Bases de datos NoSQL[1]

CENTIC, 2018

Jesús García Molina, Diego Sevilla Ruiz

Facultad de Informática
Universidad de Murcia

{jmolina,dsevilla}@um.es

Junio de 2018

[1] https://github.com/dsevilla/centic18

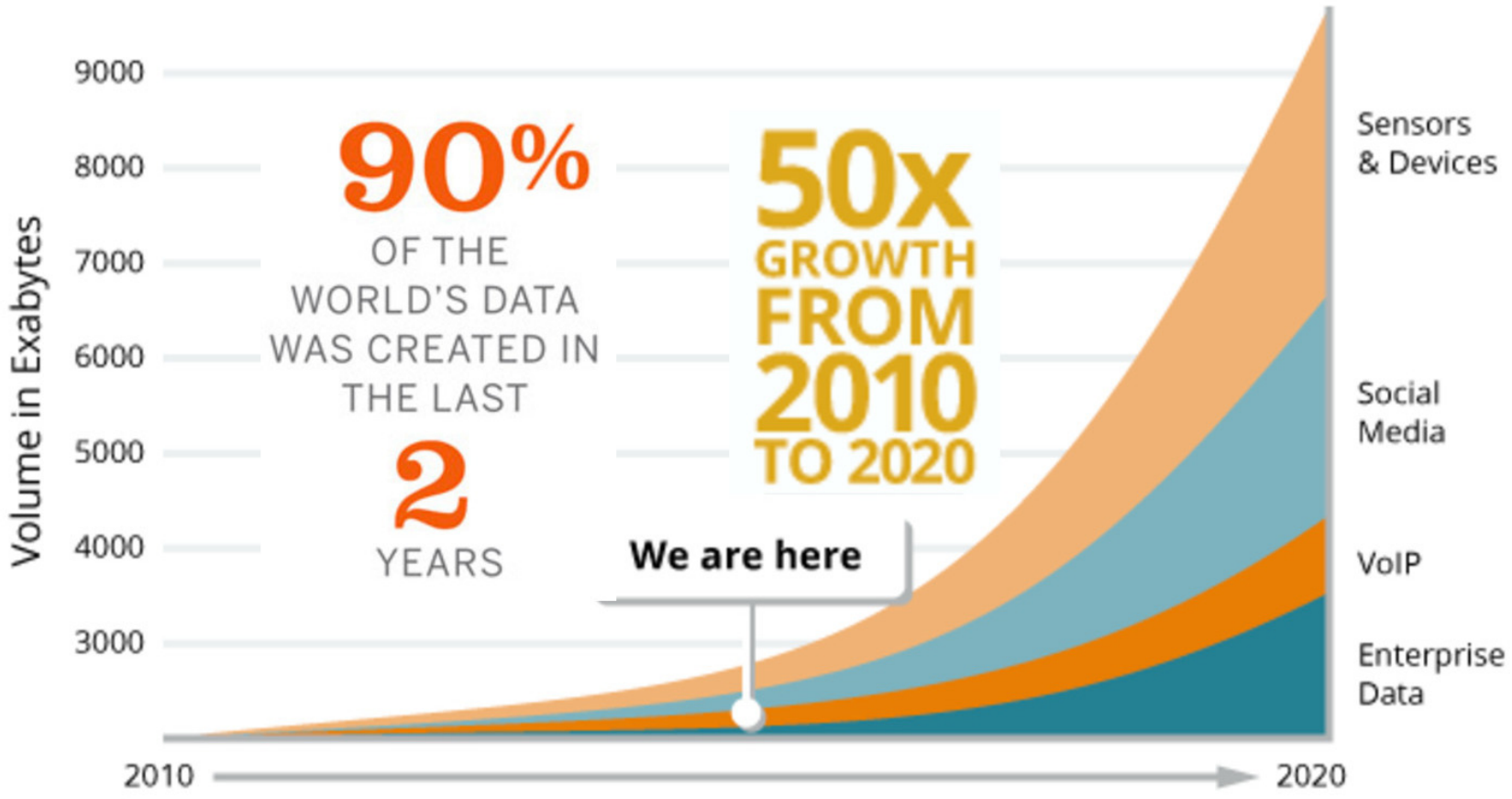
Introducción a NoSQL

- NoSQL $\Rightarrow$ *hashtag* llamativo que se eligió para una conferencia en 2009 (Johan Oskarsson de Last.fm)
- Ahora se asocia a cientos de bases de datos diferentes, que se han clasificado en varios tipos (las veremos después), caracterizadas por **no usar SQL** como modelo de datos
- NoSQL $\Rightarrow$ *Not Only SQL* (no sólo SQL)

NoSQL – ¿Por qué se plantearon?

1. **Mayor escalabilidad horizontal**
   - conjuntos de datos muy muy grandes
   - sistemas de alto volumen de escrituras (*streaming* de eventos, aplicaciones sociales)
2. **Demanda de productos de software libre** (crecimiento de las *start-ups*)
3. **Consultas especializadas** no eficientes en el modelo relacional (JOINs)
4. **Expresividad, flexibilidad, dinamismo**. Frustración con **restricciones** del modelo relacional

NoSQL – ¿Por qué se plantearon? (II)

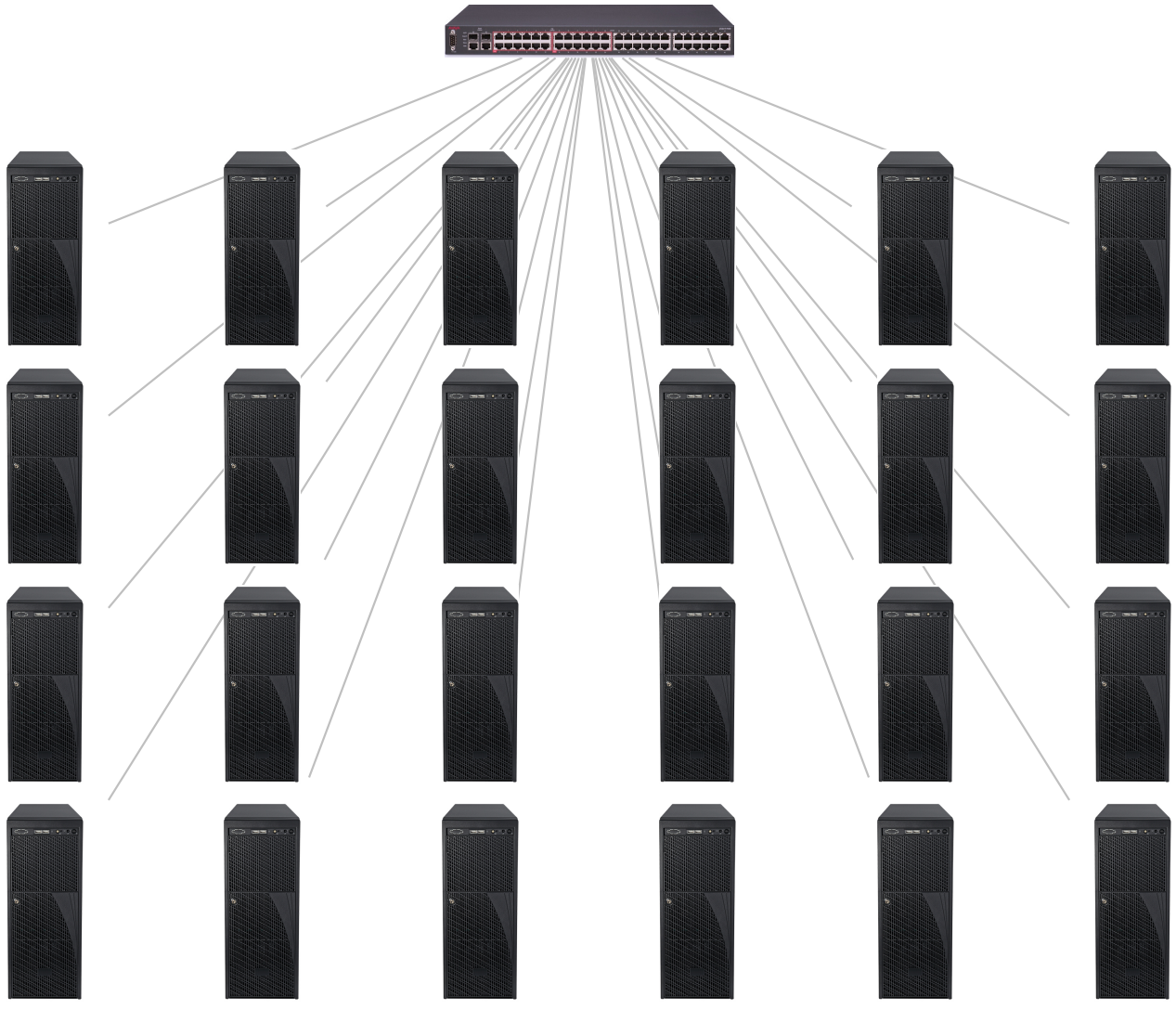


Source: https://es.slideshare.net/sfamilian/visual-design-with-data-feb-2017/

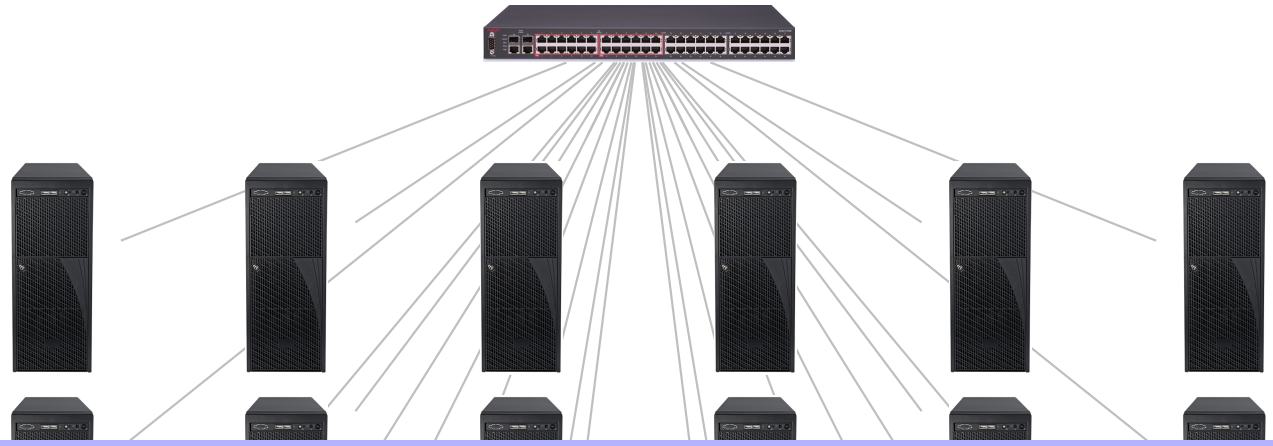
NoSQL: Características

- No se basan en SQL
- Modelos de datos más ricos
- Orientadas a la **Escalabilidad**
- Generalmente no obligan a definir un esquema $\Rightarrow$ *Schemaless*
- Surgidos de la comunidad para solucionar problemas, y muchas de ellas son **libres/*open source***
- Diseñadas $\Rightarrow$ **procesamiento distribuido**
- Principios funcionales $\Rightarrow$ **MapReduce**
- Generalmente implementan **consistencia relajada**

