

# Tutorial tecnologías NoSQL<sup>1</sup>

JISBD 2017, La Laguna, Tenerife

Diego Sevilla Ruiz

Dpto. Ingeniería y Tecnología de Computadores  
Facultad de Informática  
Universidad de Murcia

dsevilla@um.es

Julio de 2017

---

<sup>1</sup><https://github.com/dsevilla/jisbd17-nosql>

# ¿Qué veremos aquí?

¿Qué es NoSQL?

¿Por qué NoSQL?

Introducción a NoSQL y modelado

Desventajas de NoSQL

# ¿Qué veremos aquí? (II)

## Tipos de BBDD NoSQL

- ▶ Key/Value
- ▶ Documentos
- ▶ Columnares (*wide column*)
- ▶ Grafos

## Academia y NoSQL

## NewSQL

# ¿Qué *NO* veremos aquí?

SQL vs. NoSQL

Al menos no una batalla

Tecnologías en profundidad

Por falta de tiempo

Cuestiones avanzadas

Replicación, seguridad, etc.

# Pero antes, un inciso

- ▶ Imaginemos que quiero hacer una BBDD de estas diapositivas
- ▶ Imaginemos que uso NoSQL:

```
$ docker pull mongo
$ docker run --rm -d --name mongo -p
27017:27017 mongo
```

- ▶ Imaginemos que uso Python



## Pero antes, un inciso (II)

- ▶ Quiero crear la base de datos "presentations" y la colección "jisbd17"

```
import pymongo
from pymongo import MongoClient
client = MongoClient("localhost", 27017)
```

```
db = client.presentations
jisbd17 = db.jisbd17 # colección jisbd17
```

# Pero antes, un inciso (III)

- ▶ Datos para una diapositiva:

```
jisbd17.insert_one(  
    {'_id': 'jisbd17-000',  
     'title': 'blah',  
     'text': '',  
     'image': None,  
     'references':  
         [ {'type': 'web',  
             'ref': 'http://nosql-database.org'  
                 },  
             { 'type' : 'book',  
               'ref': 'Sadalage, Fowler. NoSQL  
                     Distilled, 2009'}  
         ],  
     'xref': ['jisbd17-010', 'jisbd17-002'],  
     'notes': 'blah blah'  
 })
```

# Pero antes, un inciso (IV)

- ▶ ¡Las imágenes!

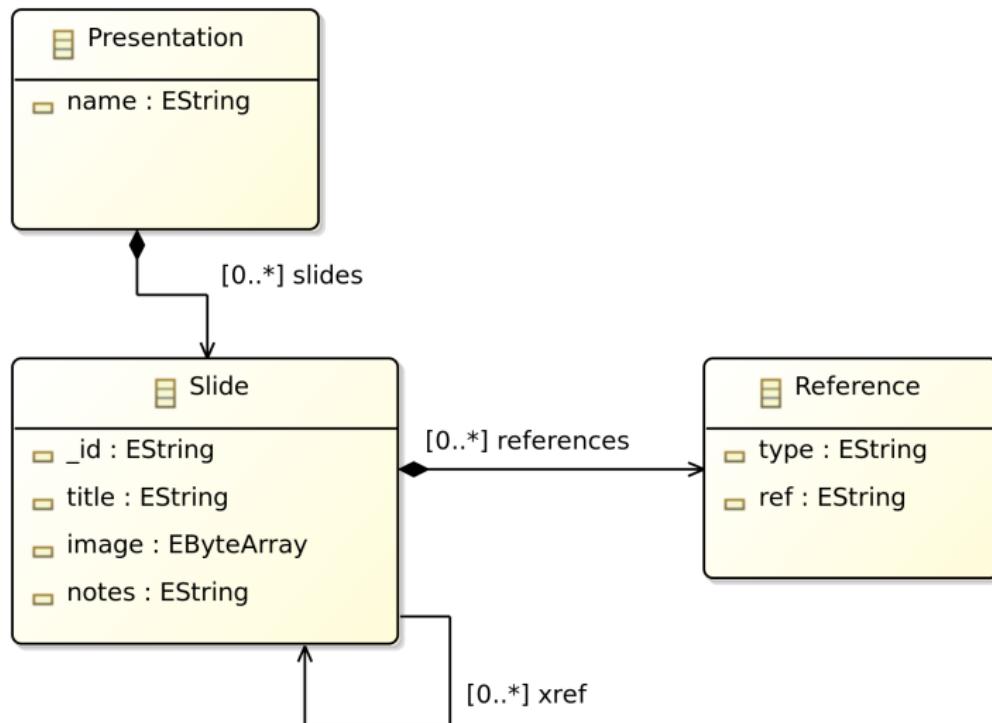
```
import os
import glob
files = glob.glob('slides/slides-dir/*.png')
for file in files:
    img = load_img(file)
    img_to_thumbnail(img)
    slidename = os.path.basename(
        os.path.splitext(file)[0])
    jisbd17.update_one(
        {'_id': slidename},
        {'$set' : {'image':
            img_to_bytebuffer(img)}},
        True)
```

## Pero antes, un inciso (V)

- ▶ Se añaden características a los *slides*
- ▶ En el segundo bucle se **actualiza image**

¿Y el modelado de datos?

# Pero antes, un inciso (VI)



## Pero antes, un inciso (VII)

- ▶ OK, pensemos en **relacional**
  - ▶ **CREATE TABLE ...**
  - ▶ ¿Cuántas tablas?
  - ▶ Claves ajenas, restricciones de integridad...
  - ▶ **Gap semántico:**
    - ▶ La tabla **Reference** no está contenida sino **enlazada** (por clave ajena)
    - ▶ O bien se crea la tabla **Slide\_Reference**
    - ▶ **ID Artificial** para referencias
    - ▶ Se tiene que crear la tabla **Slide\_xref**

El modelo de documentos es **más natural** en este caso

# Pero antes, un inciso (VIII)

¿Y toda la información de diapositiva?

```
SELECT * FROM Slides WHERE '_id' = 'jisbd17-000';
```

- ▶ ¿xref? ¿references?

```
SELECT * FROM Slides s
JOIN References r ON 's._id' = r.slide_id
JOIN Slide_xref x ON 's._id' = x.slide_id
WHERE 's._id' = 'jisbd17-000';
```

- ▶ Filas con ¡¡información replicada!!
- ▶ ¿Y si se añade más información a **Slide**?

# Pero antes, un inciso (IX)

## MongoDB

```
slide = jisbd17.find_one({'_id': 'jisbd17-000'})
```

# Introducción a NoSQL

- ▶ **NoSQL** ⇒ *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** ⇒ *Not Only SQL* (no sólo SQL)



# NoSQL



Your Ultimate Guide to the  
Non-Relational Universe!

[including a historic [Archive](#) 2009-2011]  
News Feed covering some changes [here](#) !

**NoSQL DEFINITION:** Next Generation Databases mostly addressing [some of the points](#): being non-relational, distributed, open-source and horizontally scalable.

The original intention has been [modern web-scale databases](#). The movement began early 2009 and is growing rapidly. Often more characteristics apply such as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge amount of data and more. So the misleading term "nosql" (the community now translates it mostly with "not only sql") should be seen as an alias to something like the definition above. (based on 7 sources, 15 constructive feedback emails (thanks!) and 1 disliking comment . Agree / Disagree? [Tell](#) me so! By the way: this is a strong definition and it is out there here since 2009!)

[nosql-databases.org](http://nosql-databases.org)

## LIST OF NOSQL DATABASES [currently >225]

Core NoSQL Systems: [Mostly originated out of a Web 2.0 need]

### Wide Column Store / Column Families

[Hadoop](#) / [HBase](#) API: Java / any writer, Protocol: any write call, Query Method: MapReduce Java / any exec, Replication: HDFS Replication, Written in: Java, Concurrency: ?, Misc: Links: 3 Books [[1](#), [2](#), [3](#)]

[MapR](#), [Hortonworks](#), [Cloudera](#) Hadoop Distribution and professional services .

[Cassandra](#) massively scalable, partitioned row store, masterless architecture, linear scale performance, no single points of failure, read/write support across multiple data centers & cloud availability zones. API / Query Method: CQL and Thrift, replication: peer-to-peer, written in: Java, Concurrency: tunable consistency, Misc: built-in data compression, MapReduce support, primary/secondary indexes, security features. Links: [Documentation](#), [PlanetC\\*](#), [Company](#).

[Scylla](#) Cassandra-compatible column store, with consistent low latency and more transactions per second. Designed with a thread-per-core model to maximize performance on modern multicore

### NoSQL RELATED EVENTS:

- 12/14 Sept [Flink Forward](#) »
- 26 Sept [Big Data Guide Event Frankfurt](#) »



**Flink Forward**

BERLIN 12/13 OCT 2015

Register your event 4free: [»](#)

### NoSQL ARCHIVE



 **ArangoDB**  
the multi-model NoSQL DB

### NoSQL FORUMS

- Global NOSQL Forum [»](#)
- Forum Berlin [»](#)
- Forum France [»](#)
- Forum Japan [»](#)



# NoSQL

APACHE  
**HBASE**



CouchDB  
relax

Cassandra

riak



mongoDB

ACCUMULO™

RAVENDB

Couchbase

HYPERTABLE<sup>INC</sup>

Neo4j



redis

amazon  
DynamoDB

MarkLogic™

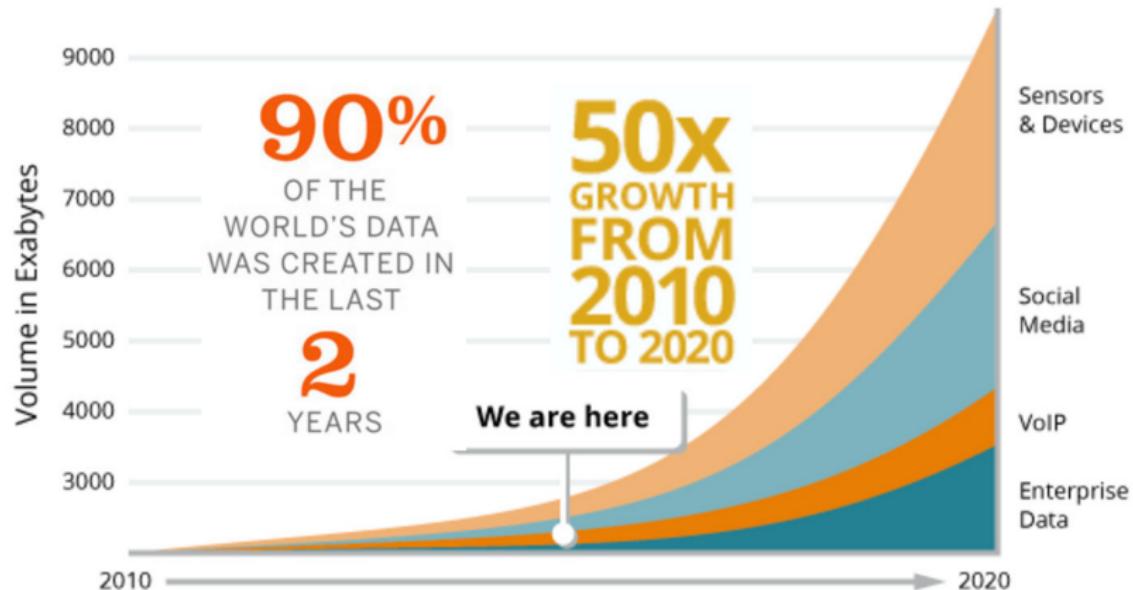
SCYLLA

OrientDB®

# 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 ⇒ *Schemaless*
- ▶ Surgidos de la comunidad para solucionar problemas, y muchas de ellas son *libres/open source*
- ▶ Diseño basado en **procesamiento distribuido**
- ▶ Principios funcionales ⇒ *MapReduce*

# NoSQL: Características (II)

## Categorías de NoSQL

- ▶ Bases de datos Key-Value
- ▶ Bases de datos Documentales
- ▶ Bases de datos columnares
- ▶ Bases de datos de grafos
- ▶ Bases de datos de arrays



# Evolución desde el modelo relacional

- ▶ El modelo relacional ⇒ predominante en los últimos ~30 años
- ▶ Tiene sus raíces en el denominado *business data processing*, procesamiento de transacciones y *batch*
- ▶ Propuesto por Codd en los 70, de alto nivel
- ▶ Actualmente los **sistemas SQL** están muy optimizados:
  - ▶ el grado de implantación es mayoritario
  - ▶ para el 99 % de los problemas (que caben en un ordenador) es eficiente y adecuado

# Adopción de NoSQL

Twitter cambiando a Cassandra por 2010  
Cassandra desarrollada en Facebook en 2009

TRENDING: What you need to know about Windows 10's new upgrade · First impressions: iPad Pro 'wicked fast' · Resources/White Papers



Popular Now: ▾



Home > Enterprise Applications > Database Management

## NEWS

# Twitter growth prompts switch from MySQL to 'NoSQL' database

Hopes open source technology can further boost improving Twitter uptime record

By Eric Lai

Computerworld | Feb 23, 2010 4:10 PM PT

## RELATED TOPICS

Database Management

Enterprise Applications

Ryan King, an engineer at Twitter, today told the blog MyNoSQL that the social networking company plans to move from MySQL to the [Cassandra database](#) for what he called its resilience, scalability and large community of open-source developers.

