistentes y rápidas
    - W tiene que ser alto, para que R sea bajo
    - Las escrituras serán lentas
  - ▷ Escrituras rápidas y consistencia baja
    - W tiene que ser bajo.
    - Para que R sea bajo también tenemos que usar un N bajo
      - La consistencia será baja por lo tanto



Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

**Versiones**

- BD NoSQL suelen proporcionar atomicidad dentro del agregado
- Transacciones del sistema no funcionan bien con la interacción con el usuario
  - ▷ Uso de marcas de versiones (comprobar si la versión leída ha cambiado)
- **Transacciones de negocio y de sistema**
  - ▷ Ya hemos visto la diferencia
  - ▷ Transacción de negocio dividida en dos partes
    - Parte I: Interacción con el usuario
    - Parte II: Transacción de sistema (bloqueo de recursos durante poco tiempo)
  - ▷ Técnicas de concurrencia offline: Ejemplo **Optimistic Offline Lock**
    - Leer de nuevo todo para comprobar si ha cambiado
    - Utilizar una **marca de versión** para comprobarlo
    - **Implementación** de las marcas
      - Contadores
      - secuencias aleatorias
      - Hashing de los elementos
      - Marcas de tiempo
    - Utilización para proporcionar **consistencia de sesión**

CouchDB combina  
un Contador con un  
Hash del elemento

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Marcas de versión en múltiples nodos

- ▷ Marcas de versión funcionan bien en **un nodo o en maestro-esclavo**
  - Solo un nodo controla las versiones
- ▷ En **peer-to-peer** dos nodos pueden tener valores distintos por distintas razones
  - Actualización llegó a uno pero no al otro
    - Elegir la más reciente
  - Puede haber otro tipo de inconsistencia en la modificación (necesita solución)
- ▷ **Forma más simple de implementación: Contador**
- ▷ Si tenemos más de un maestro, se necesita una solución más avanzada
  - Todos los nodos tengan historial de versiones
  - Cliente mantiene versiones / servidor responde con versiones
  - No se usa en NoSQL. Típico en sistemas de control de versiones
- ▷ **Uso de timestamps**
  - Problemático: necesita consistencia temporal entre nodos
  - Si las modificaciones se hacen con mucha rapidez puede haber problemas

Introducción

Modificaciones

Lecturas

Relajando  
Consistencia

Relajando  
Durabilidad

Quorums

Versiones

## ■ Marcas de versión en múltiples nodos

### ▷ **Vector Stamp** (solución muy usada)

- Un contador por nodo en un vector
  - (3, 5, 6)
  - Si el segundo nodo modifica el dato, el vector cambia a (3, 6, 6)
- Cuando dos nodos se comunican se sincronizan los vectores
  - Diferentes formas de sincronización
- Un vector será más reciente que otro, si todos los contadores son mayores o iguales
- Si ambos vectores tienen un valor mayor que el otro entonces existe un conflicto de tipo write-write
- Si un valor de un vector falta se trata como cero
  - Permite añadir nuevos nodos
- Muestra las inconsistencias, pero no las resuelve
- La resolución de inconsistencias depende bastante del dominio de aplicación
  - Compromiso consistencia / latencia
    - Asumir que los fallos de red harán que el sistema falle
    - O por lo contrario detectar y tratar las inconsistencias

# Bases de Datos NoSQL: Consistencia

**José R.R. Viqueira**

Centro Singular de Investigación en Tecnologías Intelixentes (CITIUS)  
Rúa de Jenaro de la Fuente Domínguez,  
15782 - Santiago de Compostela.

**Despacho:** 209

**Telf:** 881816463

**Mail:** [jrr.viqueira@usc.es](mailto:jrr.viqueira@usc.es)

**Skype:** jrviqueira

**URL:** <https://citius.gal/team/jose-ramon-rios-viqueira>

Curso 2023/2024