Cambio de perspectiva: Red



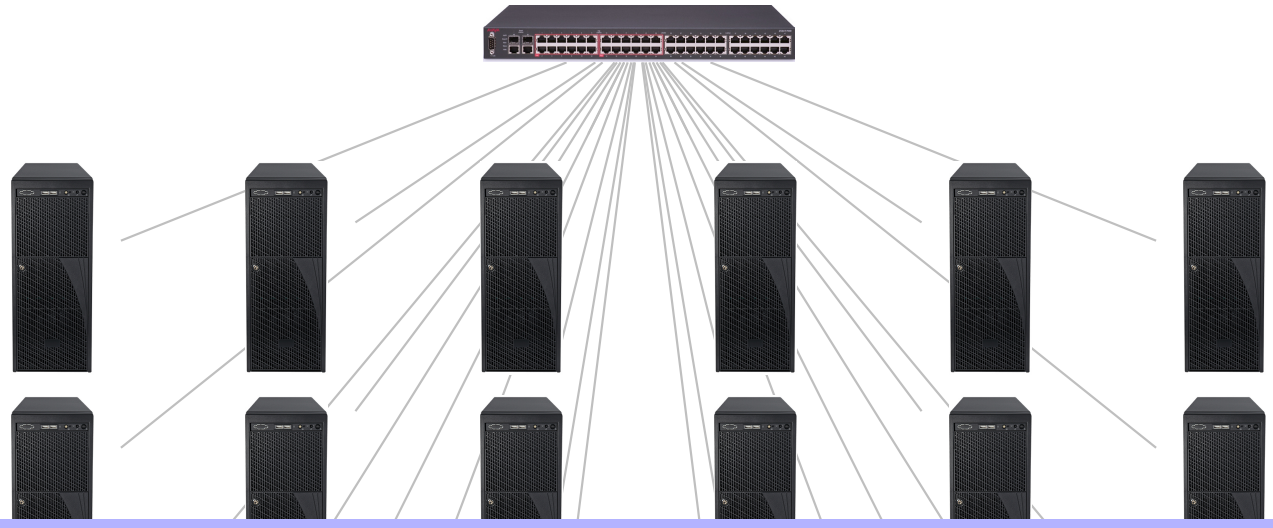
Cambio de perspectiva: Red



Procesamiento distribuido

- Necesidad de **paralelización máxima**
- **Escalabilidad**
- Explotación de la **localidad de los datos**:
  - Datos producidos en cada nodo se utilizan en siguientes iteraciones
  - Cada nodo puede hacer de servidor para recibir datos

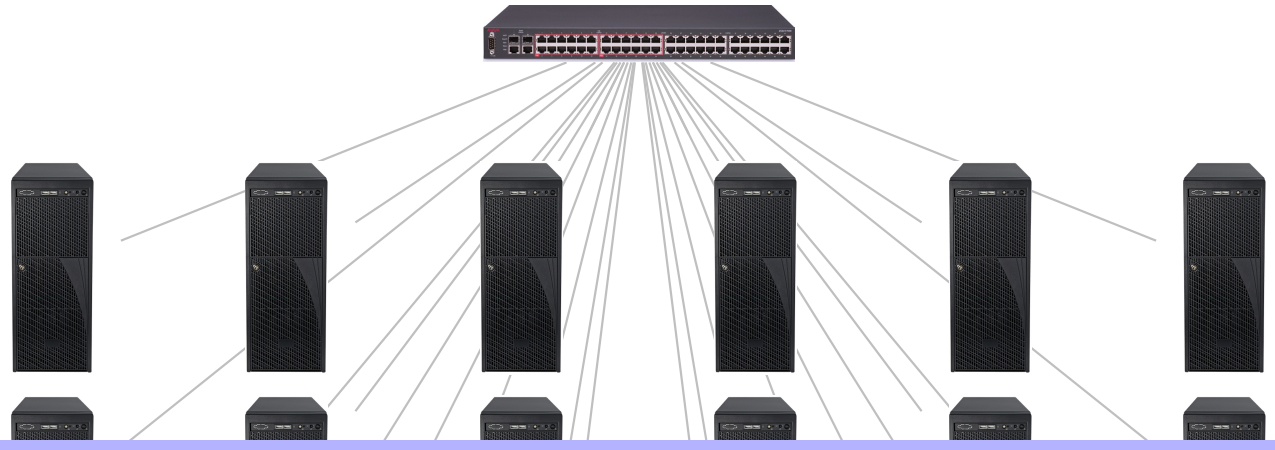
Cambio de perspectiva: Red



Procesamiento distribuido

- Vuelta al modelo funcional inherentemente paralelo: (e.g. **Map-Reduce**)
- Almacenamiento distribuido: (e.g. **HDFS**)
- Coordinación distribuida: (e.g. **Zookeeper**)

Cambio de perspectiva: Red



Modelo de datos

- ¿Si se pudiera ver como un **GRAN ARRAY**?
  - Cada nodo almacenaría una parte del array
  - Búsqueda aleatoria **muy rápida** (árboles B)
  - Uso de **objetos complejos** (p. ej. **documentos JSON**), para mantener la **localidad espacial de datos relacionados** (+ después)
  - Transacciones limitadas al **objeto complejo**

Schemaless

- Las BBDD NoSQL (en general) **no requieren de un esquema**

SCHEMALESS

- **Flexibilidad**: Posibilidad de almacenar documentos con una estructura diferente
  - Tratar información incompleta
  - Evolucionar la base de datos/esquema
  - Añadir nuevas características a las aplicaciones

Schemaless (II)

schema-on-write	$\Rightarrow$	*schema-on-read*
SQL		NoSQL
Los datos conforman cuando se **escriben**		Los datos leídos conforman a un **esquema implícito**
Tipado estricto (estático)		*Duck-Typing* (dinámico)
Datos **homogéneos**		Datos **heterogéneos**
Proceso analítico a través de **consultas**		*Use as read*

Schemaless (III)

- **Ejemplo**: Añadir el campo `first_name` a partir del campo `name`
- Los nuevos objetos se crean con el nuevo formato
- A la hora de leerlos, se puede hacer:

```
if (user && user.name && !user.first_name) {
    // Docs anteriores a 2013 no tienen first_name
    user.first_name = user.name.split(" ")[0];
}
```

Schemaless (IV)

- En SQL:
  - Puede ser un proceso muy costoso
  - Procesa toda la tabla
  - *Locking*
  - Puede obligar a parar las aplicaciones

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name =
    substring_index(name, ' ', 1);
```

Schemaless (V)

¿Cuándo es apropiado *schemaless*?

- **Objetos heterogéneos**
- Estructura de los datos **impuesta externamente**
- Si intuimos que los datos **cambiarán en el futuro**

Modelado de datos en NoSQL

- En general ofrecen más **flexibilidad** en el modelado de datos:
  - **Documentos**: Posibilidad de **agregación** (además de referencia)
  - **Grafos**: Gran número de relaciones entre elementos
- **Optimización guiada por las consultas**
- Es "barato" **duplicar (desnormalizar)** los datos si con ello se consigue **mayor eficiencia de acceso**

Representación relacional de un CV

Kleppmann, 2016. *Designing Data Intensive Applications*

http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…
457	251	Co-founder, Chairman	Microsoft

education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {
      "job_title": "Co-chair",
      "organization": "Bill & Melinda Gates Foundation"
    },
    {
      "job_title": "Co-founder, Chairman",
      "organization": "Microsoft"
    }
  ],
  "education": [
    {
      "school_name": "Harvard University",
      "start": 1973,
      "end": 1975
    },
    {
      "school_name": "Lakeside School, Seattle",
      "start": null,
      "end": null
    }
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

Key/Value Stores

- A cada pieza de datos se le asigna un identificador
- Como valor se almacena cualquier conjunto de información
- El caso de uso estándar es:
  - **Cachés de acceso** a partes de programas
  - Por ejemplo, los iconos de los usuarios de una página, iconos de productos, etc.
- Key-Value: **Riak**, **Redis**, **Memcached**, **LevelDB**, **Amazon Dynamo**

Key/Value Stores (II)

```
define getCustomer(p_customerID):
  begin
    if exists(customerCache[p_customerID])
      return(customerCache[p_customerID])
    else
      return(addQueryResultsToCache(p_customerID,
             'SELECT * FROM customers
              WHERE customerID = p_CustomerID'))
    end
  end
```

(de *NoSQL for Mere Mortals*, Dan Sullivan, Addison-Wesley, 2015)

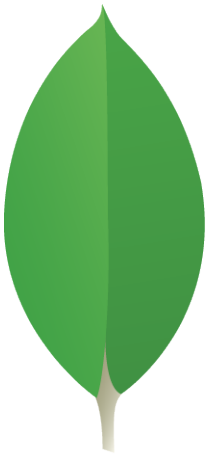
Bases de Datos Documentales

- A cada pieza de datos se le asigna un identificador
- La diferencia entre las key/value
  - En **Key/Value**, el valor es opaco (es un *blob*)
  - En las documentales, la base de datos puede ver el contenido del agregado, y utilizar su información como parte de las búsquedas y actualizaciones
- Documentos $\Rightarrow$ formatos jerárquicos tipo JSON o XML

Bases de Datos Documentales (II)

- La diferencia entre ambas un poco difusa
  - Por ejemplo, Riak es Key-Value pero permite realizar búsquedas indexadas parecidas a las de Solr/Lucene
  - Redis permite que los valores de datos sean estructurados en arrays, estructuras, mapas
- **CouchDB**, **MongoDB**, **OrientDB** (también soporta grafos)

Bases de Datos Documentales



mongoDB®

Conceptos de MongoDB

Base de datos En MongoDB, se pueden crear diferentes *bases de datos* en cada servidor. Las bases de datos son un conjunto de *colecciones*

Colección Una colección es un conjunto de *documentos*, cada uno de ellos identificado por una **clave**. Se puede ver como un **diccionario** que relaciona *ID* $\Rightarrow$ *documento*

Documento Conjunto de pares *clave*, *valor* que puede tener una estructura anidada. Se suelen representar como valores JSON, aunque internamente se representan eficientemente como BSON

Conceptos de MongoDB (II)

```
{
  "_id": "2",
  "type": "Movie",
  "title": "Truth",
  "year": 2015,
  "director_id": "345679",
  "genre": "Drama",
  "rating": {
    "score": 6.8,
    "voters": 12682
  },
  "criticisms":[
    {
      "journalist": "Jordi Costa",
      "media": {
        "name": "El Pais",
        "url": "http://elpais.com/"
      },
      "color": "red"
    },
    {
      "journalist": "Lou Lumenick",
      "media": "New York Post",
      "color": "green"
    }
  ]
}
```

Conceptos de MongoDB (III)

Consultas Posee un API para consultas sencillas con un lenguaje propio (no SQL)

Distribución Permite **distribuir** el contenido de las tablas entre diferentes ***Grupos de Réplicas***, que además permiten la **tolerancia a fallos**

Procesamiento/Consultas complejas El procesamiento y las consultas complejas se realizan usando el API de Map-Reduce o el de Agregación. Estos APIs permiten explotar la **distribución y paralelismo**

Conexión

Usando el lenguaje Python

```
import pymongo
from pymongo import MongoClient

client = MongoClient("localhost",
   27017)
```

```
>>> MongoClient(host=['localhost:27017'],
   document_class=dict, tz_aware=False, connect=
   True)
```

Creación de la base de datos "centic18"

```
db = client.centic18
```

Creación de la colección "productos"

```
productos = db.productos
```

```
>>> Collection(Database(MongoClient(host=['
    localhost:27017'], document_class=dict,
    tz_aware=False, connect=True), 'centic18'), '
    productos')
```

Métodos de inserción y actualización

MongoDB ofrece métodos para inserción y actualización:

- `insert_one()`, `insert_many()` (batch)
- `update_one()` – Permite actualizar un objeto con nuevos campos
- `update_many()` – Permite poner nuevos valores calculados a un conjunto de objetos

Inserción de productos

```
productos.insert_one(
{
    'nombre': 'Samsung XX40',
    'descripción' : 'Televisión
      Samsung, 40"...',
    'precio' : '600',
    'fabricante': 'Samsung',
    'stock': 20,
    'tags': ['tv', '40"', 'lcd', '
      hdmi', 'smart-tv']
})
```

Inserción de productos (II)