## MORE LIKE THIS

The end of SQL and relational databases? (part 3 of 3)

InfoWorld's top 10 emerging enterprise technologies

InfoWorld review: Databases primed for social networks



# Adopción NoSQL. Ranking julio 2017

Fuente: <http://db-engines.com/en/ranking/>

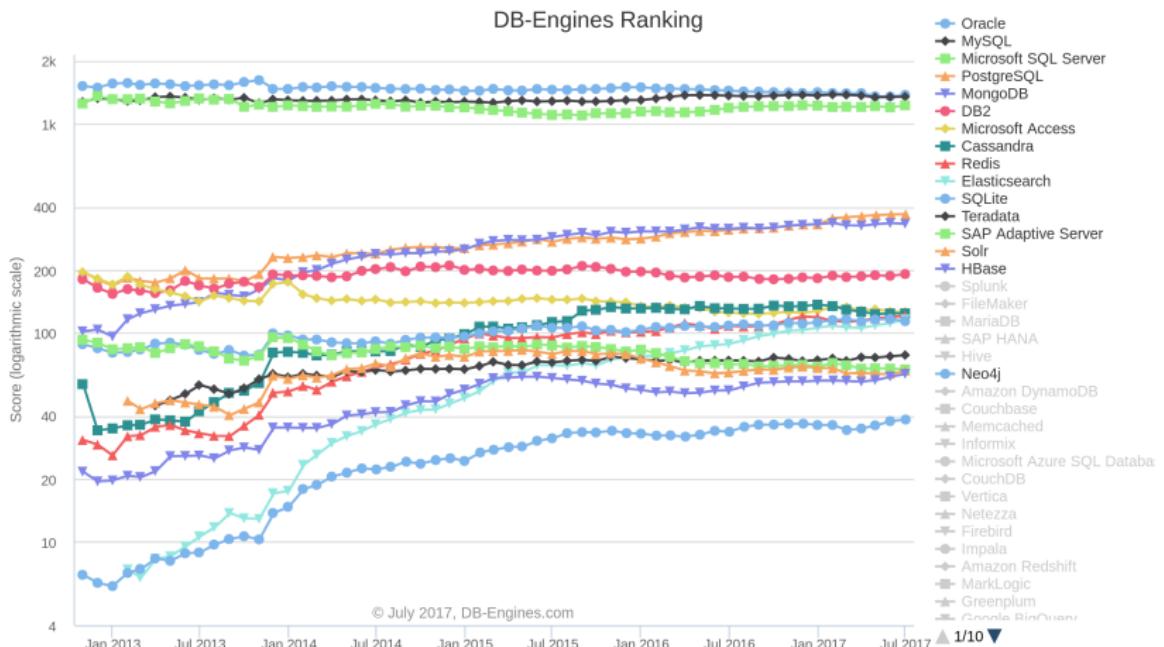
328 systems in ranking, July 2017

Rank				DBMS	Database Model	Score		
	Jul 2017	Jun 2017	Jul 2016			Jul 2017	Jun 2017	Jul 2016
1.	1.	1.	Oracle	Relational DBMS	1374.88	+23.11	-66.65	
2.	2.	2.	MySQL	Relational DBMS	1349.11	+3.80	-14.18	
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1226.00	+27.03	+33.11	
4.	4.	↑5.	PostgreSQL	Relational DBMS	369.44	+0.89	+58.28	
5.	5.	↓4.	MongoDB	Document store	332.77	-2.23	+17.77	
6.	6.	6.	DB2	Relational DBMS	191.25	+3.74	+6.17	
7.	7.	↑8.	Microsoft Access	Relational DBMS	126.13	-0.42	+1.23	
8.	8.	↓7.	Cassandra	Wide column store	124.12	-0.00	-6.58	
9.	9.	↑10.	Redis	Key-value store	121.51	+2.63	+13.48	
10.	↑11.	↑11.	Elasticsearch	Search engine	115.98	+4.42	+27.36	
11.	↓10.	↓9.	SQLite	Relational DBMS	113.86	-2.84	+5.33	
12.	12.	12.	Teradata	Relational DBMS	78.37	+1.04	+4.43	
13.	13.	13.	SAP Adaptive Server	Relational DBMS	66.91	-0.61	-3.82	
14.	14.	14.	Solr	Search engine	66.02	+2.41	+1.33	
15.	15.	15.	HBase	Wide column store	63.62	+1.75	+10.48	
16.	16.	↑18.	Splunk	Search engine	60.30	+2.78	+13.65	
17.	17.	↓16.	FileMaker	Relational DBMS	58.65	+1.57	+7.09	
18.	18.	↑20.	MariaDB	Relational DBMS	54.36	+1.47	+18.56	
19.	19.	19.	SAP HANA	Relational DBMS	47.94	+0.45	+6.14	
20.	20.	↓17.	Hive	Relational DBMS	46.20	+1.82	-1.34	
21.	21.	21.	Neo4j	Graph DBMS	38.52	+0.65	+4.83	
22.	22.	↑25.	Amazon DynamoDB	Document store	36.46	+2.44	+11.53	
23.	23.	↑24.	Couchbase	Document store	33.02	+1.10	+7.04	



# Adopción NoSQL.Tendencia julio 2017

Fuente: <http://db-engines.com/en/ranking/>

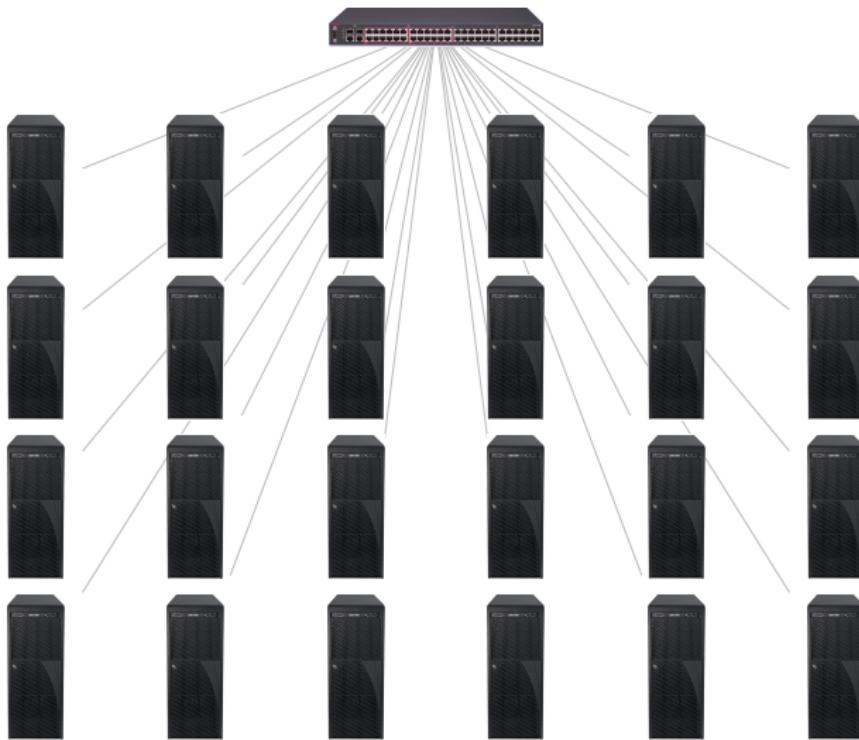


# Adopción NoSQL. Análisis

## Análisis

- ▶ Dominan los grandes SGBDR
- ▶ El *Open Source* tiene una importancia crucial (MySQL, MongoDB, etc.)
- ▶ Varias bases de datos NoSQL entre las 10 primeras. Muchas en las 20 primeras
- ▶ La distancia entre los grandes SGBDR y el primer NoSQL (MongoDB) es de 5×
- ▶ Paradigmas más “atrevidos” como el de grafos están entre los 20 primeros (Neo4j)

# Cambio de perspectiva: Red



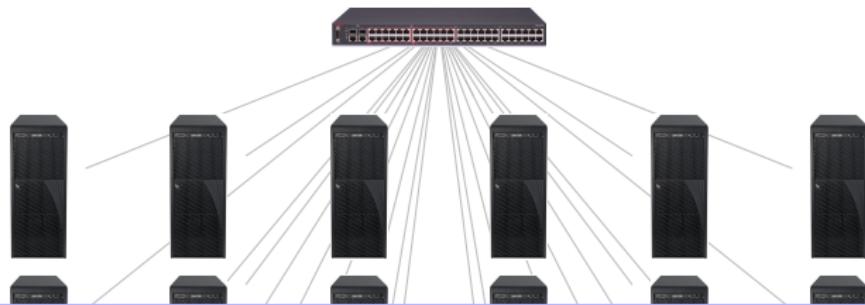
# Cambio de perspectiva: Red



## Almacenamiento distribuido

- ▶ Desde los 90's: Clústers/NOC/COW:  
procesamiento masivamente paralelo  
**SIN EMBARGO...**
- ▶ Almacenamiento no distribuido
- ▶ Ahora los nodos ⇒ también  
**almacenamiento**
- ▶ Minimizar el verdadero cuello de botella:  
trasiego de información por la red

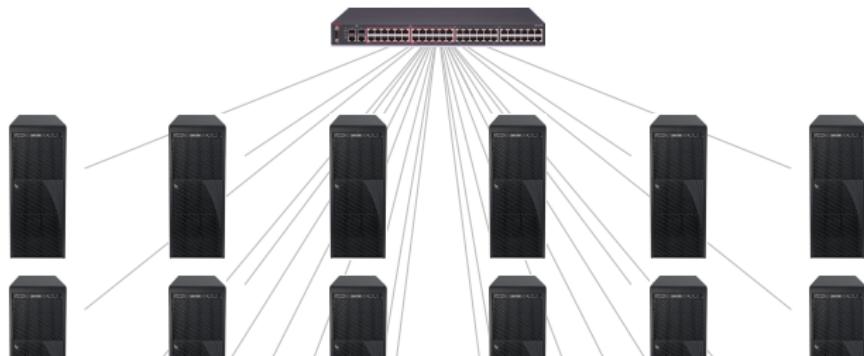
# Cambio de perspectiva: Red



## Procesamiento distribuido

- ▶ Necesidad de paralelización máxima
- ▶ **Escalabilidad**
- ▶ Explotación de la **localidad de los datos**:
  - ▶ Datos producidos se utilizan en siguientes iteraciones
  - ▶ Datos recibidos directamente (clientes simultáneos)

# 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

- ▶ El modelo relacional limita a tablas con valores primitivos y relaciones *Primary Key/Foreign Key*
- ▶ Pero en programación se utilizan **listas, arrays, tipos de datos compuestos** (*gap semántico*)
- ▶ ACID es **muy compleja** y **costosa** en ambientes distribuidos (quizá **no necesaria** en algunas aplicaciones)

# Cambio de perspectiva: Red



## Modelo de datos (ii)

- ▶ ¿Si se pudiera ver como un **GRAN ARRAY**?
  - ▶ Cada nodo almacenaría una parte del array
  - ▶ Búsqueda aleatoria **muy rápida** (árboles B)
  - ▶ Uso de **filtros de Bloom**
  - ▶ 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)

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

# 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 que haya que 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**

# Schemaless (VI)

## SIN EMBARGO...

A veces un esquema es conveniente

- ▶ Facilita el desarrollo y evita inconsistencias
  - ▶ Mongoose para MongoDB:

```
var Comment = new Schema({  
    name: { type: String, default: 'Anonymous' },  
    date: { type: Date, default: Date.now },  
    text: Buffer  
});  
// a setter with on-line modification  
Comment.path('name').set(function (v) {  
    return capitalize(v);  
});
```

# Choose wisely



SQL

NoSQL

Polyglot  
Persistence

# Modelado de datos en NoSQL

El modelado de datos debe ser:

- ▶ Realizado al mayor nivel de abstracción posible
- ▶ Independiente de la tecnología subyacente

Sin embargo, en NoSQL:

- ▶ Se tiene que tener en cuenta el diseño distribuido
- ▶ Optimización guiada por las consultas



# Modelado de datos en NoSQL (II)

Con respecto al modelo de datos:

- ▶ Se mantienen los conceptos de entidad, relación, cardinalidades, etc.
- ▶ El modelado relacional se centra en especificar **qué datos tenemos y podemos ofrecer**
- ▶ El modelo NoSQL se centra en **optimizar qué consultas vamos a servir**
- ▶ 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>



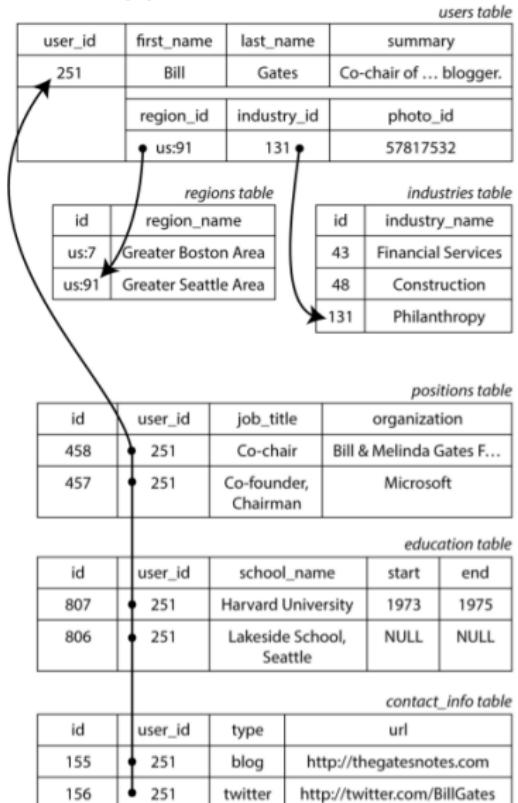
**Bill Gates**  
Greater Seattle Area | Philanthropy

**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](http://thegatesnotes.com)  
Twitter: @BillGates



# Representación de relaciones

## Relaciones uno a muchos

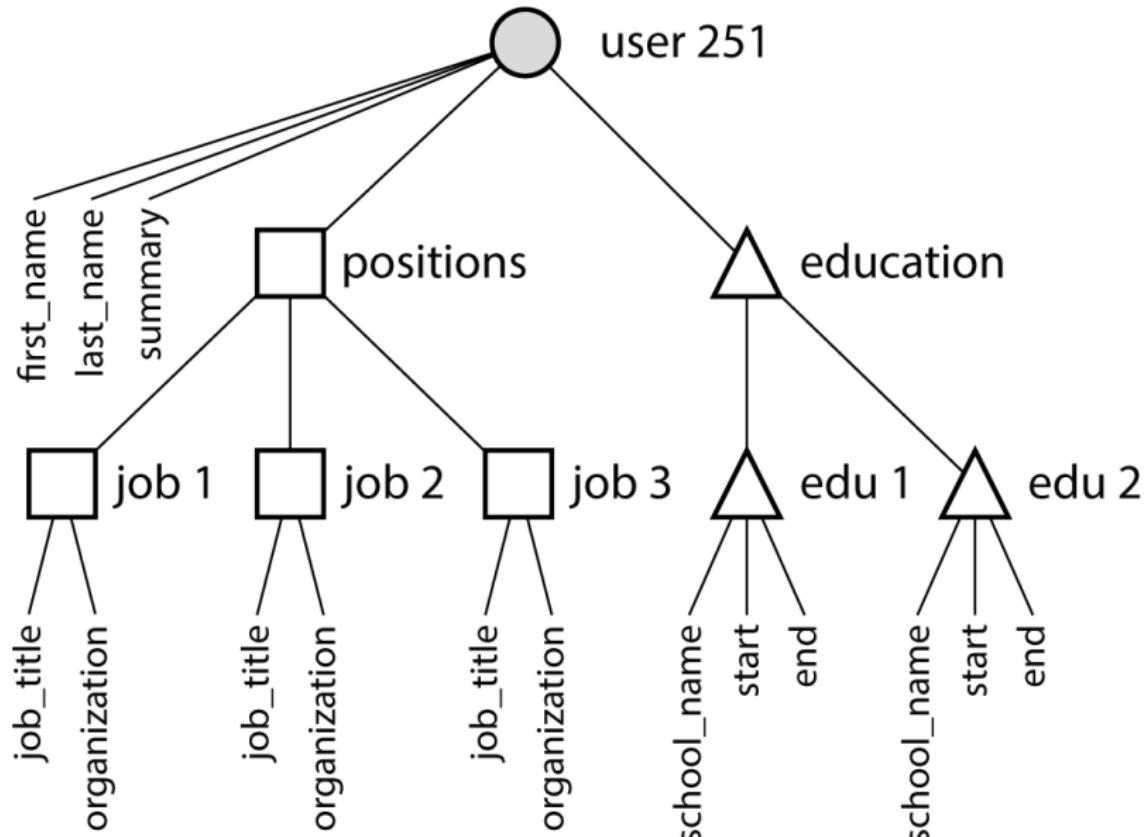
Las relaciones uno a muchos (e.g. **positions**) en el modelo relacional:

- ▶ Normalización usando varias tablas (**Positions** con **user\_id**)
  - ▶ Necesidad de más de una tabla
  - ▶ Necesidad de uso de **JOIN** ⇒ ineficiencia
- ▶ Algunos SGBDR ofrecen la posibilidad de tener tipos de datos estructurados y campos XML o JSON. (P. ej. PostgreSQL)
  - ▶ Alternativa interesante, aunque...
  - ▶ No son estándar

```
{  
    "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"  
    }  
}
```



# CV como un árbol (equivalente)



# Representación de Relaciones

## Modelo de documentos

- ▶ Modelo de documentos ⇒ analogía del array/mapa gigante
- ▶ Conjunto de documentos (objetos complejos)
  - ▶ Un identificador único, campo *id*
  - ▶ Búsqueda aleatoria eficiente por clave (*referencia*)
  - ▶ Estructura jerárquica de sub-documentos contenidos ⇒ *agregación*

Más flexibilidad que el modelo relacional  
(elección entre *referencia* y *agregación*)



# Representación de Relaciones

## Uno a muchos (ii) – NoSQL

- ▶ Relaciones Uno a Muchos (**positions**):
  - ▶ Opción 1: Agregando la tabla **positions**
  - ▶ Opción 2: Convertir las empresas en entidades, y utilizar una *referencia*
- ▶ ¿Qué opción elegir?

# Representación de Relaciones (II)

## Uno a muchos (ii) – NoSQL

¡Modelo guiado por el acceso a datos!

- ▶ Si los elementos “muchos” tienen una estructura sencilla ⇒ **Opción 1**
- ▶ Si los elementos “muchos” son usualmente recuperados en una consulta junto con el elemento “uno” ⇒ **Opción 1**
- ▶ Si los elementos “muchos” son relativamente grandes, o bien son recuperados siempre de forma separada ⇒ **Opción 2**

# Representación de Relaciones

Muchos a uno y muchos a muchos

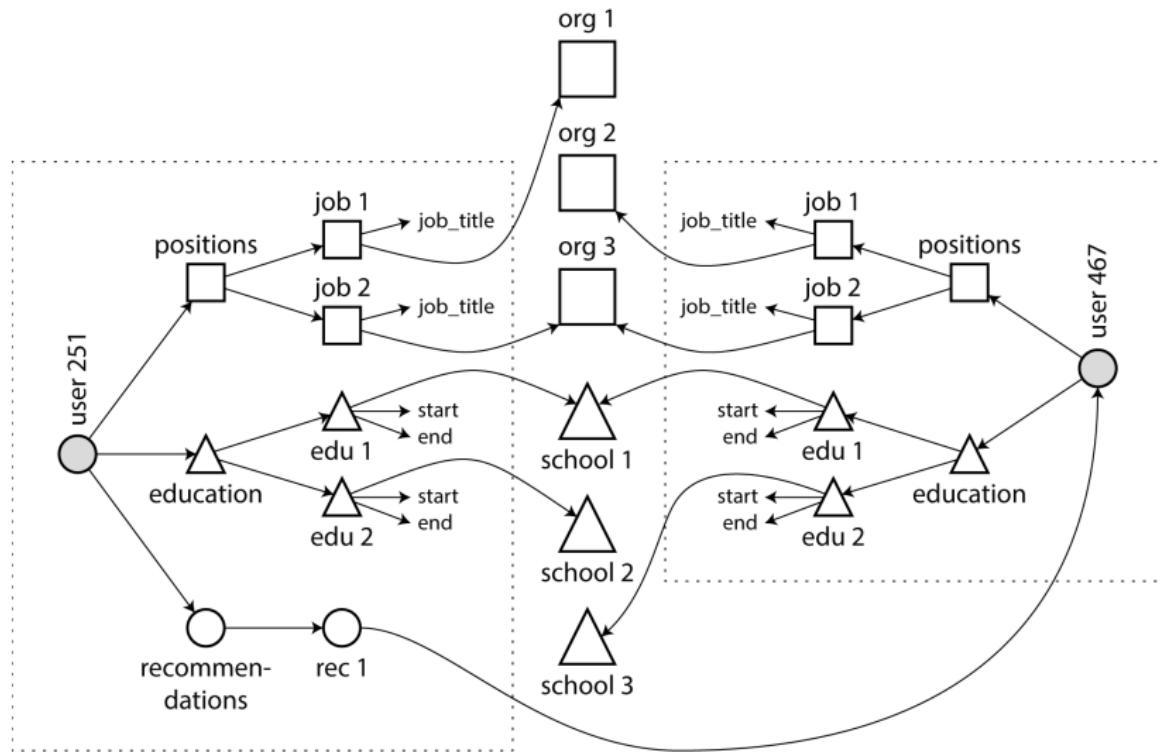
Relaciones muchos a uno y muchos a muchos:

- ▶ Personas que viven en una región
- ▶ Preguntas que refieren a Tags

El modelo de documentos no aporta ventajas

- ▶ La *agregación* daría lugar a mucha **duplicación** (y a problemas de sincronización)
- ▶ ⇒ **Referencias** (sobre el ID), similar a una FK en el modelo relacional
  - ▶ ⇒ al no haber **JOINS** la aplicación tiene que hacer más de una petición a la BD

# Muchos a muchos – referencia



# Eficiencia *raw*

- ▶ Los sistemas NoSQL tienen que competir también con los SQL en términos de eficiencia neta (también llamada *raw*)
- ▶ Un pequeño test sintético nos puede ayudar a hacernos una idea<sup>2</sup>
- ▶ La prueba se realizó sobre MongoDB y sobre MySQL (se ha adaptado el original, que era para PostgreSQL)<sup>3</sup>

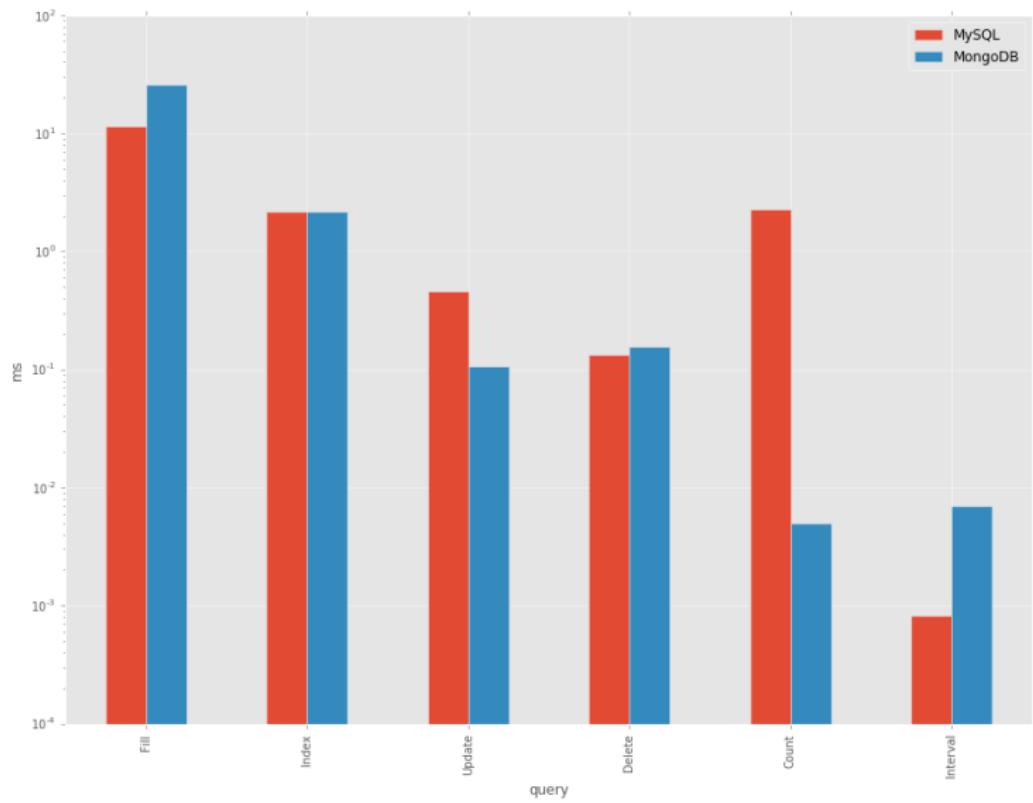
## Eficiencia raw (II)