```
productos.insert_one(
    {'nombre': 'LG T42',
     'descripción' : 'Televisión
       LG 42"',
     'precio' : 655,
     'fabricante': "LG",
     'stock': 10,
     'tags' : ['tv', '42"', 'led',
        'hdmi', 'smart-tv']
    })
```

Inserción de productos (III)

- El ID se le asigna automáticamente
- O bien se puede asignar a mano añadiendo un campo especial "`_id`"
- Los `tags`, si bien son simulables con tecnologías relacionales, aquí ofrecen mucha flexibilidad
- Como se verá, la búsqueda se puede hacer por *tag*

Métodos de búsqueda sencilla

- El método de búsqueda principal es `find()`, que tiene muchas opciones
- Existe también una variante `find_one()`, que busca sólo un elemento (si existe)
- En general permite especificar:
  - El filtro de búsqueda
  - Ordenación de resultados por algún campo
  - Proyección de los campos mostrados en el resultado
  - Número de resultados máximo (*limit*)
  - Número de elementos iniciales a ignorar (*skip*)
  - El tamaño del *batch*

Métodos de búsqueda sencilla (II)

- Proyección:

```
db.productos.find({<criterio>},
        {"fabricante" : 1, "stock" : 1})
// =>
[{'_id': ObjectId('5b1d09b3b3d977001bc53e03'
    ),
  'fabricante': 'Samsung',
  'stock': 20},
 {'_id': ObjectId('5b1d09beb3d977001bc53e04'
    ),
  'fabricante': 'LG',
  'stock': 10}]
```

Métodos de búsqueda sencilla (III)

- Se pueden utilizar condicionantes para la búsqueda:

```
db.productos.find({"stock" :
  {"$gt" : 10, "$lte" : 30}})

db.productos.find({ "$or" :
     [ {"fabricante" : "Samsung" },
       {"fabricante" : "LG"} ] })
```

- Búsqueda con *tags* seleccionadas:

```
db.productos.find({"tags" :
    {"$elemMatch" :
        {'$in' : ['tv', 'smart-tv']}}})
```

Ups, ¡valoraciones de usuarios!

- En un momento dado podemos querer añadir las valoraciones de usuarios
- En un entorno relacional hubiera obligado a **crear tablas**, **cambiar cosultas**, **realizar JOINs**, etc.
- En el caso de MongoDB:
  - El ID del objeto se ha obtenido al mostrarlo
  - El **`user`** es del *login* del usuario

```
db.productos.update_one(
 {'_id': ObjectId('5b1d09b3b3d977001bc53e03'
    )},
 {'$push' : {'valoraciones' :
   {'user': 'Pepe', 'estrellas' : 3.5, '
      comment': 'bah'}}})
```

Ups, ¡valoraciones de usuarios! (II)

- No hace falta definir de antemano el campo "`valoraciones`"
- Las modificaciones sobre el objeto producto son **atómicas**
  - No hay necesidad de **transacciones ACID**
- Todos los campos se recuperan **en la consulta**

Map-Reduce

Map-Reduce es el principal mecanismo de búsqueda y transformación en BBDD NoSQL. Tiene su origen en **lenguajes funcionales**:

`map()`

Ejecuta una misma función sobre todos los elementos de un conjunto

`reduce()`

Procesa un conjunto de valores para producir un valor de salida

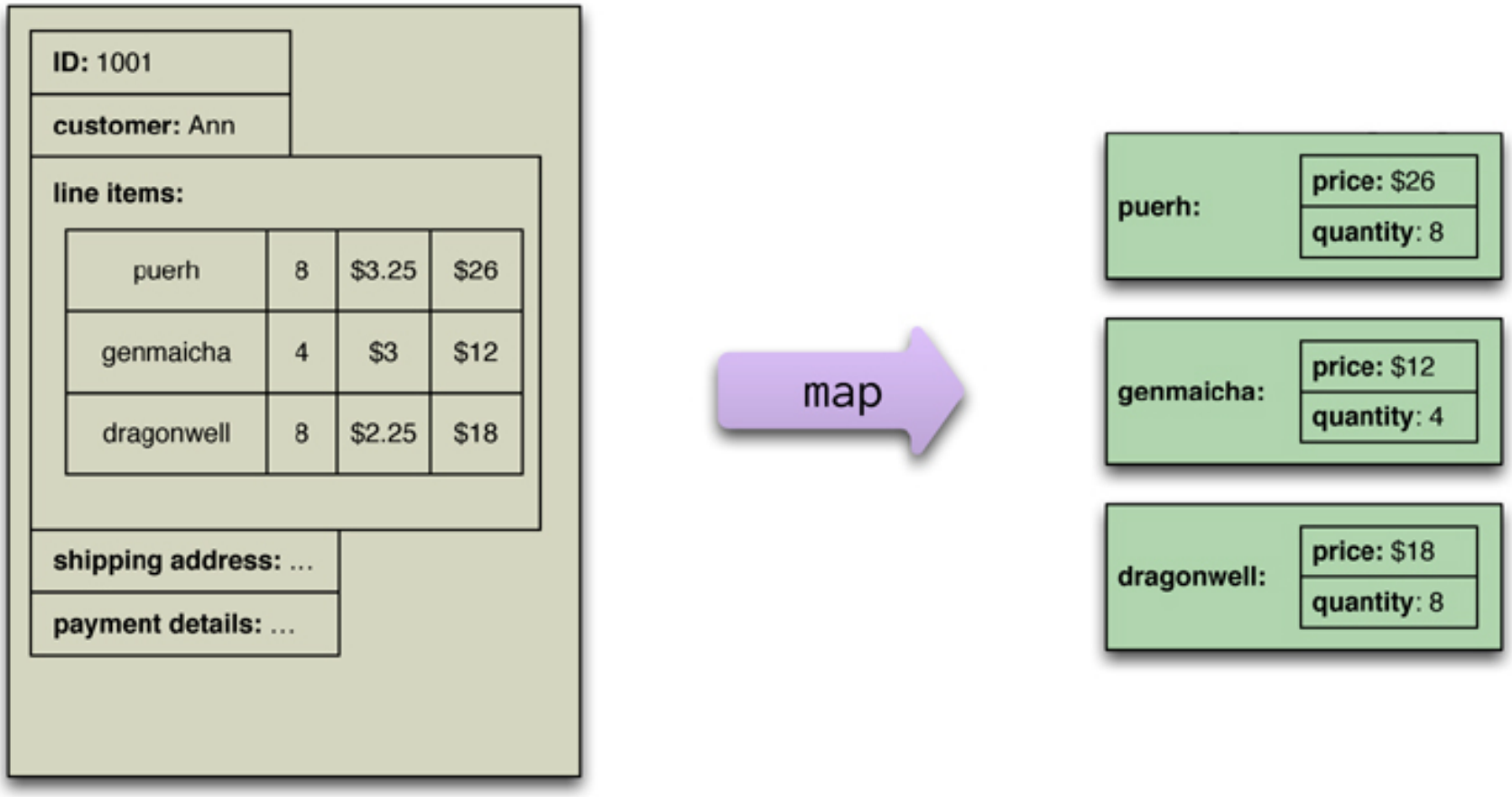
Map-Reduce (II)

Map-Reduce combina ambas operaciones:

- Una misma operación `map()` a cada dato residente en un nodo se realiza de forma **paralela en todos los nodos**
- Con los resultados parciales de cada nodo, una función `reduce()` genera un resultado (o conjunto de resultados) final
- Hay un proceso intermedio de *shuffle* para agrupar valores relacionados antes del `reduce()`
- Resultados parciales en el mismo nodo (localidad) $\Rightarrow$ procesamientos **en cadena**

Map-Reduce

NoSQL Distilled. Sadalage, Fowler, Addison-Wesley, 2012



Map-Reduce (II)

NoSQL Distilled. Sadalage, Fowler, Addison-Wesley, 2012

Map-Reduce como generalización de consultas

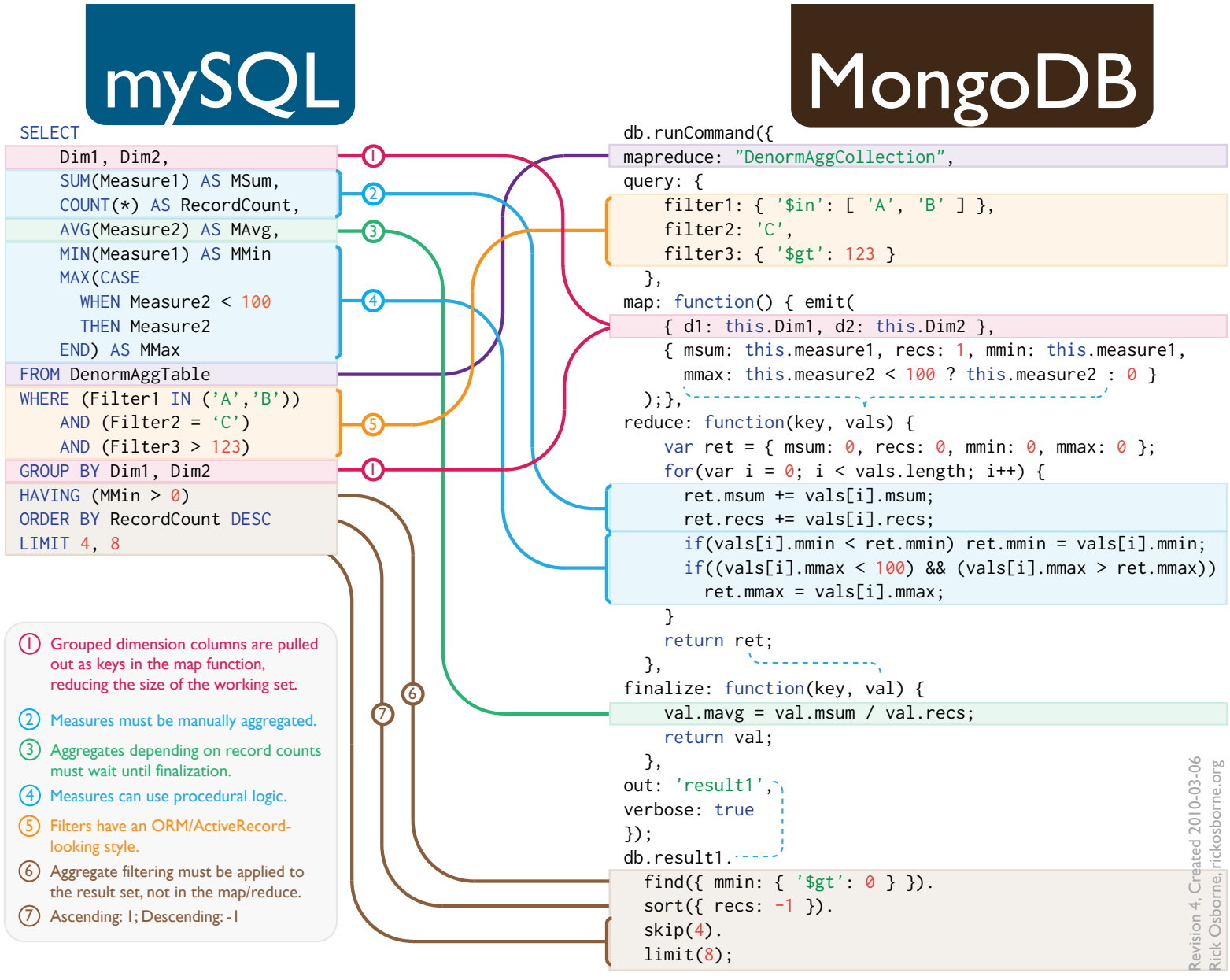
Ejemplo: Imagínese un biólogo marino que hace anotaciones de cada animal que ve en el océano, y quiere saber cuántos tiburones ha visto por mes:

```
SELECT MONTH(observation_timestamp) AS
    observation_month,
        sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

Map-Reduce como generalización de consultas (II)

MongoDB con el API de MapReduce:

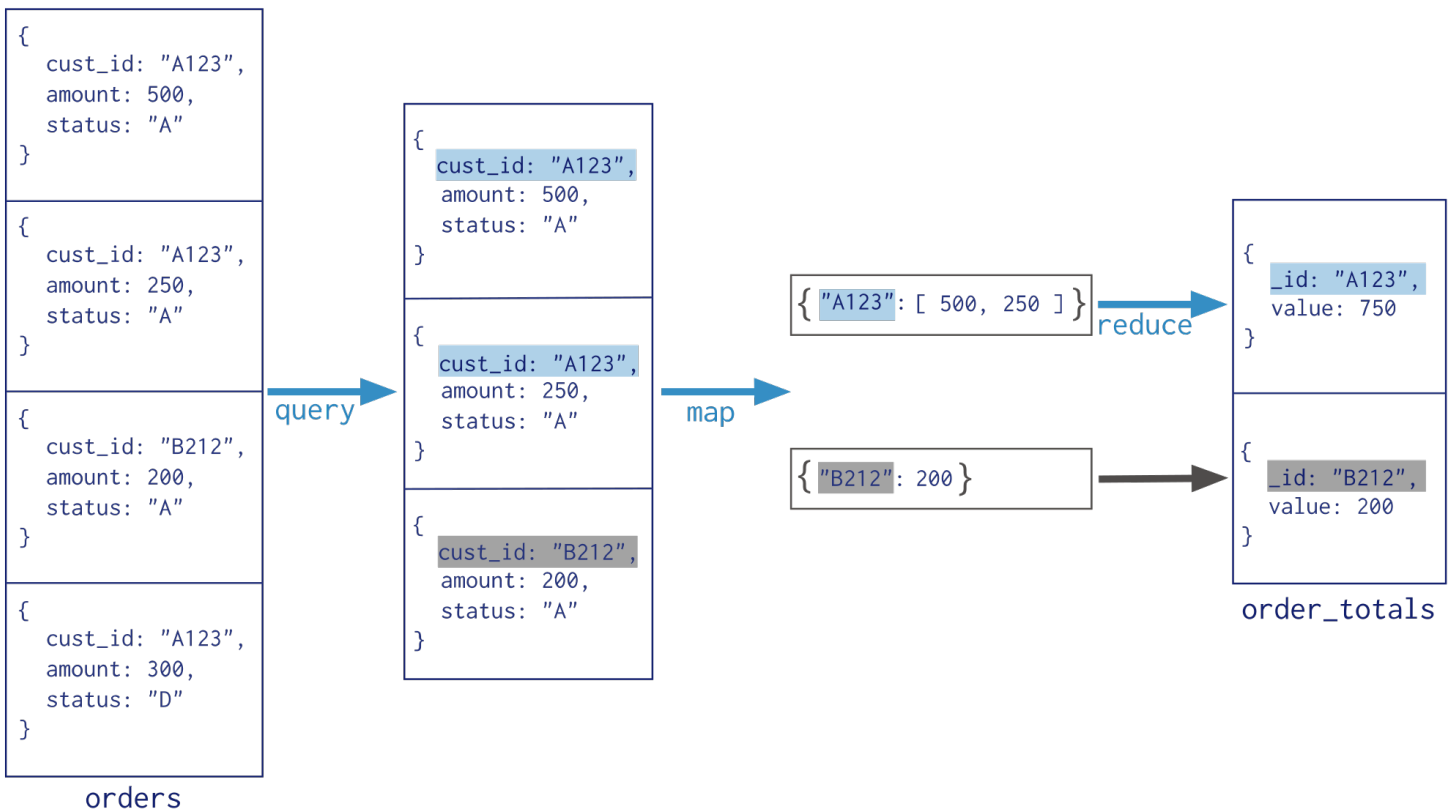
```
db.observations.mapReduce(
  function map() {
    var year = this.observationTimestamp.
        getFullYear();
    var month = this.observationTimestamp.
        getMonth() + 1;
    emit(year + "-" + month, this.numAnimals);
  },
  function reduce(key, values) {
    return Array.sum(values);
  },
  {
    query: { family: "Sharks" },
    out: "monthlySharkReport"
  });
```


mySQL
MongoDB
SELECT
Dim1, Dim2,
SUM(Measure1) AS MSum,
COUNT(*) AS RecordCount,
AVG(Measure2) AS MAvg,
MIN(Measure1) AS MMin
MAX(CASE
WHEN Measure2 < 100
THEN Measure2
END) AS MMax
FROM DenormAggTable
WHERE (Filter1 IN ('A','B'))
AND (Filter2 = 'C')
AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8
db.runCommand({
mapreduce: "DenormAggCollection",
query: {
filter1: { '$in': ['A', 'B'] },
filter2: 'C',
filter3: { '$gt': 123 }
},
map: function() { emit(
{ d1: this.Dim1, d2: this.Dim2 },
{ msum: this.measure1, recs: 1, mmin: this.measure1,
mmax: this.measure2 < 100 ? this.measure2 : 0 }
);},
reduce: function(key, vals) {
var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 };
for(var i = 0; i < vals.length; i++) {
ret.msum += vals[i].msum;
ret.recs += vals[i].recs;
if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax))
ret.mmax = vals[i].mmax;
}
return ret;
},
finalize: function(key, val) {
val.mavg = val.msum / val.recs;
return val;
},
out: 'result1',
verbose: true
});
db.result1.
find({ mmin: { '$gt': 0 } }).
sort({ recs: -1 }).
skip(4).
limit(8);
1 Grouped dimension columns are pulled out as keys in the map function, reducing the size of the working set.
2 Measures must be manually aggregated.
3 Aggregates depending on record counts must wait until finalization.
4 Measures can use procedural logic.
5 Filters have an ORM/ActiveRecord-looking style.
6 Aggregate filtering must be applied to the result set, not in the map/reduce.
7 Ascending: 1; Descending: -1
Revision 4, Created 2010-03-06
Rick Osborne, rickosborne.org

Map-Reduce

Collection

```
db.orders.mapReduce(
        map      →   function() { emit( this.cust_id, this.amount ); },
        reduce   →   function(key, values) { return Array.sum( values ) },
                     {
        query    →     query: { status: "A" },
        output   →     out: "order_totals"
                     }
                   )
```



Map-Reduce

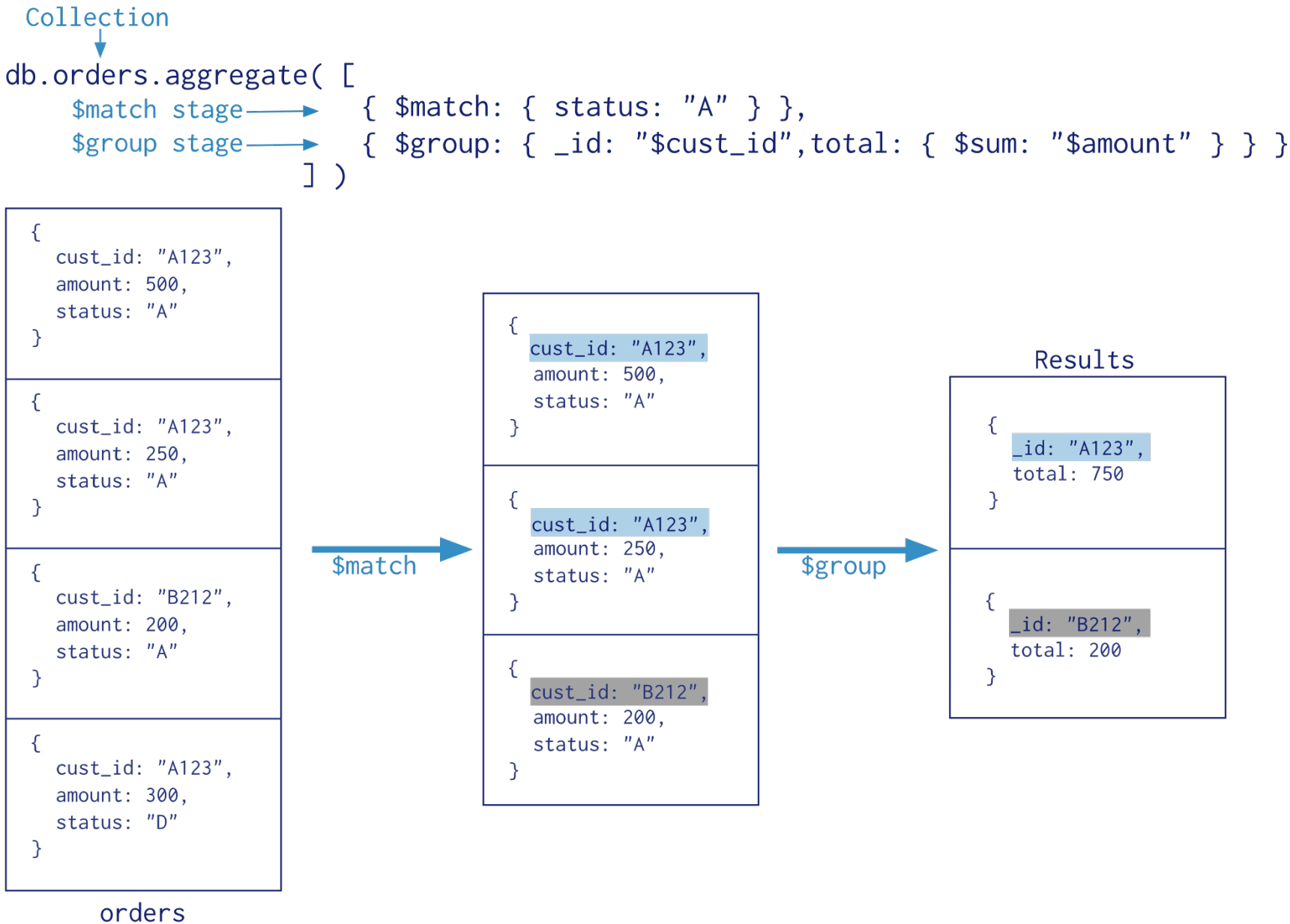
```
from bson.code import Code
map = Code(
'''function () {
     obj = this
     if ('valoraciones' in this)
         this.valoraciones.forEach(function (v) {
             emit(obj.fabricante, v.estrellas);
         })
   }''')
reduce = Code(
'''function (key, values) {
         return Array.sum(values) / values.length;
   }''')
results = db.productos.map_reduce(map, reduce, "
   valoración_media")
```

Map-Reduce (II)

La colección **`valoración_media`** tendrá valores:

```
[{'_id': 'Samsung', 'value': 4.25},
 {'_id': 'LG', 'value': 4},
 ...
]
```

Framework de agregación



BD Documentales: Casos de uso

- *Drop-in Replacement*
  - Todo lo que se puede hacer con SQL se puede hacer con Documentales
  - La eficiencia (tiempo y espacio es similar)
  - Ofrecen características de escalabilidad horizontal
- Trabajo con datos...
  - Cambiantes, externos, JSON, procedentes de APIs REST
- Entorno **ágil** (no hace falta esquema)

BD Documentales: Casos de uso (II)

- Flexibilidad en el modelo de datos
  - Utilización de agregados
  - Los ORMs también permiten usarlos
  - Pero el código que generan a veces **no es óptimo**
  - Y **crea deuda técnica con la implementación** (¿mantenimiento, evolución?, etc.)
- Trabajo con **datos geográficos** (GeoJSON, consultas por proximidad geográfica)
- **Eficiencia** para ciertos tipos de cargas de trabajo (muchas actualizaciones)