- ▶ Se parte de una tabla sencilla con cuatro valores, que muestran medidas de sensores con localización, valor de la lectura y una marca de tiempo
- ▶ Se realizan seis pruebas que pueden corresponder a un conjunto de consultas normales:

# Eficiencia raw (III)

1. Inicialmente se insertan **un millón** de elementos generados al azar, con fechas que permitan la búsqueda por rango (**Fill**)
2. Se crea un índice en la tabla para la fecha de la lectura (**Index**)
3. Se actualizan los valores de un conjunto de entradas seleccionadas por rango de fechas (**Update**)
4. Se eliminan un conjunto de filas seleccionadas por rango de fechas (**Delete**)
5. Obtiene el número de filas restantes (**Count**)
6. Se obtiene un **subconjunto** de filas extraído de una consulta dada por un rango de fechas (**Interval**)

# Eficiencia raw (IV)



# Eficiencia raw (V)

- ▶ El gráfico tiene escala logarítmica en el eje Y (las diferencias pequeñas se acentúan)
- ▶ A simple vista, ambos están muy igualados
  - ▶ SQL (MySQL) lleva *muchos años* de optimizaciones
  - ▶ MongoDB tiene menos historia a sus espaldas en cuanto a optimizaciones, etc.
- ▶ Hay casos en los que uno es más rápido que el otro y viceversa



# Eficiencia raw (VI)

- ▶ No se puede decir cuál es mejor
  - ▶ ⇒ DEPENDE DEL PATRÓN DE ACCESOS QUE VAYA A TENER NUESTRA APLICACIÓN
  - ▶ (p. ej. contado en MongoDB mucho más rápido que en MySQL; actualización algo más rápida)

---

<sup>2</sup>[http://bonesmoses.org/2016/07/15/  
pg-phriday-a-postgres-persepctive-on-mongodb/](http://bonesmoses.org/2016/07/15/pg-phriday-a-postgres-persepctive-on-mongodb/).

<sup>3</sup>Datos disponibles en: <https://github.com/dsevilla/bdge/tree/master/addendum>

# Key-Value Stores y Documentales



# Key-Value Stores y Documentales

- ▶ A cada pieza de datos se le asigna un identificador
- ▶ La diferencia entre ambas es que:
  - ▶ 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 ⇒ formatos jerárquicos tipo JSON o XML

# Key-Value Stores y 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 complejas, mapas
- ▶ Key-Value: Riak, Redis, Memcached, LevelDB
- ▶ Documentos: CouchDB, MongoDB, OrientDB (mixta grafos)

```
{ "type": "Movie",
  "title": "Citizen Kane",
  "year": 1941,
  "director_id": "123451",
  "genre": "Drama",
  "_id": "1",
  "rating": {
    "score": 8.4,
    "voters": 310768
  },
  "prizes": [
    { "year": 1941,
      "event": "Oscar",
      "names": ["Best original screenplay"] },
    { "year": 1941,
      "event": "NY Film Critics",
      "names": ["Best screenplay"] }
  ],
  "criticisms": [
    { "journalist": "R. Brody",
      "media": "The New Yorker",
      "color": "green" }
  ]
},
{ "_id": "123451",
  "name": "Orson Welles",
  "type": "director",
  "directed_movies": ["1"],
  "acted_movies": ["1"]
},
```

```
{ "_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"
    }
  ],
  {
    "_id": "345679",
    "name": "James Vanderbilt",
    "type": "director",
    "directed_movies": ["2"]
  }
```

# Map-Reduce

Las BD Documentales (y la mayoría de las NoSQL) usan **Map-Reduce** como principal mecanismo de búsqueda y transformación

- ▶ Origen en **lenguajes funcionales**:
  - ▶ **map( )**: Ejecuta una misma función sobre todos los elementos de un conjunto  
`map :: (a -> b) -> [a] -> [b]`
  - ▶ **reduce( )**: Procesa un conjunto de valores para producir un valor de salida  
`foldl :: (a -> b -> a) -> a -> [b] -> a`

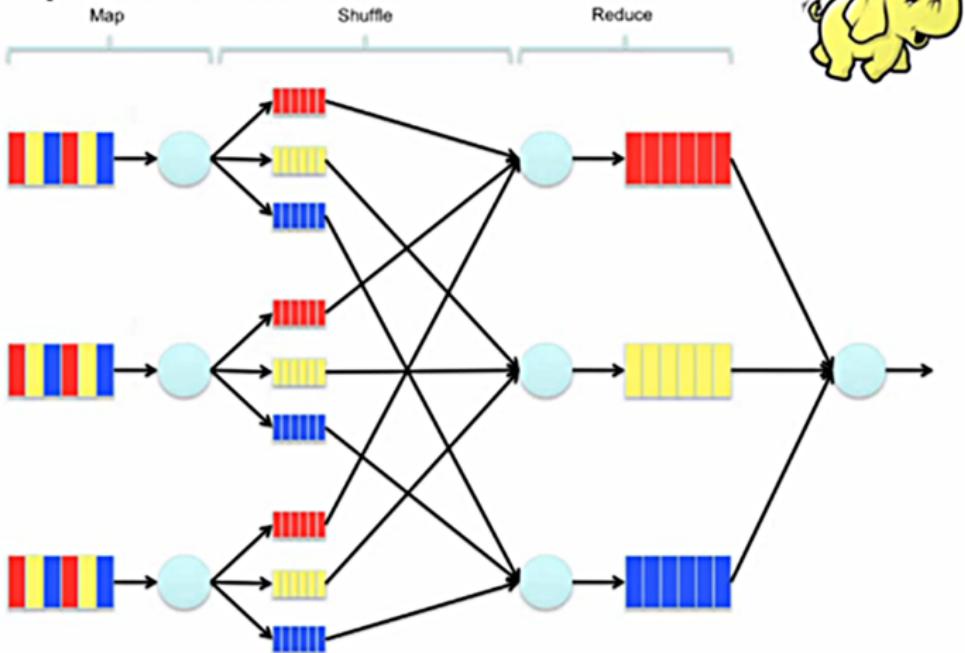


# Map-Reduce (II)

- ▶ 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 simultáneamente
  - ▶ Con los resultados parciales de cada nodo, una función `reduce()` genera un resultado (o un conjunto de resultados) final
  - ▶ Hay un proceso intermedio de *shuffle* para agrupar valores relacionados antes del `reduce()`
  - ▶ Resultados parciales en el mismo nodo (localidad) ⇒ procesamientos **en cadena**

# Map-Reduce (III)

## MapReduce Overview

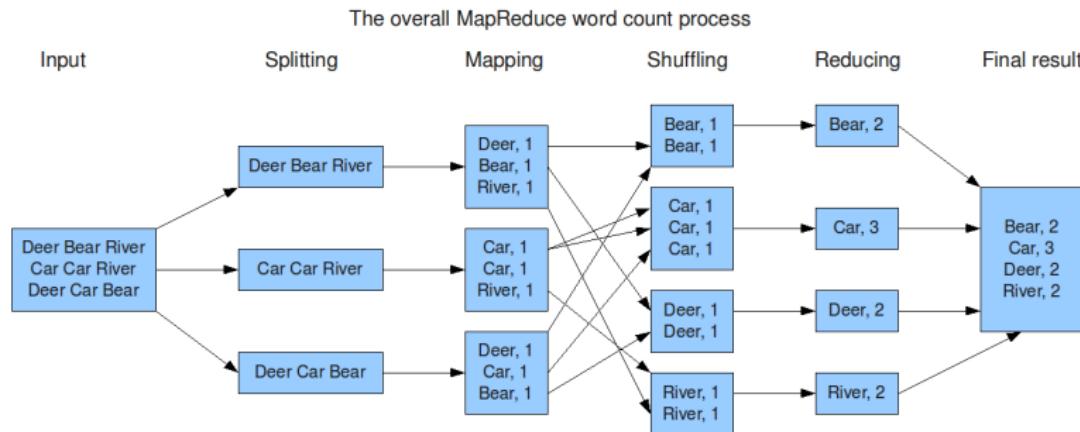


(adapted from <http://code.google.com/p/mapreduce-framework/wiki/MapReduce>)



# Map-Reduce (IV)

(de <http://www.milanor.net/blog/an-example-of-mapreduce-with-rmr2/>)



# Map-Reduce

- ▶ Map-Reduce en entornos Big-Data/NoSQL
  - ▶ Entrada  $\Rightarrow$  siempre pares  $\langle key, value \rangle$
  - ▶ **map()** produce otro conjunto de valores  
 $\{ \langle key1, value1 \rangle, \langle key2, value2 \rangle, \dots \}$
  - ▶ *Shuffle* agrupa los valores con la misma clave:  
 $\{ \langle key1, \{ val1, val3, \dots \} \rangle, \langle key2, \{ val2, val4, \dots \} \rangle, \dots \}$
  - ▶ **reduce()** procesa cada lista de valores con la misma clave, y produce otros elementos  
 $\langle key', value' \rangle$
  - ▶ Hay procesamientos difíciles de expresar en Map-Reduce  $\Rightarrow$  varias operaciones M/R en cadena



# Map-Reduce y Consultas

- ▶ Map-Reduce puede usarse no sólo para computación distribuida, sino también como una **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 y 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"  
    }  
);
```



# Map-Reduce y Consultas (III)

- ▶ MongoDB ofrece además un API alternativo para funciones de agregación:

```
db.observations.aggregate([
  { $match: { family: "Sharks" } },
  { $group: {
    _id: {
      year: { $year: "$observationTimestamp" },
      month: { $month: "$observationTimestamp" }
    },
    totalAnimals: { $sum: "$numAnimals" }
  }
});

```

# MySQL

```

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

```

- 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

# MongoDB

```

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);

```

# Uso de MongoDB

- ▶ MongoDB permite el uso desde su *shell* o bien desde clientes remotos

```
$ mongo
```

```
MongoDB shell version v3.4.6
```

```
connecting to: mongodb://127.0.0.1:27017
```

```
MongoDB server version: 3.4.6
```

```
>
```

- ▶ El shell es básicamente JavaScript, con modificaciones para acceder a la base de datos

# Uso de MongoDB (II)

- ▶ La variable **db** siempre está disponibles, y guarda la base de datos especificada con `use base-de-datos`
- ▶ Después, `db.colección` crea la colección “colección”
- ▶ Dentro de cada colección hay un conjunto de documentos, identificados por su campo “`_id`”

# Uso desde Pymongo

- ▶ Para el uso remoto utilizaremos `pymongo`, una biblioteca para Python:  
`sudo pip2 install pymongo`
- ▶ Los nombres de los métodos de las colecciones son muy similares a las equivalentes de Javascript
- ▶ Docs: <http://api.mongodb.com/python/current/api/pymongo/collection.html>

# Métodos de búsqueda

- ▶ El método de búsqueda principal es `find()`, que tiene muchas opciones
- ▶ En general permite especificar:
  - ▶ El filtro de búsqueda
  - ▶ Ordenación de resultados por algún campo
  - ▶ Proyección para no obtener todos los campos del documento
  - ▶ Número de resultados máximo (*limit*)
  - ▶ Número de elementos iniciales a ignorar (*skip*)
  - ▶ El tamaño del *batch*
- ▶ Como la variabilidad es muy grande, veremos ejemplos en la hoja Jupyter Notebook y también en la documentación

# Métodos de búsqueda

- ▶ La función `find()` tiene un gran número de posibilidades para especificar la búsqueda. Se pueden utilizar cualificadores complejos como:
  - ▶ `$and`
  - ▶ `$or`
  - ▶ `$not`
- ▶ Estos calificadores unen “objetos”, no valores



# Métodos de búsqueda (II)

- ▶ Por otro lado, hay otros calificadores que se refieren a valores:
  - ▶ `$lt` (menor)
  - ▶ `$lte` (menor o igual)
  - ▶ `$gt` (mayor)
  - ▶ `$gte` (mayor o igual)
  - ▶ `$regex` (expresión regular)