Bases de Datos Columnares

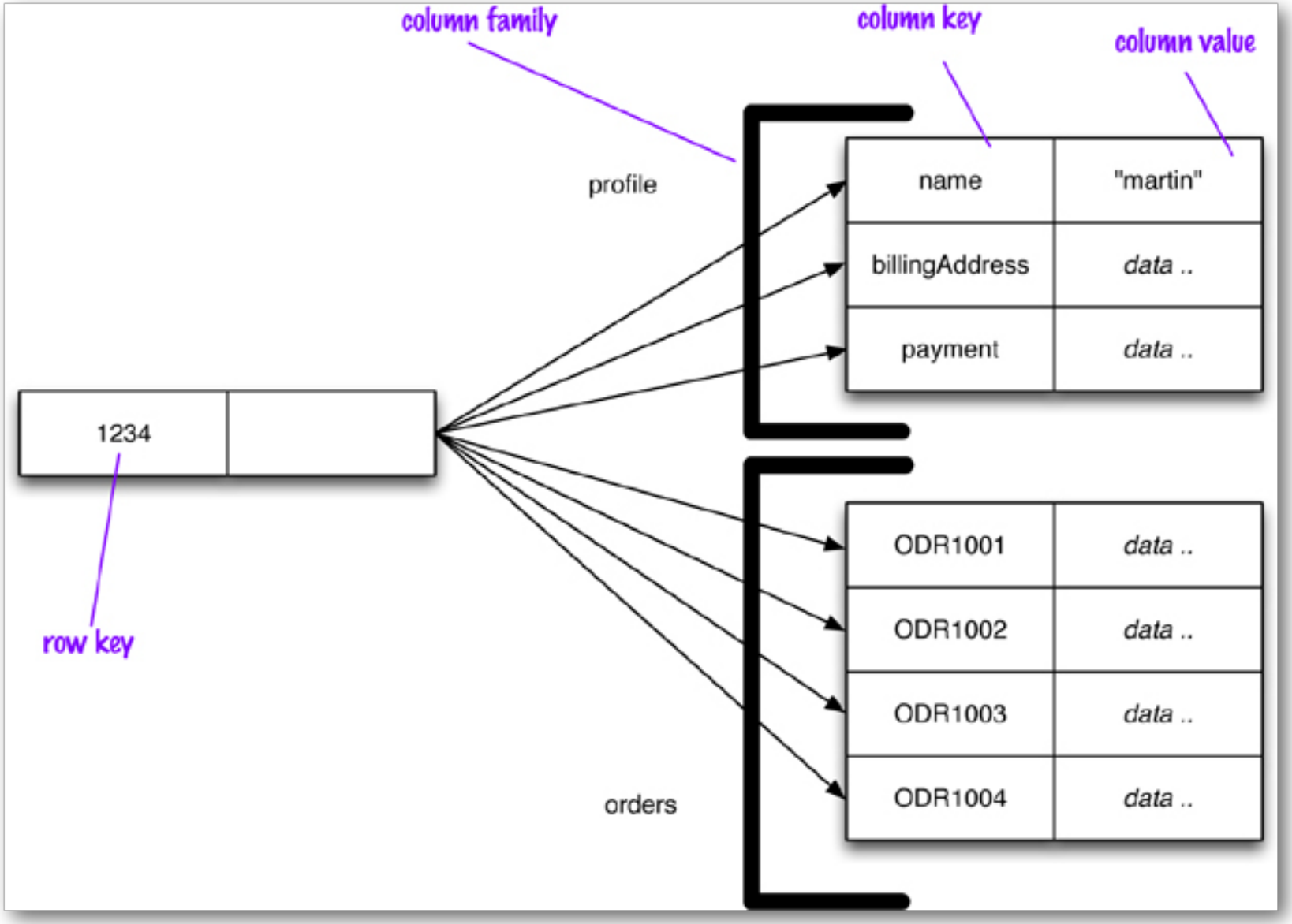
- Influenciadas por el artículo de Google de 2006 sobre BigTable[2]
- En general, parecidos a las tablas SQL, salvo que cada fila puede:
  - Tener un **conjunto de columnas diferente**
  - Almacenar *series temporales* dentro de una misma fila (varias *versiones* de un mismo conjunto de columnas)
- Cada fila tiene un identificador y es un agregado de familias de columnas (*column family*)

Bases de Datos Columnares (II)

- Cambian el modo de almacenamiento para favorecer ciertas aplicaciones (almacenamiento **por columnas en vez de por filas**)
- Bases de datos: **HBase**, **Cassandra**, **Vertica**, **H-Store**

---

[2]Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C; Wallach, Deborah A; Burrows, Michael 'Mike'; Chandra, Tushar; Fikes, Andrew; Gruber, Robert E (2006), *Bigtable: A Distributed Storage System for Structured Data*, (PDF), Google.


column family
column key
column value
profile
name
"martin"
billingAddress
data ..
payment
data ..
1234
row key
ODR1001
data ..
ODR1002
data ..
ODR1003
data ..
ODR1004
data ..
orders

Introducción a HBase

Base de datos *wide column store*

- Importante: Permite variabilidad en el número de columnas de cada fila
- Crece en número de filas. También puede crecer en **número de columnas** sin coste
- Una dimensión más de diseño
- También: Permite almacenar todos los valores antiguos para un dato (**series temporales**)

Introducción a HBase (II)

Diseñada para guardar cantidades de datos ingentes en clústers de ordenadores. "**La base de datos de Hadoop**"

- HDFS para almacenar la base de datos de forma distribuida
- HDFS permite acceso secuencial eficiente y trabajos *batch*
- HBase implementa acceso *random access* muy rápido
- Map/Reduce para realizar búsquedas complejas y procesamiento sobre la BD

Introducción a HBase (III)

Tabla *hash* enorme, tolerancia a fallos y distribución automática y guiada

- La BD mantiene **físicamente juntas** las claves ordenadas **lexicográficamente**
- Es **muy rápido** encontrar filas consecutivas
- Además, cada columna se almacena **independientemente**
- El diseño correcto de la clave es crítico

Introducción a HBase (IV)

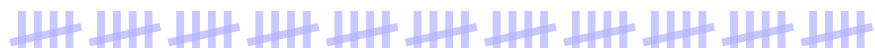
Operaciones muy rápidas:

- Operaciones CRUD (*Create*, *Read*, *Update*, *Delete*) rápidas sobre documentos identificados por una clave (se implementa alguna versión de árbol B+)
- Búsqueda secuencial con filtrado, ya que las claves están ordenadas
- Almacenamiento de un número ilimitado de **versiones de los datos**

¿Cuándo usar HBase?

Necesidad de realizar cientos, miles de operaciones por segundo en TB/PB de datos...

...con patrones de acceso bien conocidos de antemano y sin gran complejidad

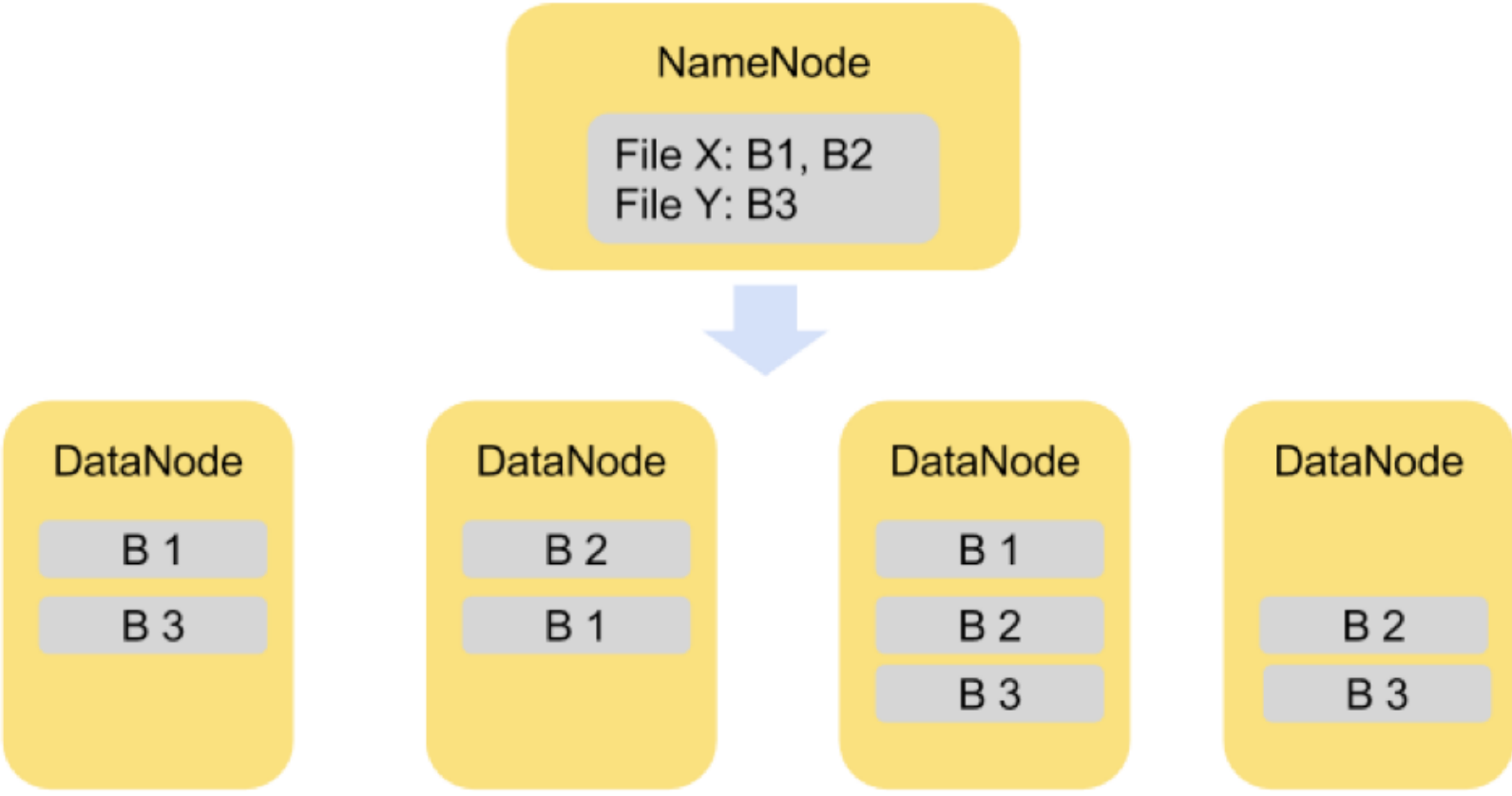


Tecnologías habilitadoras de HBase

HBase sobre Hadoop (HDFS y Map-Reduce)

- Una **tabla** está formada por un número de filas, identificadas por una clave
- Cada tabla se divide en **regiones** (por rangos de clave ordenada lexicográficamente) (**partición horizontal**)
- Una instalación de HBase utiliza un conjunto de *Region Servers*: nodos de computación con almacenamiento local (y conectados al clúster HDFS)

HDFS


NameNode
File X: B1, B2
File Y: B3
DataNode
B 1
B 3
DataNode
B 2
B 1
DataNode
B 1
B 2
B 3
DataNode
B 2
B 3

Hbase Shell

Comandos básicos CRUD

- Creación de tablas, especificando las *column families* iniciales:

```
hbase> create 'productos', 'd', 'val', 'tags
    '

0 row(s) in 1.3810 seconds
=> Hbase::Table - productos
```

- Se crea la tabla **`productos`** con tres familias de columnas, **`d`**, que guardará los datos del producto, **`val`**, que guardará las valoraciones de los usuarios, y **`tags`**, que guardará los tags de los productos
- Se puede insertar y consultar datos

Creación de la base de datos

```
try:
    connection.create_table('productos',
    {
      'd': dict(max_versions=1),
      'val' : dict(max_versions=1,compression='GZ
          '),
      'tags' : dict(max_versions=1)
    })
except:
    print("Error creando database: productos")
    pass
```

Añadimos los datos

Añadiremos ahora los productos:

```
productos.put(b'tv40-sxx40',
 {
   'd:nombre': 'Samsung XX40',
   'd:descripción' : 'Televisión Samsung, 40"...'
      ,
   'd:precio' : '600',
   'd:fabricante': 'Samsung',
   'd:stock': '20',
   'd:tags' : 'tv,40",lcd,hdmi,smart-tv'
 })
```

Añadimos los datos (II)

Obtención de filas:

```
productos.row(b'tv40-sxx40')
# ==>
{b'd:descripci\xc3\xb3n': b'Televisi\xc3\xb3n
    Samsung, 40"...',
 b'd:fabricante': b'Samsung',
 b'd:nombre': b'Samsung XX40',
 b'd:precio': b'600',
 b'd:stock': b'20',
 b'd:tags': b'tv,40",lcd,hdmi,smart-tv'}
```

Añadimos los datos (III)

- Varias consideraciones del código anterior:
  - Al ser una tabla ordenada por clave, la elección de la clave es **crucial**
  - En nuestro caso incluye la categoría inicial del producto
    - Obtener filas consecutivas es **muy rápido**
    - Por ejemplo, todas las televisiones se pueden obtener buscando '`tv`'
- Además de poder almacenar millones de filas, las filas pueden **crecer tanto como se quiera también en columnas**

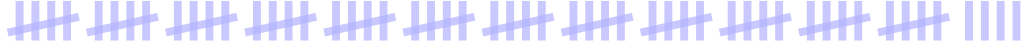
Añadimos los datos (IV)

- Esta estrategia se puede usar para almacenar relaciones **maestro/detalle** de manera eficiente, ya que se obtiene en una sola consulta
- Todos los datos relacionados se obtienen **con un solo get**
- La usaremos para los *tags*

Añadimos los datos (V)

```
productos.put(b'tv40-sxx40', {
 'tags:tv' : '',
 'tags:40"' : '',
 'tags:lcd' : '',
 'tags:hdmi' : '',
 'tags:smart-tv' : ''})
```

- Las columnas no tienen valor
- Sólo se usan como marcador de búsqueda rápida



Añadimos los datos (VI)

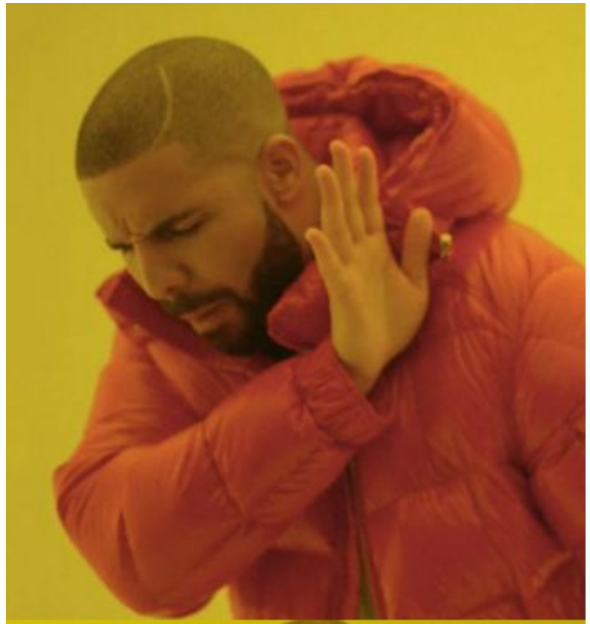
La búsqueda se realiza de forma muy rápida:

```
productos.scan('tv40',columns=['tags:hdmi'])
# ==>
# [(b'tv40-sxx40', {b'tags:hdmi': b''}),
#  (b'tv42-LGT42', {b'tags:hdmi': b''})]
```

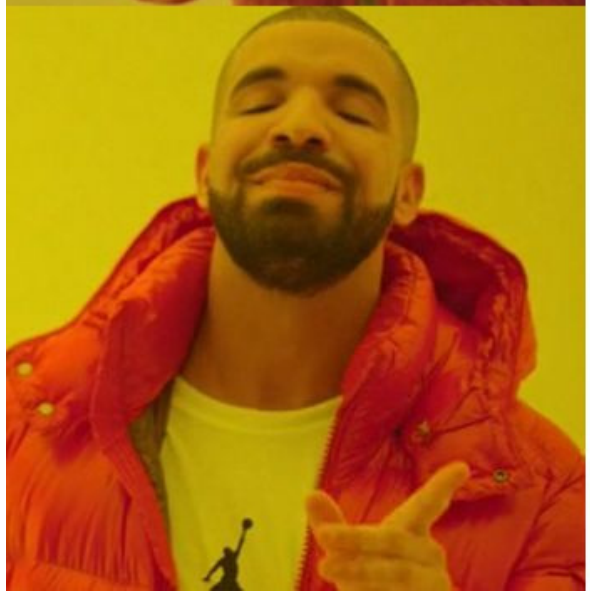
En cualquier caso, se debe huir de usar **`scan`**

$\Rightarrow$ HBase está construido para accesos O(1)

`scan()` *vs.* `get()`



`scan()`



`get()`

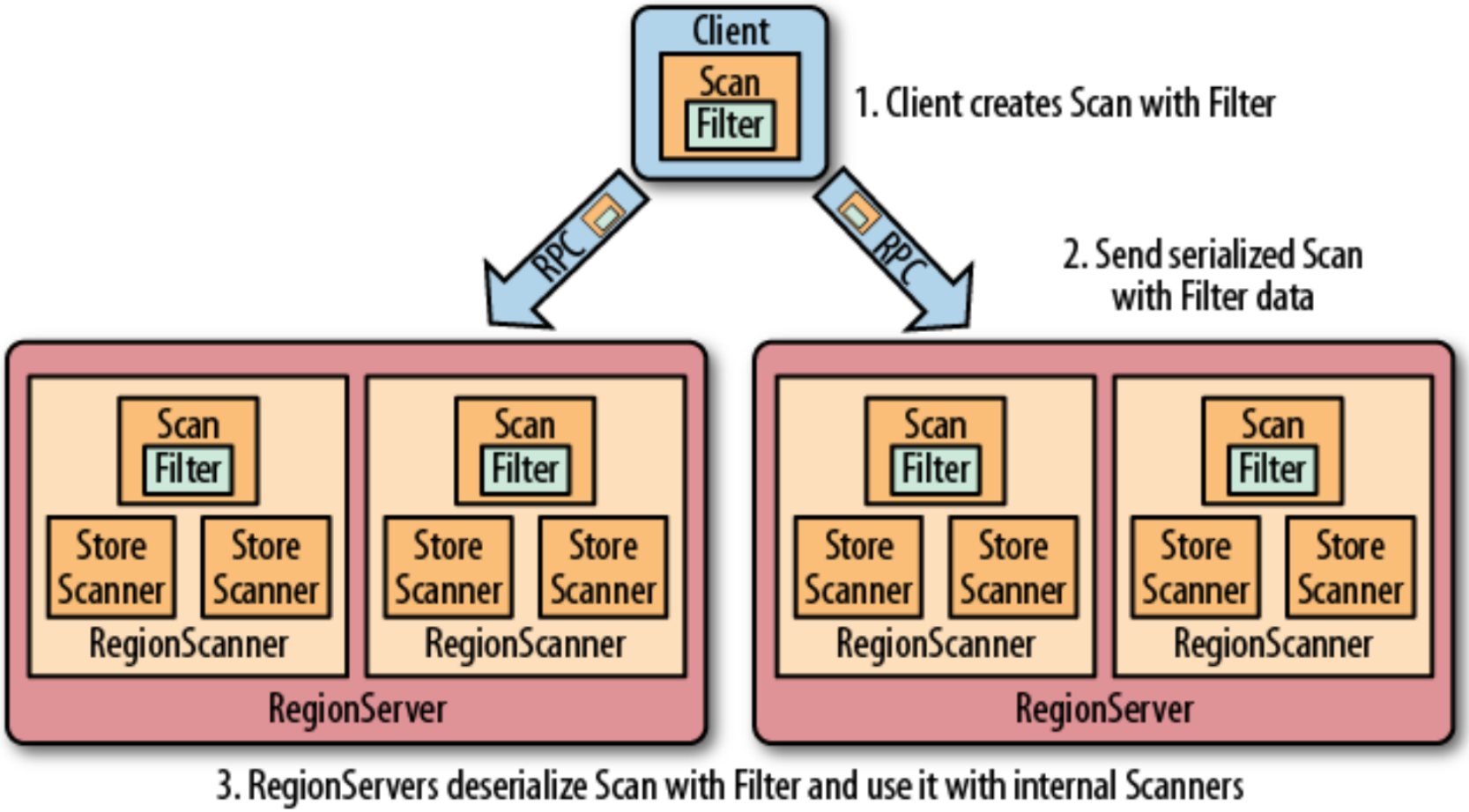
Búsquedas y filtrado

- Como se ha visto, el diseño de las tablas HBase tiene que ir orientado a la **optimización de las lecturas**
- **Desnormalizar todo lo necesario**
- **Con un único acceso** se obtenga toda la información necesaria
- Sin embargo, esto no es posible siempre:
  - Por ejemplo, se tiene que procesar un conjunto de elementos no predeterminado
  - Se quieren calcular resultados agregados semanales, diarios, etc.

Búsquedas y filtrado (II)

- HBase $\Rightarrow$ lenguaje de filtrado $\Rightarrow$ el servidor filtre los resultados
- Clientes Thrift remotos y desde el *shell*
- Filtrado de cada región en paralelo
- Escalabilidad horizontal con varios `RegionServer`s
- Aunque ofrece diversos mecanismos de filtrado, **no todos son igual de eficientes**
- De hecho, el principal problema es que hay mucha diferencia (hasta el punto de hacer algunos impracticables) si no se usan bien

Búsquedas y filtrado



Columnares: Casos de uso

- Almacenamiento masivo y Acceso aleatorio basado en clave
  - Almacenamiento sencillo
  - Distribución en conjunto de servidores escalable
  - Tolerante a fallos
- Procesamiento analítico
  - El almacenamiento es por columnas, sólo se recuperan los datos que se usan
  - (En SQL los bloques de datos contienen toda la fila)

Columnares: Casos de uso (II)