```
jisbd17.find_one({'text': {'$regex' : '[Mm]ongo',
                           }})[ '_id' ]
```



# Métodos de inserción y actualización

- ▶ También ofrece métodos para inserción y actualización:
  - ▶ `insert_one()`, `insert_many()` (batch)
  - ▶ `update_one()` – Permite actualizar un objeto con nuevos campos. El objeto se crea si se pone el parámetro `upsert` a `True`
  - ▶ `update_many()` – Permite poner nuevos valores calculados a un conjunto de objetos

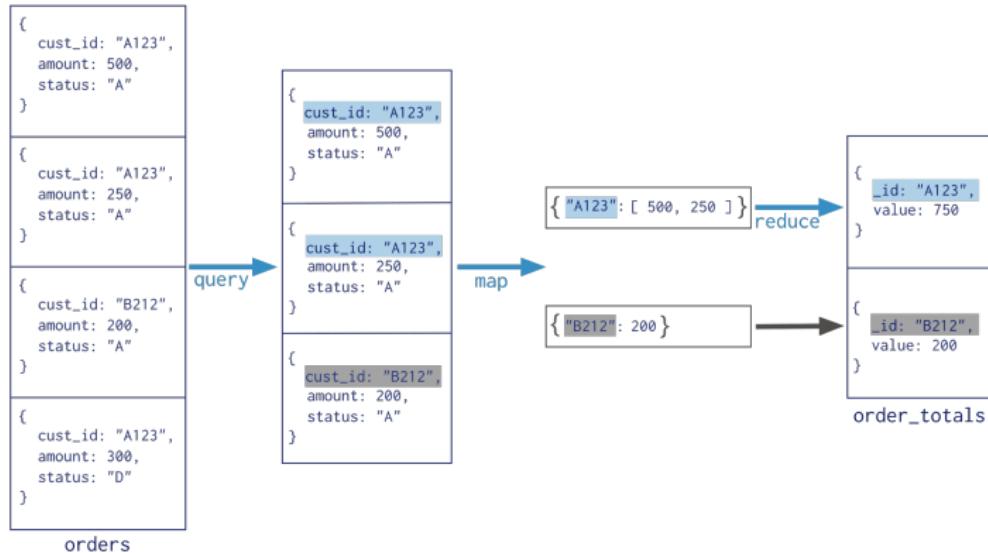


# Índices, `explain()`

- ▶ La llamada `.explain()` a cualquier búsqueda muestra el plan de ejecución
- ▶ Se puede crear un índice si la búsqueda por ese campo va a ser crítica
- ▶ Se pueden crear más índices:
  - ▶ `ASCENDING`
  - ▶ `DESCENDING`
  - ▶ `HASHED`
  - ▶ y geoespaciales

# Map-Reduce

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



# Map-Reduce

- ▶ La función `map_reduce()` de cada colección permite ejecutar procesamientos que guardan los resultados en otra colección o la devuelven en vivo
- ▶ Como el código se ejecuta dentro del motor de la base de datos, hay que enviarle el código en Javascript
- ▶ Por ejemplo, cálculo de histograma de tamaño del texto de las diapositivas:



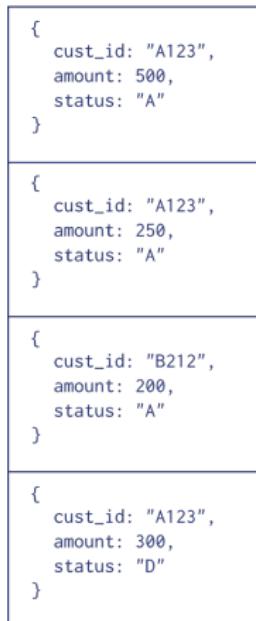
# Map-Reduce (II)

```
from bson.code import Code
map = Code(
    '''function () {
        if ('text' in this)
            emit(this.text.length, 1)
        else
            emit(0,1)
    }''')
reduce = Code(
    '''function (key, values) {
        return Array.sum(values);
    }''')
results = jisbd17.map_reduce(map, reduce, "myresults")
```

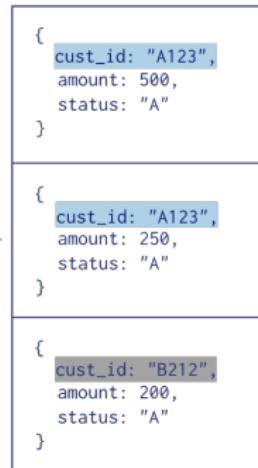
# Framework de agregación

Collection

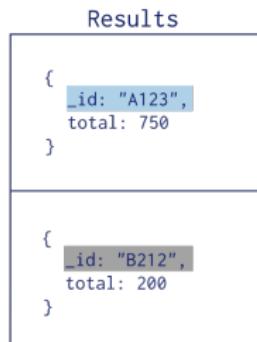
```
db.orders.aggregate( [  
    $match stage → { $match: { status: "A" } },  
    $group stage → { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }  
]
```



\$match



\$group



Results

orders

# Framework de agregación

Framework de agregación:

<https://docs.mongodb.com/manual/reference/operator/aggregation/>.

```
jisbd17.aggregate( [ { '$project' : { 'Id' : 1 } },  
                      { '$limit': 20} ])
```

# Framework de agregación (II)

```
hbase_by_length = jisbd17.aggregate( [  
    { '$project': {  
        'text' : { '$ifNull' : ['$text', '' ] }  
    },  
    { '$project' : {  
        'id' : { '$strLenBytes' : '$text' },  
        'value' : { '$literal' : 1 }  
    },  
    { '$group' : {  
        '_id' : '$id',  
        'count' : { '$sum' : 'value' }  
    },  
    { '$sort' : { '_id' : 1 }}  
])
```

# Framework de agregación (III)

Se pueden añadir etapas. En particular, por ejemplo, se puede **filtrar** inicialmente añadiendo al principio:

```
{ '$match': { 'text' : { '$regex': 'HBase' }}}}
```

La construcción **\$lookup** permite el acceso a otra (o a la misma) colección. Es equivalente a un **JOIN**



# Framework de agregación (IV)

P. ej., listar los títulos de las transparencias referenciadas (además de su `_id`)

```
jisbd17.aggregate( [  
    { '$lookup' : {  
        "from": "jisbd17",  
        "localField": "xref",  
        "foreignField": "_id",  
        "as": "xrefTitles"  
    }},  
    { '$project' : {  
        '_id' : True,  
        'xref' : True,  
        'xrefTitles.title' : True  
    }}])
```

# Bases de Datos Columnares

- ▶ Influenciadas por el Paper de Google de 2006 sobre BigTable<sup>4</sup>
- ▶ 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*)

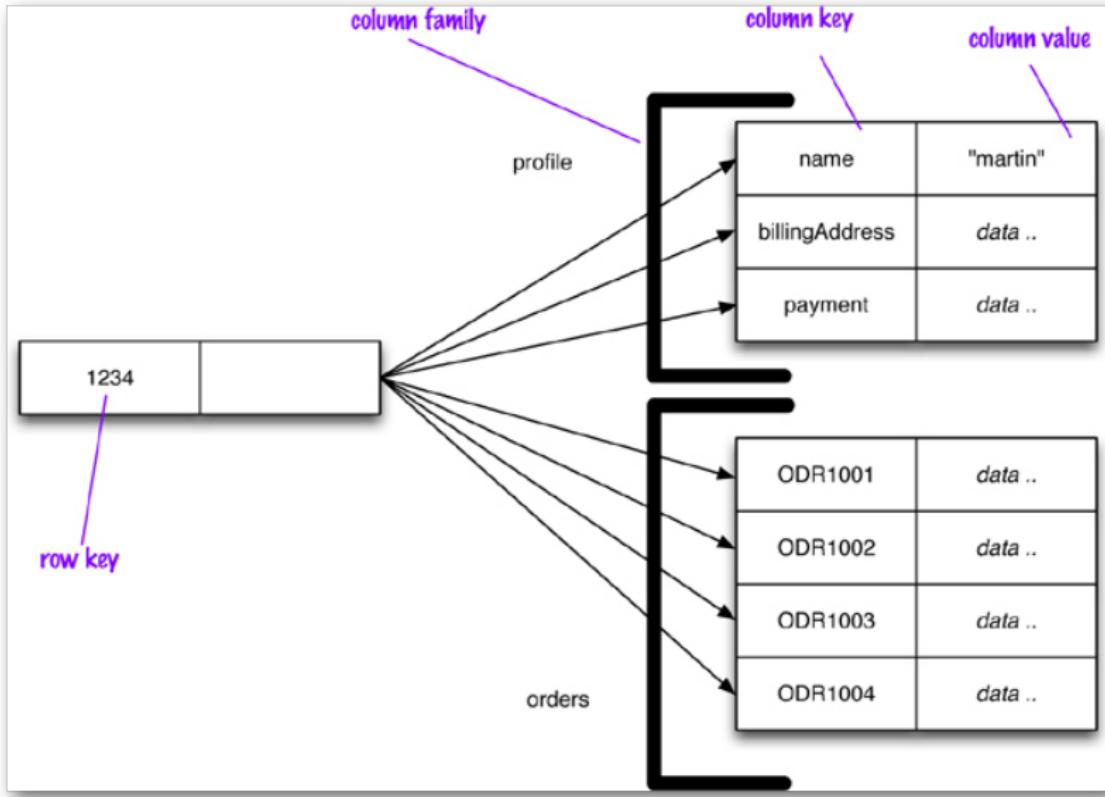


# Bases de Datos Columnares (II)

- ▶ 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

---

<sup>4</sup>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,



# Introducción a HBase



Base de datos *wide column store*

- ▶ Importante: Permite variabilidad en el número de columnas de cada fila
- ▶ Crecer 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**

# HBase

HBase is a **sparse**,  
**distributed**,  
**persistent**,  
**multi-dimensional**  
**sorted map** (or Key/Value)  
**store**

# ¿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)



# Tecnologías habilitadoras de HBase (II)

- ▶ Cada grupo de columnas (llamado *Column Family*) se almacena en un **Store** (partición vertical)
- ▶ El **Store** tiene una parte en memoria (**MemStore**) y, opcionalmente, un almacenamiento en disco, (**HFile**)
- ▶ Los **HFile** se distribuyen usando HDFS para lograr replicación y tolerancia a fallos



# HDFS

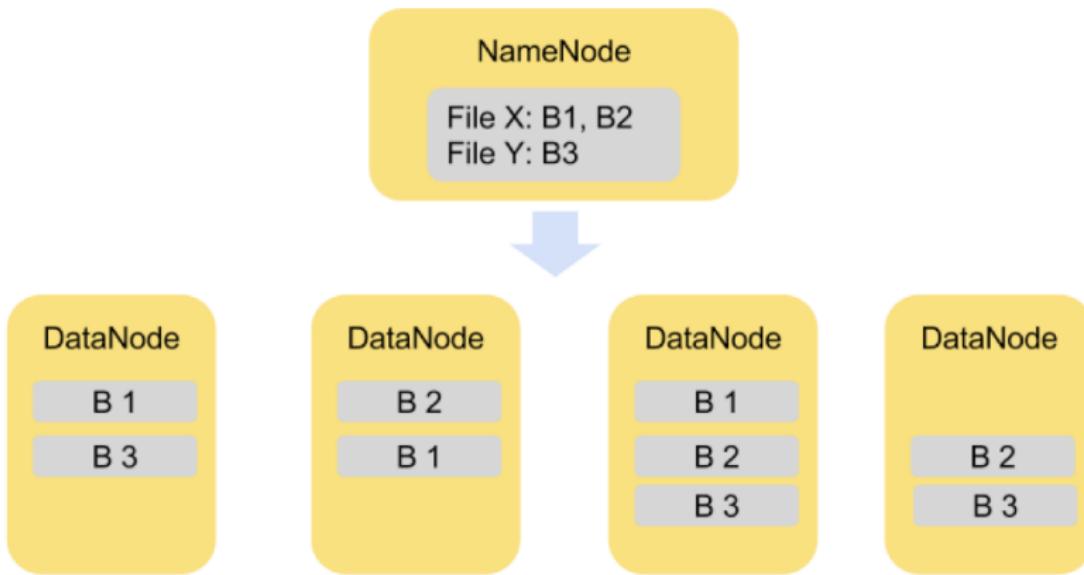
HDFS posee una arquitectura distribuida:

- ▶ **NameNode** – Almacena información sobre qué partes (*Chunks*) tiene cada fichero, y dónde están almacenadas (y replicadas)
- ▶ **Secondary Namenode** – Sustituto del NameNode en caso de fallo
- ▶ **DataNodes** – Almacenan los *chunks* de cada fichero. Cada *chunk* puede estar replicado un número de veces, dependiendo de la configuración de HDFS



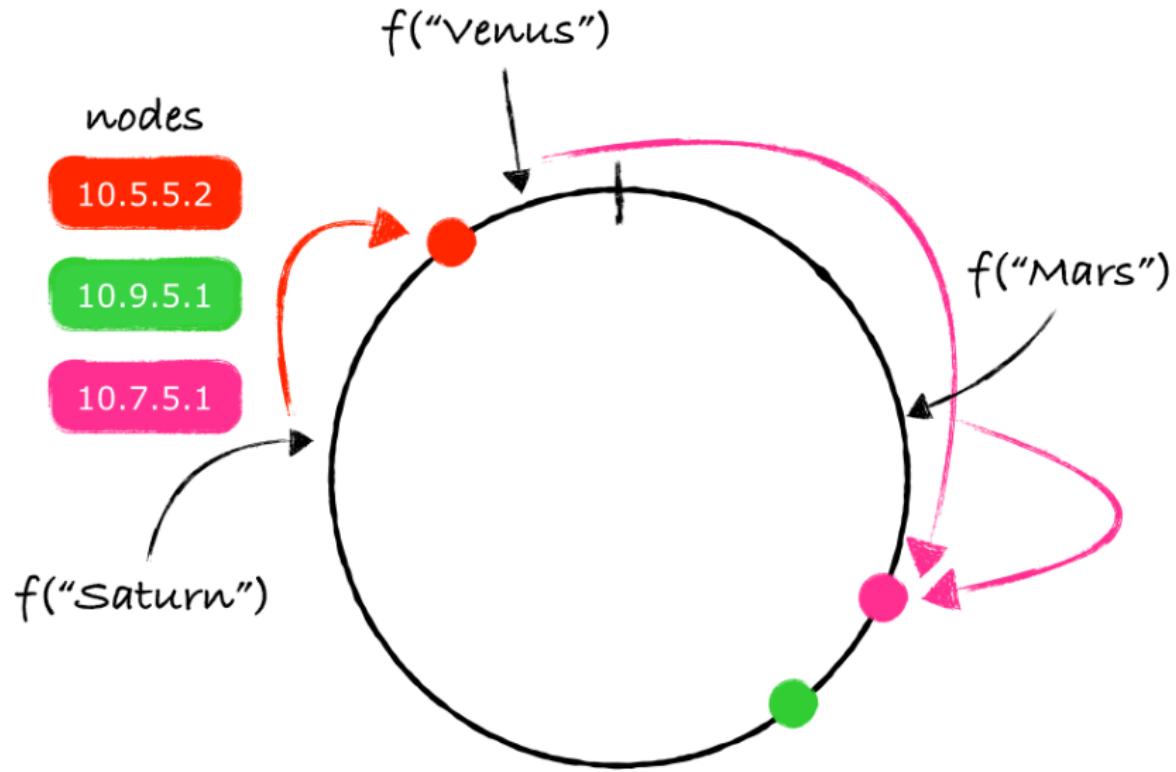
# HDFS (II)

- ▶ Zookeeper – Se encarga de mantener una consistencia de *clúster* (saber qué nodos hay conectados y activos) y sincronización de datos

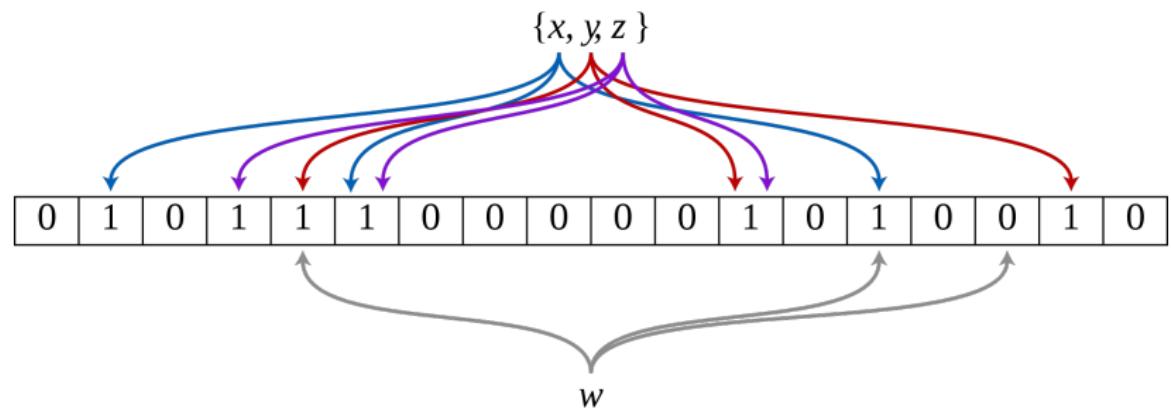


# Inciso: Consistent Hashing

<http://blog.carlosgaldino.com/consistent-hashing.html>



# Inciso: Bloom Filters



# Hbase Shell

## Comandos básicos CRUD

- ▶ Versión, estado:

```
hbase(main):002:0> version
1.1.3,
      r72bc50f5fafeb105b2139e42bbe3d61ca724989
      , Sat Jan 16 18:29:00 PST 2016
hbase(main):001:0> status
3 servers, 0 dead, 0.3333 average load
```

# Hbase Shell (II)

## Comandos básicos CRUD

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

```
hbase> create 'slides', 'slide', 'image', '  
    text', 'xref'  
0 row(s) in 1.3810 seconds  
=> Hbase::Table - slides
```

- ▶ Se crea la tabla **presentaciones** con cuatro familias de columnas, **slide**, que guardará los datos de la transparencia, **image**, que guarda la imagen de la diapositiva, **text**, que guardará el texto de la diapositiva, y **xref**, que guarda las referencias a otras slides



# Hbase Shell (III)

## Comandos básicos CRUD

- ▶ Inserción de valores. El *shell* es limitado y sólo permite añadir una columna a la vez:

```
hbase(main):006:0> put 'slides', 'jisbd17-000', 'slide:  
    title', 'Portada'  
0 row(s) in 0.1730 seconds  
hbase(main):007:0> put 'slides', 'jisbd17-000', 'slide:  
    notes', 'Notas iniciales'  
0 row(s) in 0.0110 seconds
```

(el formato de especificación es  
**familia:columna**)



# Hbase Shell (IV)

## Comandos básicos CRUD

- ▶ Obtener el documento:

```
hbase> get 'slides', 'jisbd17-000'
COLUMN          CELL
slide:notes     timestamp=1500300013076, value=Notas iniciales
slide:title    timestamp=1500299771997, value=Portada
2 row(s) in 0.1700 seconds
```

- ▶ O toda la tabla:

```
base> scan 'slides'
ROW           COLUMN+CELL
jisbd17-000   column=slide:notes, timestamp=1500300013076, value=
              Notas iniciales
jisbd17-000   column=slide:title, timestamp=1500299771997, value=
              Portada
1 row(s) in 0.0400 seconds
```

# Opciones de creación de tablas

- ▶ **VERSIONS** es por defecto 1, y se pueden especificar todas o un número finito de ellas
- ▶ En el caso de la familia **text**, aplicamos compresión, ya que la familia albergará gran cantidad de texto
- ▶ También aplicamos al parámetro **BLOOMFILTER** el valor **ROW**

# Opciones de creación de tablas (II)

- ▶ **BL0OMFILTER** puede estar deshabilitado (**NONE**), o tomar dos valores:  
**ROW** Permite establecer rápidamente si una fila no existe
  - ▶ Como hemos visto, HBase mantiene una estructura de árbol B+ para encontrar las filas buscadas
  - ▶ Sin embargo, el árbol guarda rangos de claves, pero no si una clave específica existe o no
  - ▶ Poniendo el valor **ROW**, se crea un filtro Bloom que permite descartar cuando una clave no está

## Opciones de creación de tablas (III)

**ROWCOL** Realiza también un filtro Bloom para describir si una columna está en una fila concreta o no. Las columnas pueden ser muchas. Además, se utiliza un fichero para cada familia de columnas. Por lo tanto, si no se tiene este índice, se tienen que hacer muchas comprobaciones para saber si una de las columnas a mostrar está en una fila concreta o no. El filtro Bloom con valor **ROWCOL** crea un filtro Bloom para las columnas de cada fila

# happybase

- ▶ Hay varios paquetes para acceder a HBase
- ▶ Incluso podríamos haberlo implementado desde JRuby y el *shell* de HBase
- ▶ Pero lo haremos desde Python ya que lo hemos usado en todo el curso, usando la librería **happybase**<sup>5</sup>
- ▶ Hace uso del protocolo remoto Thrift
- ▶ Instalada en la máquina virtual. Si no, para instalarla:

```
$ sudo pip2 install happybase
```

- ▶ El API es sencillo, ya que HBase ofrece relativamente pocas operaciones

---

<sup>5</sup><https://happybase.readthedocs.io/en/latest/>

# Creación de la base de datos

```
try:  
    connection.create_table(  
        "jisbd17",  
        {  
            'slide': dict(bloom_filter_type='ROW'  
                           ,max_versions=1),  
            'image' : dict(compression='GZ',  
                           max_versions=1),  
            'text' : dict(compression='GZ',  
                           max_versions=1),  
            'xref' : dict(bloom_filter_type='  
                           ROWCOL',max_versions=1)  
        })  
except:  
    print ("Database jisbd17 already exists.")  
    pass
```

# Creación de la base de datos (II)

Podemos copiar la tabla **jisbd17** de Mongo  
(no se copia **xref** porque se hará una  
optimización posterior):

```
h_jisbd17 = hbasecon.table('jisbd17')
with h_jisbd17.batch(batch_size=100) as b:
    for doc in jisbd17.find():
        b.put(doc['_id'], {
            'slide:title' : doc.get('title', ''),
            'slide:notes' : doc.get('notes', ''),
            'text:' : doc.get('text', ''),
            'image:' : str(doc.get('image', '')))
    })
```

# Creación de la base de datos (III)

Con **xref** podemos usar una optimización en HBase:

- ▶ Las filas pueden crecer tanto como se quiera también en columnas
- ▶ El filtro *Bloom ROWCOL* hace muy eficiente buscar por una columna en particular

**IDEA:** Usar los elementos del array como nombres de las columnas. Convierte automáticamente a esa columna en un **índice inverso**:



# Creación de la base de datos (IV)

```
with h_jisbd17.batch(batch_size=100) as b:  
    for doc in jisbd17.find():  
        if 'xref' in doc:  
            for ref in doc['xref']:  
                b.put(doc['_id'], {  
                    'xref:' + ref : ''  
                })
```

Y la búsqueda de índice inverso: Las fila que contienen la columna que buscamos (filas que apuntan a la buscada):

```
list(h_jisbd17.scan(columns=['xref:jisbd17-002']))  
=> [('jisbd17-000', {'xref:jisbd17-002': ''})]
```

# Creación de la base de datos (V)

Finalmente, en HBase, un **scan** es una *pérdida de tiempo* ⇒ Se debería precomputar la referencia inversa e incluirla en cada *slide*. La búsqueda así es  $O(1)$



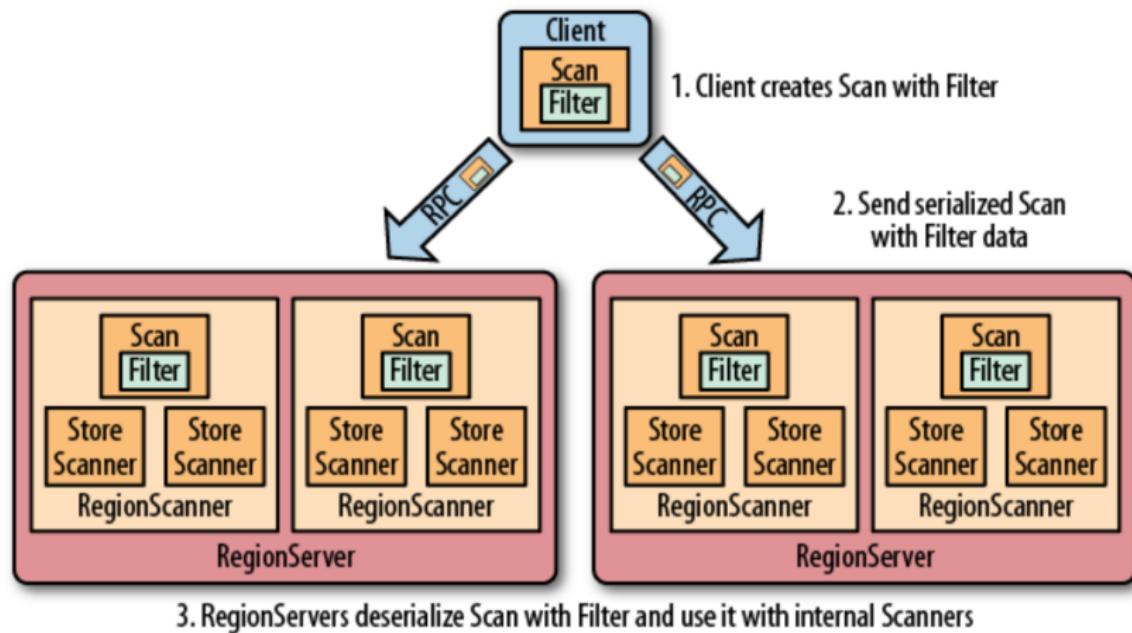
# Búsquedas y filtrado

- ▶ 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 ⇒ lenguaje de filtrado ⇒ el servidor filtre los resultados
- ▶ Clientes Thrift remotos y desde el *shell*
- ▶ Filtrado de cada región en paralelo
- ▶ Escalabilidad horizontal con varios **RegionServers**
- ▶ 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



# Especificación de filtros

- ▶ Los filtros se especifican como parte de la llamada **scan**
- ▶ Hay dos niveles de filtrado:
  - ▶ El referido a columnas, que se especifica con el parámetro **COLUMNS** en el shell y **columns=** en **happybase**:

```
scan 'tabla', {COLUMNS => ['c1', 'c2']}
```

- ▶ Los filtros completos:

```
scan 'tabla' , { FILTER => "Filtro" }
```

(y se pueden combinar ambos)



# Especificación de filtros (II)

- ▶ En las bibliotecas cliente como `happybase` con un parámetro de la función `scan`:

```
table.scan(filter="Filtro...")
```

# Especificación de filtros (III)

- ▶ Ordenación de las consultas en cuanto a su velocidad de ejecución:
  1. Consultas de orden constante, como `get` con clave o un conjunto de columnas
  2. Consultas rápidas porque hacen uso de los filtros Bloom: Prefijos de claves, familias de columnas, prefijos de familias de columnas, ...
  3. Consultas más lentas, que requieren en algunos casos el recorrido de toda la tabla (a evitar): Búsqueda por un valor en concreto, búsqueda de expresiones regulares, etc.  
(Todavía se benefician del escaneo en paralelo de todas las regiones en RegionServers)



# Sintaxis de filtros

- ▶ La sintaxis general de los filtros es:  
**Filtro (parámetro1, parámetro2,...)**
- ▶ Los filtros se pueden unir con expresiones complejas que incluyan paréntesis y también **AND**, **OR**, **SKIP** y **WHILE**
  - ▶ **SKIP** especifica que se ignore la fila si falla el filtro alguno de los pares columna/valor
  - ▶ **WHILE** muestra todos los conjuntos columna/valor de cada fila hasta que un conjunto no cumple el filtro
- ▶ Se ofrecen también operaciones de comparación: **<**, **<=**, **=**, **!=**, **>=**, etc.

# Sintaxis de filtros (II)

- ▶ Y un conjunto de “comparadores”:
  - ▶ *BinaryComparator* – Comparador binario lexicográfico de elementos. Se representa con “**binary**”
  - ▶ *BinaryPrefixComparator* – Comparador binario de prefijos. Se representa con “**binaryprefix**”
  - ▶ *RegexStringComparator* – Comparador de cadenas usando una expresión regular: “**regexstring**”
  - ▶ *SubStringComparator* – Comparador de subcadena: “**substring**”



# Sintaxis de filtros (III)

- ▶ El formato de especificación es:

**comparador:valor**

Por ejemplo:

- ▶ **binary:abc**
- ▶ **binaryprefix:abc**
- ▶ **regexstring:ab\*c\***
- ▶ **substring:def**

# Filtros

- ▶ **KeyOnlyFilter** – No acepta argumentos y retorna sólo las claves de los pares clave/valor:

```
hbase> scan 'jisbd17', {FILTER => "KeyOnlyFilter()"}  
...  
    jisbd17-154          column=image:, timestamp=1500328162013, value=  
    jisbd17-154          column=slide:notes, timestamp=1500328162013,  
        value=  
    jisbd17-154          column=slide:title, timestamp=1500328162013,  
        value=  
    jisbd17-154          column=text:, timestamp=1500328162013, value=  
    jisbd17-155          column=image:, timestamp=1500328162013, value=  
    jisbd17-155          column=slide:notes, timestamp=1500328162013,  
        value=  
    jisbd17-155          column=slide:title, timestamp=1500328162013,  
        value=  
    jisbd17-155          column=text:, timestamp=1500328162013, value=  
319 row(s) in 0.5430 seconds
```

- ▶ **FirstKeyOnlyFilter** – Retorna sólo la primera clave de cada fila

# Filtros (II)

- ▶ **PrefixFilter** – Prefijo de fila dado:

```
hbase> scan 'jisbd17', {FILTER => "PrefixFilter('jisbd17-0')"}  
...  
jisbd17-099      column=slide:notes, timestamp=1500328162137, value=  
jisbd17-099      column=slide:title, timestamp=1500328162137, value=  
                  ={\x5Ctt  
                    happybase  
jisbd17-099      column=text:, timestamp=1500328162137, value= \  
                  x5Cbegin{i  
                    temize}\x0A\x5Citem Hay varios paquetes para  
                    acceder a HBa  
100 row(s) in 8.7800 seconds
```

- ▶ **ColumnPrefixFilter** – Prefijo de columna dado:

```
hbase> scan 'jisbd17', {FILTER => "ColumnPrefixFilter('t')"}  
...  
jisbd17-159      column=slide:title, timestamp=1500328162182, value=  
jisbd17-160      column=slide:title, timestamp=1500328162182, value=  
jisbd17-161      column=slide:title, timestamp=1500328162182, value=  
jisbd17-162      column=slide:title, timestamp=1500328162182, value=
```

## Filtros (III)

- ▶ **MultipleColumnPrefixFilter** – (lista)
- ▶ **ColumnCountGetFilter** – Retorna hasta el número  $n$  de columnas dado
- ▶ **PageFilter** – Permite paginación de resultados por clave de fila

# Filtros (IV)

- ▶ **RowFilter** – Coge un operador de comparación y un comparador. Si la fila encaja con la comparación, la fila entera si muestra:

```
hbase> scan 'jisbd17', {FILTER => "RowFilter(<,'binary:jisbd17-1')"}  
...  
jisbd17-099          column=slide:notes, timestamp=1500328162137, value=  
jisbd17-099          column=slide:title, timestamp=1500328162137, value  
    ={\\x5Ctt  
                    happybase  
jisbd17-099          column=text:, timestamp=1500328162137, value= \  
    x5Cbegin{i  
                    temize}\\xA\\x5Citem Hay varios paquetes para  
                    acceder a HBa  
100 row(s) in 8.7800 seconds
```

## Filtros (V)

- ▶ **FamilyFilter** – Acepta un operador y un comparador. Muestra todas las columnas de familias de las columnas que cumplen el filtro. Puede ser rápido si se ha definido un filtro Bloom **ROWCOL** para las familias de columnas
- ▶ **QualifierFilter** – Acepta un operador y un comparador. Muestra todas las columnas dentro de cualquier familia de columnas que cumplen el filtro. De nuevo, con filtros **ROWCOL** funciona mejor porque puede eliminar muchas filas



# Filtros (VI)

- ▶ **ValueFilter** – Acepta un operador y un comparador. Si algún valor de algún par clave/valor coincide, se muestran todos los elementos de la fila

# Filtros (VII)

- ▶ **SingleColumnValueFilter** – Acepta una familia de columnas, un calificador de columna, un operador de comparación y un comparador. Si la columna no está o está y tiene el valor especificado, se lista toda la fila

```
hbase> scan 'jisbd17', {FILTER => "SingleColumnValueFilter('slide', 'title', '=', 'binary:HBase')"}  
jisbd17-085      column=slide:notes, timestamp=1500328162109, value=  
jisbd17-085      column=slide:title, timestamp=1500328162109, value=  
                  HBase  
jisbd17-085      column=text:, timestamp=1500328162109, value= \\\n                  x5Cbegin{b  
                                lock}{}\x0A  {\x5CLarge  HBase is a {\x5Ccolor{  
                                red} spar  
...  
1 row(s) in 0.3610 seconds
```

## Filtros (VIII)

- ▶ **ColumnRangeFilter** – Acepta dos valores de columna y dos booleanos. Muestra las columnas que están entre los valores. Los booleanos especifican si se incluyen los valores límite o no.

# Resumen de filtros y características

Filter	Batch <sup>a</sup>	Skip <sup>b</sup>	While-Match <sup>c</sup>	List <sup>d</sup>	Early Out <sup>e</sup>	Gets <sup>f</sup>	Scans <sup>g</sup>
RowFilter	✓	✓	✓	✓	✓	✗	✓
FamilyFilter	✓	✓	✓	✓	✗	✓	✓
QualifierFilter	✓	✓	✓	✓	✗	✓	✓
ValueFilter	✓	✓	✓	✓	✗	✓	✓
DependentColumnFilter	✗	✓	✓	✓	✗	✓	✓
SingleColumnValueFilter	✓	✓	✓	✓	✗	✗	✓
SingleColumnValueExcludeFilter	✓	✓	✓	✓	✗	✗	✓
PrefixFilter	✓	✗	✓	✓	✓	✗	✓
PageFilter	✓	✗	✓	✓	✓	✗	✓
KeyOnlyFilter	✓	✓	✓	✓	✗	✓	✓
FirstKeyOnlyFilter	✓	✓	✓	✓	✗	✓	✓

# Resumen de filtros y características

Filter	Batch <sup>a</sup>	Skip <sup>b</sup>	While-Match <sup>c</sup>	List <sup>d</sup>	Early Out <sup>e</sup>	Gets <sup>f</sup>	Scans <sup>g</sup>
FirstKeyValueMatchingQualifiersFilter	✓	✓	✓	✓	✗	✓	✓
InclusiveStopFilter	✓	✗	✓	✓	✓	✗	✓
FuzzyRowFilter	✓	✓	✓	✓	✓	✗	✓
ColumnCountGetFilter	✓	✓	✓	✓	✗	✓	✗
ColumnPaginationFilter	✓	✓	✓	✓	✗	✓	✓
ColumnPrefixFilter	✓	✓	✓	✓	✗	✓	✓
MultipleColumnPrefixFilter	✓	✓	✓	✓	✗	✓	✓
ColumnRange	✓	✓	✓	✓	✗	✓	✓
TimestampsFilter	✓	✓	✓	✓	✗	✓	✓
RandomRowFilter	✓	✓	✓	✓	✗	✗	✓
SkipFilter	✓	✓/✗ <sup>a</sup>	✓/✗ <sup>h</sup>	✓	✗	✗	✓
WhileMatchFilter	✓	✓/✗ <sup>h</sup>	✓/✗ <sup>h</sup>	✓	✓	✗	✓
FilterList	✓/✗ <sup>h</sup>	✓/✗ <sup>h</sup>	✓/✗ <sup>h</sup>	✓	✓/✗ <sup>h</sup>	✓	✓

# Resumen de filtros y características

- <sup>a</sup> Filter supports Scan.setBatch(), i.e., the scanner batch mode.
- <sup>b</sup> Filter can be used with the decorating SkipFilter class.
- <sup>c</sup> Filter can be used with the decorating WhileMatchFilter class.
- <sup>d</sup> Filter can be used with the combining FilterList class.
- <sup>e</sup> Filter has optimizations to stop a scan early, once there are no more matching rows ahead.
- <sup>f</sup> Filter can be usefully applied to Get instances.
- <sup>g</sup> Filter can be usefully applied to Scan instances.
- <sup>h</sup> Depends on the included filters.

# Filtros vs. get

- ▶ Por muy eficiente que se haga el filtrado, un **scan** siempre va a ser mucho más lento que obtener una fila con **get**
- ▶ P.ej (3580ms):

```
hbase> scan 'wlinks', { COLUMNS => ['to:19']
    }
...
Yilmar Mosquera      column=to:19,
    timestamp=1479230326585, value=
Yukiko Iwai          column=to:19,
    timestamp=1479230347810, value=
113 row(s) in 35.7690 seconds
```

# Filtros vs. get (II)

- ▶ vs (18ms):

```
hbase> get 'wlinks', '19', { COLUMNS => [ 'from' ] }
```

...