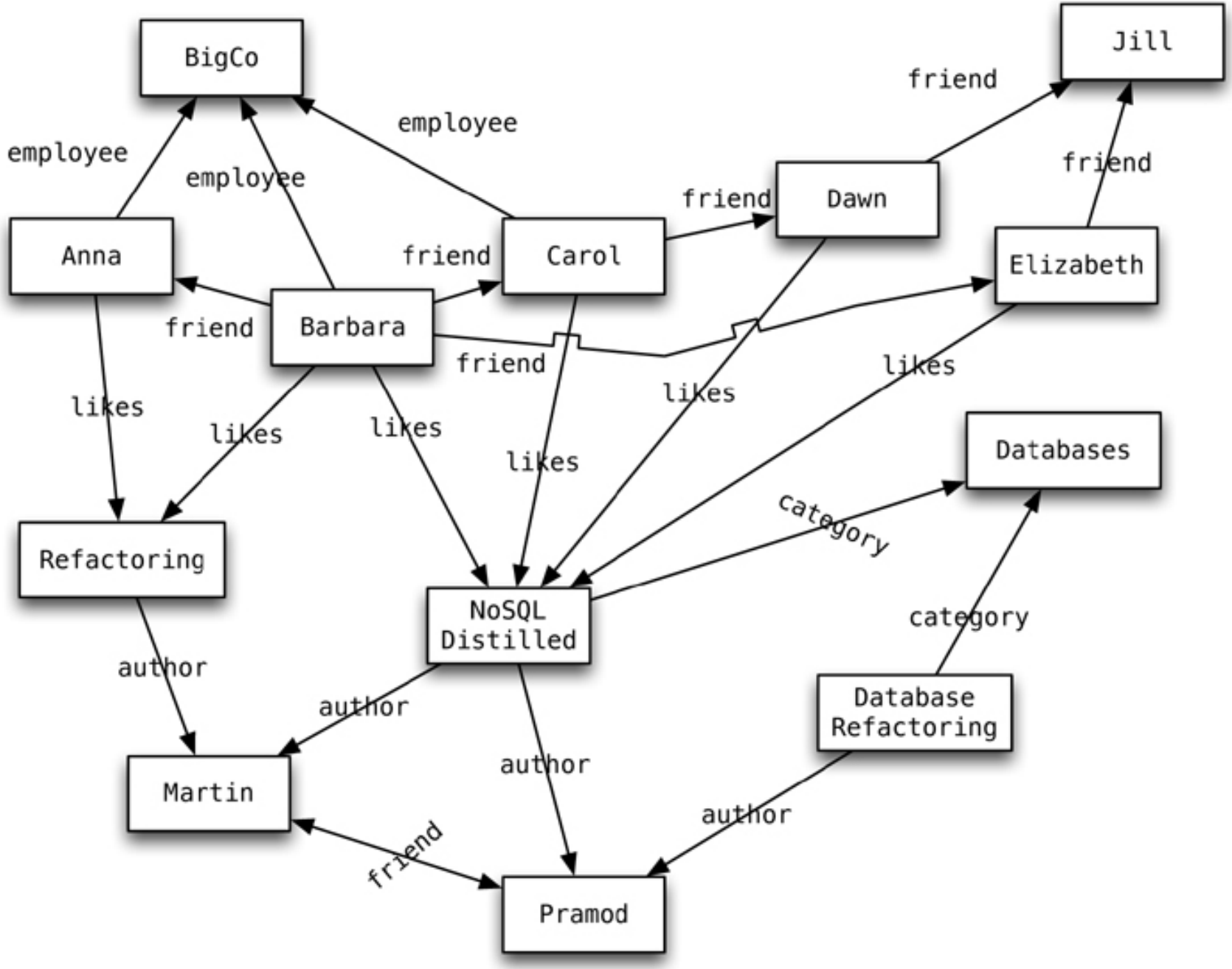
- Procesamiento de series temporales/Datos de IoT
  - Cada celda puede guardar un histórico de datos
  - Existen adaptaciones de HBase (como Accumulo o TSDB) que permiten almacenar y procesar eficientemente series temporales

Bases de Datos de Grafos

- Las bases de datos de grafos llevan el mecanismo *muchos a muchos* al extremo
- Datos en los que existen muchas relaciones entre sí y **las relaciones** tienen un significado primordial
- Las bases de datos de grafos se basan en la construcción y consulta de un grafo que consta de
  - **Vértices** también llamados *nodos* o *entidades*, y
  - **Aristas** (***Edges***), también llamados *relaciones*

Bases de Datos de Grafos (II)

- Los grafos pueden capturar relaciones complejas entre entidades y ofrecen lenguajes de búsqueda, actualización y creación que permiten trabajar con subconjuntos del grafo
- Origen en las bases de datos de hechos (*Datalog*)
- Ejemplos: **FlockDB**, **Neo4J**, **OrientDB**


BigCo
Jill
friend
employee
employee
employee
friend
friend
Dawn
Anna
friend
Carol
Elizabeth
friend
Barbara
friend
likes
likes
likes
likes
likes
likes
Databases
Refactoring
category
NoSQL
Distilled
category
author
author
Database
Refactoring
Martin
author
author
friend
Pramod

Ejemplo de datos y consulta en Neo4J

```
CREATE
  (NAmerica:Location {name:'North America', type:
     'continent'}),
  (USA:Location {name:'United States', type:'
     country' }),
  (Idaho:Location {name:'Idaho', type:'state' }),

  (Lucy:Person {name:'Lucy' }),

  (Idaho)-[:WITHIN]->(USA)-[:WITHIN]->(NAmerica),
  (Lucy) -[:BORN_IN]-> (Idaho)
```

Ejemplo de datos y consulta en Neo4J (II)

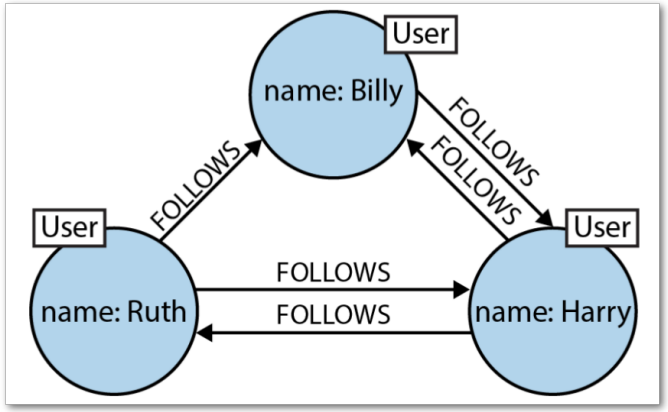
Y de consulta:

```
MATCH
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us:
    Location {name:'United States'}),
(person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:
    Location {name:'Europe'})
RETURN person.name
```

(con esta consulta tan cercana al lenguaje natural, estamos buscando los emigrantes de EEUU en Europa)

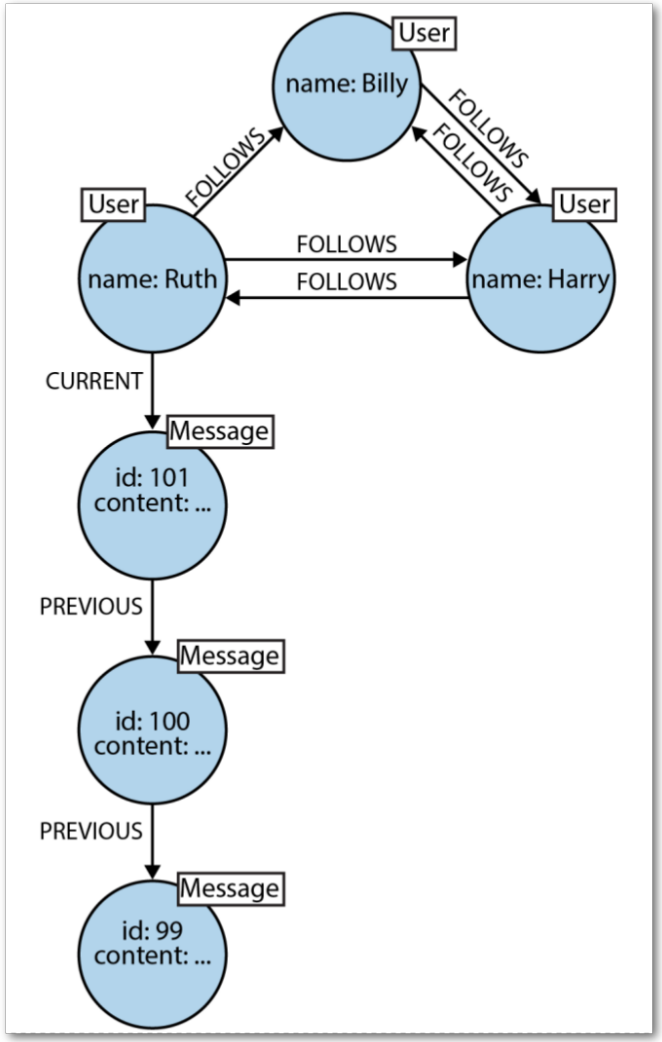
Aplicabilidad de Grafos

- Los grafos, conceptualmente, aparecen en casi cualquier dominio
- Además, su flexibilidad hace que se puedan aplicar de diferentes formas
- Por ejemplo, una relación de *follow* entre usuarios:



Aplicabilidad de Grafos (II)

- (nótese cómo **Billy no ha seguido a Ruth**: las relaciones pueden ser unidireccionales o bidireccionales)
- Incluso se puede usar para guardar el conjunto de mensajes que se intercambian:



Flexibilidad y eficiencia

- Las bases de datos basadas en grafos vienen a suplir dos carencias fundamentales:
  1. La carencia expresiva del resto de paradigmas para expresar ciertos algoritmos que se expresan de forma natural en forma de grafos
  2. La eficiencia del tratamiento de estos procesos en grafos vs. otros paradigmas

Introducción a Neo4j

- Ofrece una base de datos de grafos con posibilidad de extenderse a varios ordenadores (aunque sólo uno de los ordenadores soporta escritura, replicación *master-slave*)
- Es compatible con el estándar Apache TinkerPop para la creación de grafos
- Ofrece un lenguaje de creación y consulta de grafos: **Cypher**

Introducción a Neo4j (II)

- Ofrece también un *browser* web para lanzar consultas
- También una consola que interpreta el lenguaje Cypher
- Se puede usar desde Jupyter Notebook con la extensión `ipython-cypher`
- Finalmente, ofrece todos sus servicios a través de un API REST

Grafos en Neo4j

- Los grafos en Neo4j son grafos etiquetados y con propiedades
- Están compuestos por **nodos**, **relaciones**, **propiedades** y **etiquetas**
- Los nodos contienen propiedades, en la forma de pares clave-valor. Las claves son cadenas de caracteres y los valores pueden ser tipos primitivos o *arrays*
- A los nodos se les puede etiquetar con una o más ***etiquetas***. Las etiquetas agrupan nodos por rol dentro del grafo

Grafos en Neo4j (II)

- Las relaciones conectan nodos y estructuran el grafo. Una relación siempre tiene:
  - una dirección,
  - un nombre propio,
  - un nodo de inicio y otro de fin
  - un conjunto de propiedades
- Las propiedades permiten añadir información adicional al hacer el recorrido, como el peso asociado a atravesar ese enlace o la calidad del mismo

El lenguaje Cypher

- Lenguaje de especificación de búsquedas y modificaciones en el grafo
- En las búsquedas y creaciones de nodos se especifican nodos con la sintaxis:

```
(nombre:Etiqueta {propiedad: valor, ... })
```

- Las relaciones se especifican entre nodos de la siguiente manera, usando *ASCII art*:

```
(nodo_origen)-[:RELACIÓN]->(nodo_destino)
```

El lenguaje Cypher (ii)

- Creación de nodos y enlaces: **CREATE**

```
CREATE (nodo1:Etiqueta1 { propX: valorX, ... } ),
       (nodo2:Etiqueta2 { propY: valorY, ... } ),
       ...
       (nodo1)-[:RELACIONADO_CON]->(nodo2)
```

En nuestro caso de productos

```
CREATE (n:Producto
          {nombre: 'LG T42',
           descripción : 'Televisión LG 42"',
           precio : 655,
           fabricante: "LG",
           stock: 10,
           tags : 'tv,42",led,hdmi,smart-tv' })
```

Consultas

- Las consultas se realizan con el operador `MATCH`
- Es un operador de consulta a través de ejemplos (*query by example*)
- Especifica nodos y propiedades como un ejemplo de los nodos y relaciones que se buscan
- La sintaxis:

```
MATCH especificación, especificación, ...
[WHERE especificación]
RETURN [DISTINCT] nodos
```

Consultas (II)

- **MATCH** también se utiliza para seleccionar nodos para borrar (**DELETE** o **DETACH DELETE**)
- Las consultas nombran nodos sobre los que se pueden buscar relaciones
- Después, con **RETURN** se especifica lo que devolver

```
MATCH (n:Producto) RETURN n
```

(retorna todos los productos)

```
MATCH (n { nombre: 'Diego' }) RETURN n.edad;
```

Consultas (III)

y también se puede especificar relaciones que se tienen que cumplir entre los nodos para ser devueltos. Por ejemplo, todos los abuelos con sus nietos:

```
MATCH (nieto), (abuelo),
   (nieto)-[:HIJO_DE]->()-[:HIJO_DE]->(
       abuelo)
 RETURN abuelo, nieto
```

o incluso:

```
MATCH (nieto)-[:HIJO_DE]->()-[:HIJO_DE]->(
    abuelo)
  RETURN abuelo, nieto
```

(nótense los nodos anónimos)

Creación de nodos a partir de otros

- Un patrón común es la creación de nodos a partir de otros:
- P. ej. apuntar los hijos de alguien:

```
MATCH (padre:Usuario { nombre: 'Diego' })
CREATE (violeta:Usuario {nombre:'Violeta'}),
       (martina:Usuario {nombre:'Martina'}),
       (violeta)-[:HIJO_DE]->(padre),
       (martina)-[:HIJO_DE]->(padre)
```

Creación de relaciones

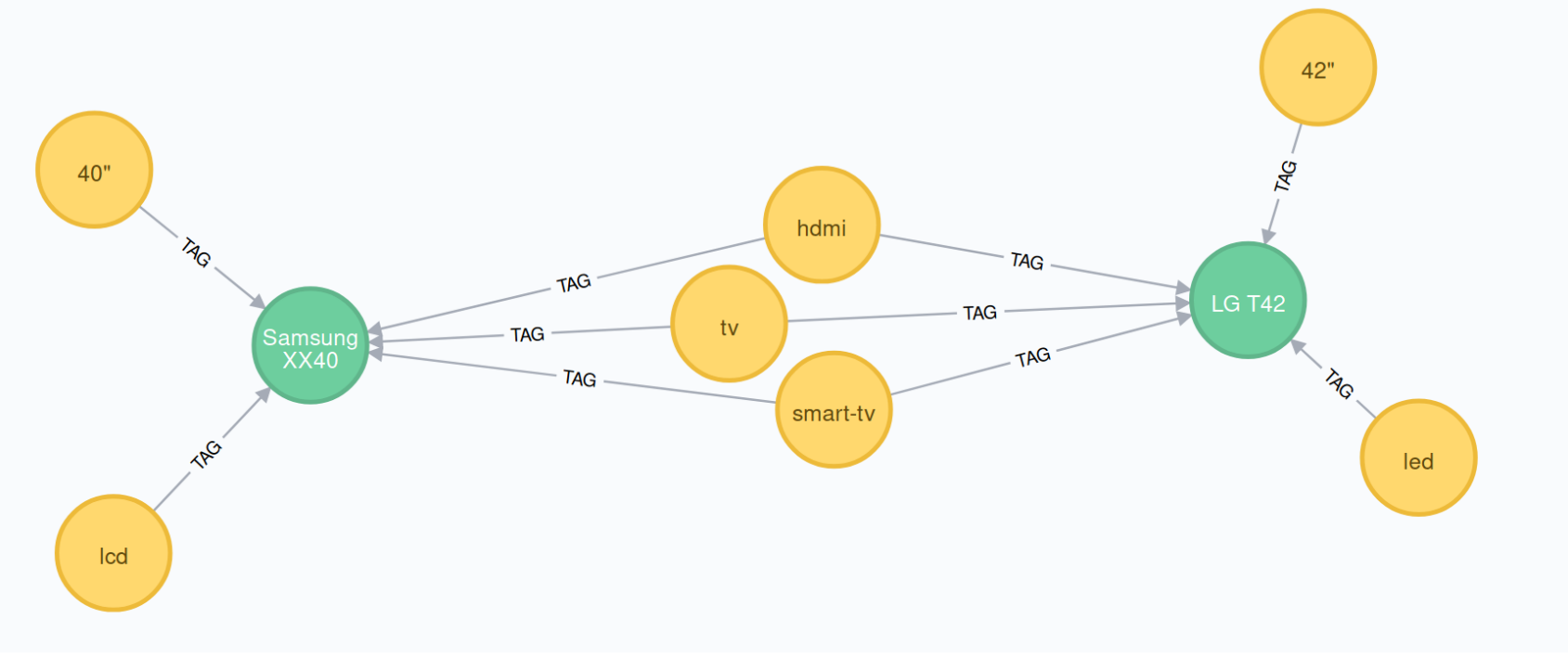
- Para crear sólo relaciones y mezclar los nodos que ya existan se suele utilizar `MERGE`
- P. ej. si alguna de mis hijas existe, no se crea. Sólo se crean las que no existen

```
MATCH (padre:Usuario { nombre: 'Diego' })
MERGE (violeta:Usuario {nombre: 'Violeta'})
MERGE (martina:Usuario {nombre: 'Martina'})
MERGE (violeta)-[:HIJO_DE]->(padre)
MERGE (martina)-[:HIJO_DE]->(padre)
```

Podemos modelar los *tags* como nodos

```
MATCH (n:Producto {nombre: 'LG T42'})
   MERGE (a:Tag {nombre:'tv'})
   MERGE (a) -[:TAG]-> (n)
   MERGE (b:Tag {nombre:'42"'})
   MERGE (b) -[:TAG]-> (n)
   MERGE (c:Tag {nombre:'led'})
   MERGE (c) -[:TAG]-> (n)
   MERGE (d:Tag {nombre:'hdmi'})
   MERGE (d) -[:TAG]-> (n)
   MERGE (e:Tag {nombre:'smart-tv'})
   MERGE (e) -[:TAG]-> (n)
```

Grafo:

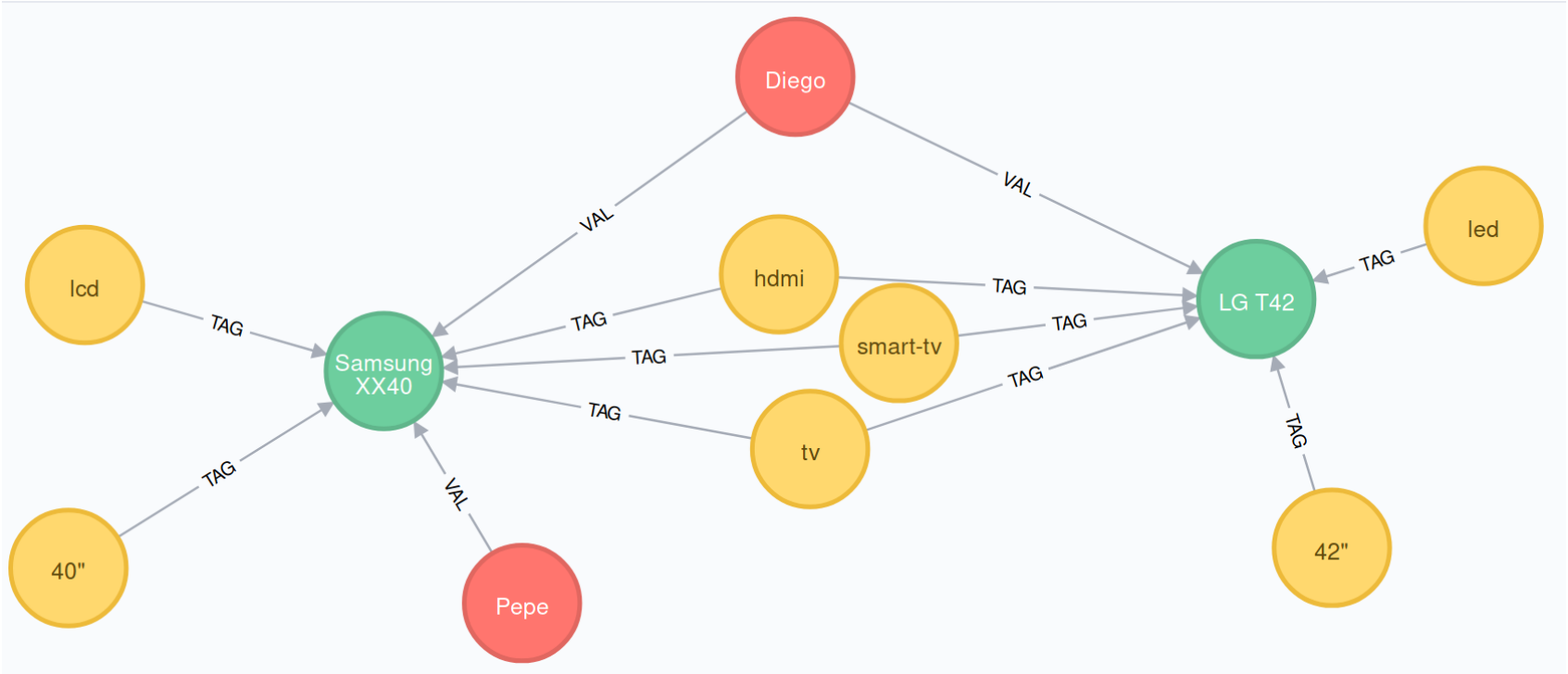

40"
TAG
Samsung XX40
TAG
lcd
hdmi
TAG
tv
TAG
smart-tv
TAG
TAG
TAG
42"
TAG
LG T42
TAG
led

También los usuarios y sus valoraciones

```
CREATE (:Usuario {nombre: 'Diego'}),
       (:Usuario {nombre: 'Pepe'})
```

```
MATCH (u:Usuario {nombre: 'Diego'}),
      (p:Producto {nombre: 'Samsung XX40'})
MERGE (u)-[:VAL {estrellas:5, comment: "Like!"
   }]->(p)
```

Grafo:

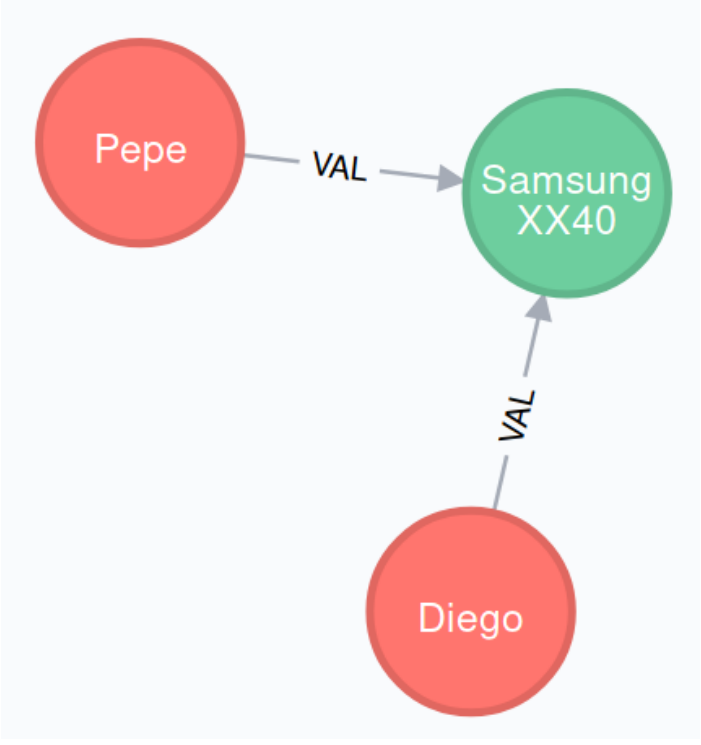

Diego
VAL
VAL
led
TAG
lcd
TAG
hdmi
TAG
LG T42
TAG
smart-tv
TAG
TAG
Samsung
XX40
TAG
TAG
tv
TAG
TAG
VAL
42"
40"
Pepe

Camino más corto entre dos usuarios

- Puede servir para encontrar productos relacionados para sugerir a cada uno
- Se puede *aumentar* con información geográfica, información de amistad, etc.

```
MATCH p=shortestPath( (u1:Usuario {nombre: "Diego
    "})-[*]-(u2:Usuario {nombre: "Pepe"}) )
    RETURN p
```

Camino más corto entre dos usuarios (II)



Grafos: Casos de uso

- Tratamiento de información geográfica
  - Planificación de rutas
  - Optimización de rutas, combustible, etc.
- Conexiones sociales
  - Amistad, proximidad geográfica, grupo de edad
- Integración de fuentes dispares de datos
  - Integrar información geográfica con información meteorológica
- Información fuertemente conectada, encontrar conexiones
  - Papeles de Panamá...
- Sistemas de recomendación