```
from:Yilmar Mosquera timestamp  
=1479230326578, value=  
from:Yukiko Iwai timestamp  
=1479230347796, value=  
113 row(s) in 0.1760 seconds
```

- ▶ También existe una especificación de fila de inicio y final que son más eficientes que un filtro *Row Prefix*:

```
hbase> scan 'wlinks', {STARTROW => '19',  
ENDROW => '20', COLUMNS => [ 'from' ] }
```

# Filtros vs. get (III)

Y en happybase:

```
table.scan(row_start='19', row_stop='20',  
          columns=['from'])
```

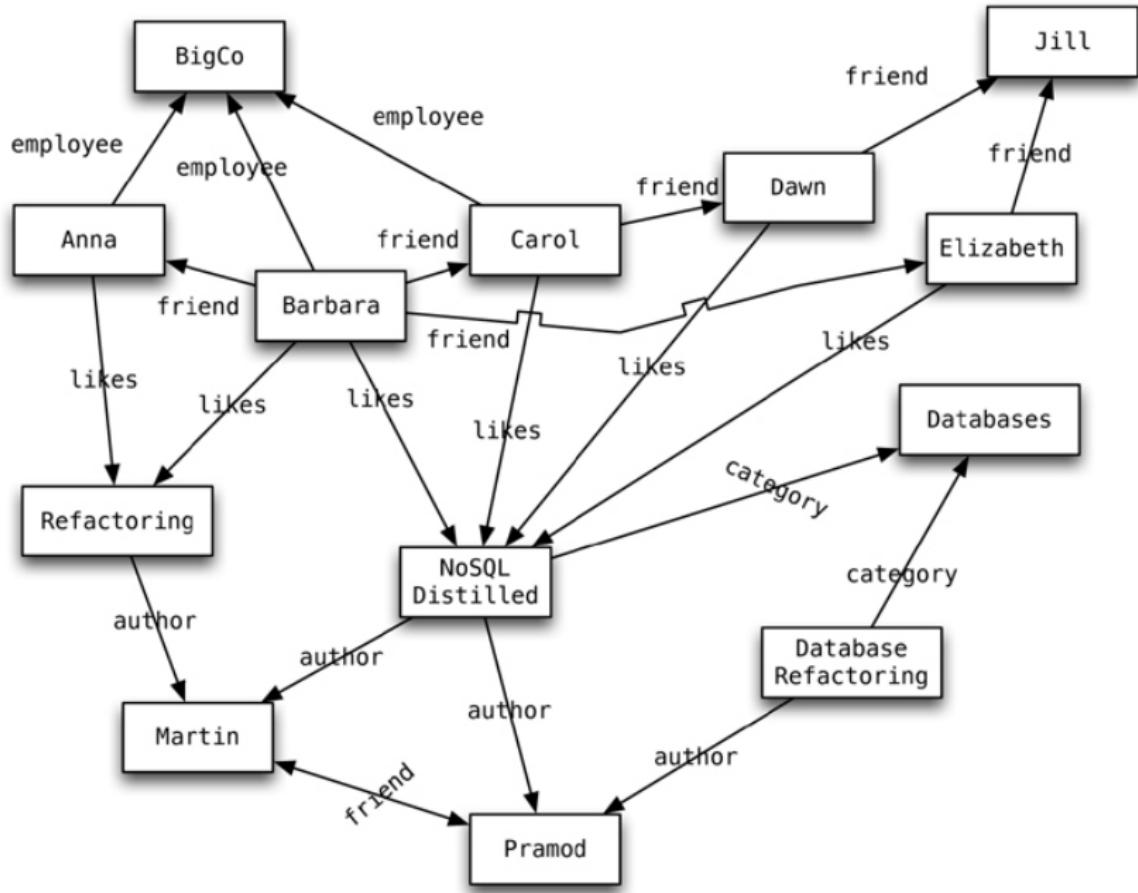
# Bases de Datos de Grafos

- ▶ 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





(Nota: Usa la sintaxis PostgreSQL para json)

```
CREATE TABLE vertices (
    vertex_id integer PRIMARY KEY,
    properties json
);

CREATE TABLE edges (
    edge_id integer PRIMARY KEY,
    tail_vertex integer REFERENCES vertices (
        vertex_id),
    head_vertex integer REFERENCES vertices (
        vertex_id),
    label text,
    properties json
);

CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

# 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:

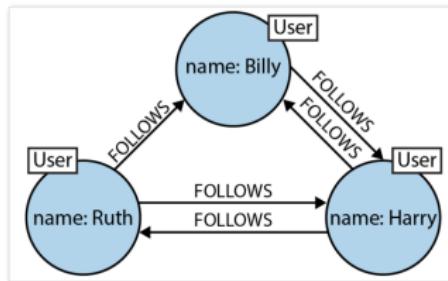
```
MATCH
  (person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us:
    Location {name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:
    Location {name:'Europe'})
RETURN person.name
```

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



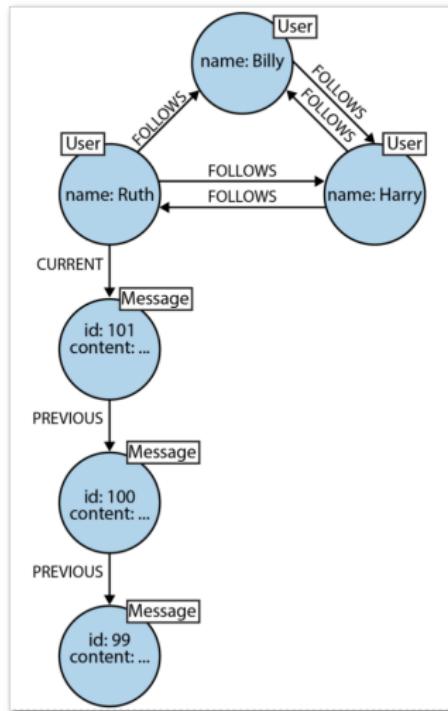
# Aplicabilidad de Grafos

- ▶ 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:

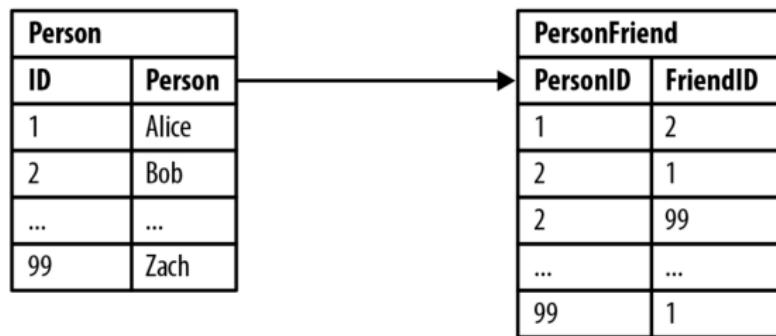


# Flexibilidad y eficiencia

- ▶ 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
- ▶ Como ejemplo, usemos una relación de amistad (*friend*) entre usuarios de una red social

# Flexibilidad y eficiencia (II)

- ▶ Una posible implementación relacional podría ser:



## Flexibilidad y eficiencia (III)

- ▶ Una consulta sencilla para obtener los amigos de Bob:

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
ON PersonFriend.FriendID = p1.ID
JOIN Person p2
ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

- ▶ Es sencilla y no es computacionalmente muy compleja
- ▶ Como la relación de amistad no es recíproca siempre, a veces hay que hacer la búsqueda inversa:

## Flexibilidad y eficiencia (IV)

```
SELECT p1.Person  
FROM Person p1 JOIN PersonFriend  
ON PersonFriend.PersonID = p1.ID  
JOIN Person p2  
ON PersonFriend.FriendID = p2.ID  
WHERE p2.Person = 'Bob'
```

- ▶ Esta consulta es más costosa que la anterior, porque se tienen que recorrer todas las filas de **PersonFriend**
- ▶ Pero ¿y si queremos “**los amigos de los amigos de Alice**”? Ahora la consulta es mucho más compleja computacionalmente y también más difícil de expresar en SQL:

# Flexibilidad y eficiencia (V)

```
SELECT p1.Person AS PERSON, p2.Person AS
    FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
ON pf2.PersonID = pf1.FriendID
JOIN Person p2
ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID
    <> p1.ID
```

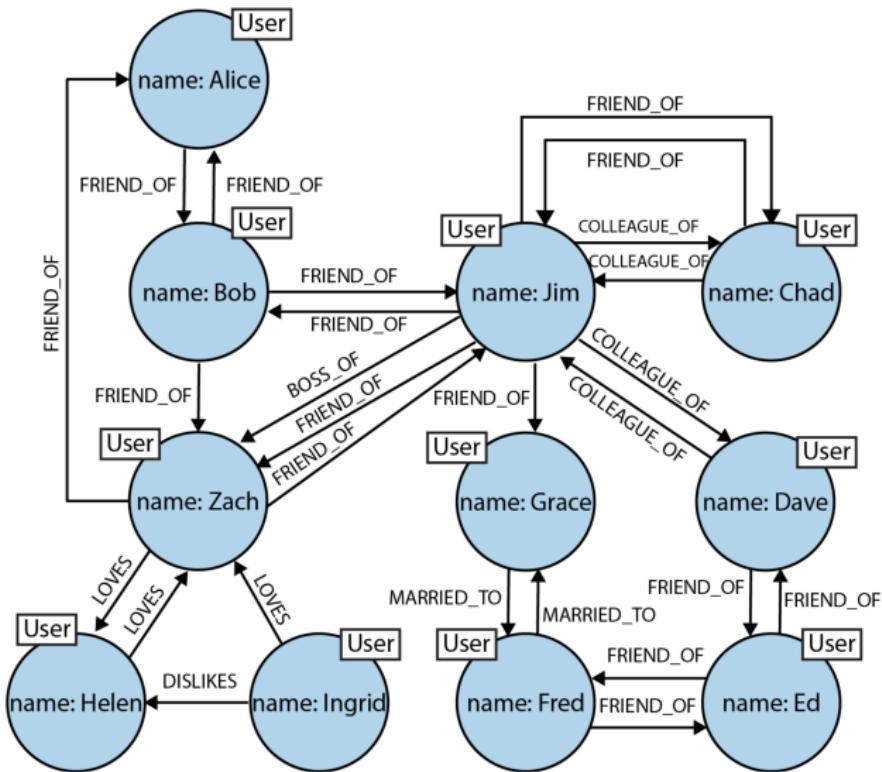
- ▶ Y sólo estamos bajando un nivel (amigos de amigos de Alice)
- ▶ Si tuviéramos que bajar otro nivel, se haría mucho más complejo por todos los **JOIN**

## Flexibilidad y eficiencia (VI)

- ▶ En *Neo4j in Action*, Partner y Vukotic hicieron un experimento con Neo4j y MySQL con esta tabla
- ▶ Buscaron amigos de amigos en diferentes niveles de profundidad
- ▶ En una red de 1 millón de personas cada uno con unos 50 amigos

Prof.	Tiempos RDBMS	Tiempos Neo4j	Resultados
2	0,016	0,01	~2500
3	30,267	0,168	~110.000
4	1543,505	1,359	~600.000
5	(no termina)	2,132	~800.000

# Flexibilidad y eficiencia (VII)



# 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)
```

# 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:Post) RETURN n
```

(retorna todos los *posts*)

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

# Índices

- ▶ Se pueden crear índices para búsquedas rápidas sobre atributos:

```
CREATE INDEX ON :Etiqueta(nombre)
```

# Bases de Datos basadas en Arrays

- ▶ Suelen presentarse como bases de datos que soportan SQL y añaden operaciones para trabajar con conjuntos de datos especiales (arrays)
- ▶ Son bases de datos muy especializadas y no las trataremos aquí
- ▶ Utilizadas para tratamiento de grandes cantidades de datos de forma estadística o de modelado y OLAP

# Bases de Datos basadas en Arrays (II)

- ▶ Soportan también datos geográficos, ya que pueden definir rangos numéricos de una o varias dimensiones (2D para cálculos geográficos)
- ▶ Ejemplos: MonetDB, SciDB, rasdaman

# Referencias

- ❖ Nathan Marz, James Warren  
*Big Data: Principles and best practices of scalable realtime data systems*  
Manning Publications, 2015
- ❖ Eric Redmond, Jim R. Wilson  
*Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*  
Pragmatic Bookshelf, 2012

# Referencias (II)

- Pramod J. Saldage, Martin Fowler  
*NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*  
Addison-Wesley, 2013
- Jim R. Wilson, Eric Redmond  
*Seven Databases in Seven Weeks*
- Kleppmann  
*Designing Data Intensive Applications*
- Lars George  
*HBase, The Definitive Guide*

# Referencias (III)

- The Apache HBase Team  
*Apache HBase Reference Guide*  
<https://hbase.apache.org/book.html>
- Ian Varley  
Vídeo: *HBase Schema Design*  
<http://www.cloudera.com/content/dam/www/marketing/resources/events/hbase-con/video-hbasecon-2012-hbasecon-2012.png.landing.html>

# Referencias (IV)

Transparencias:

[http://es.slideshare.net/ivarley/  
hbase-schema-design-hbasecon-2012](http://es.slideshare.net/ivarley/hbase-schema-design-hbasecon-2012)



Lars George

*HBase Schema Design*

[https://2013.nosql-matters.org/  
cgn/wp-content/uploads/2013/05/  
HBase-Schema-Design-NoSQL-Matters-App  
pdf](https://2013.nosql-matters.org/cgn/wp-content/uploads/2013/05/HBase-Schema-Design-NoSQL-Matters-Appdf)

Otro vídeo:

<https://vimeo.com/44715954>

Sus transparencias: <http://2012.berlinbuzzwords.de/sites/>

# Referencias (V)

2012.berlinbuzzwords.de/files/  
slides/hbase-lgeorge-bbuzz12.pdf