Descripción breve

MongoDB y Neo4J para base de datos DBLP Computer Science Bibliography, considerada como la mayor recopilación existente de referencias bibliográficasacadémicas específicamente centrada en la informática. Almacena los datos relativos a la gran mayoría de las revistas científicas y congresos académicos sobre informática.

Bases de datos no convencionales

Máster en Data Science|URJC

**Javier Pérez Clemente y Belén González Guío**

07/04/2019



Tabla de contenido

[Tabla de contenido 1](#_Toc5556375)

[**INTRODUCCIÓN** 1](#_Toc5556376)

**[PARTE I : MONGODB](#_Toc5556377)** [2](#_Toc5556377)

[**1 .- CAPTURA Y PROCESAMIENTO DE DATOS** 2](#_Toc5556378)

[**2.- DEFINICION DEL ESQUEMA DE LA BBDD** 2](#_Toc5556379)

[**2.- ALMACENAMIENTO DE DATOS** 4](#_Toc5556380)

[**3.- ANÁLISIS DE DATOS** 5](#_Toc5556381)

[**PARTE II: NEO4J** 11](#_Toc5556382)

[**1 .- CAPTURA Y PROCESAMIENTO DE DATOS** 11](#_Toc5556383)

[**2.- ALMACENAMIENTO DE DATOS** 12](#_Toc5556384)

[**3.- ANÁLISIS DE DATOS** 12](#_Toc5556385)

# **INTRODUCCIÓN**

La fuente de datos que se va a utilizar en esta práctica es una fuente de datos real que recopila **referencias bibliográficas académicas**, concretamente relacionadas con la **informática**. Esta fuente de datos es conocida como **DBPL** *Computer Science Bibliography*. Los datos que vamos a procesar y, más tarde, almacenar para que puedan ser consultados, han sido descargados manualmente del siguiente link: <https://dblp.uni-trier.de/xml/>.

La información relacionada con cada artículo que podemos encontrar en este fichero es título del artículo, autor/autores, páginas, fecha de publicación, etc. Los datos recogen **8 tipos de artículos**, pero en nuestro caso vamos a **trabajar con 3** de ellos: **artículos de revista** (*article*), **artículos en congresos** (*inproceedings*) y **artículos en libros** (*incollection*).

# **PARTE I : MONGODB**

En esta primera parte de la práctica se va a trabajar con la **base de datos no relacional** **MongoDB**, que es de **tipo documental**. Las bases de datos orientadas a documentos se caracterizan porque almacenan los **valores como documentos** (normalmente en formato JSON o XML). De esta manera, las bases de datos no relacionales documentales almacenan todos los atributos de una entidad en un único documento, en vez de almacenar cada atributo de la entidad con una clave separada. Además, existe una gran **flexibilidad en la estructura de los documentos**, a diferencia de las bases de datos relacionales.

## **1 .- CAPTURA Y PROCESAMIENTO DE DATOS**

Como se mencionó antes, la captura de los datos se ha hecho descargando manualmente del link de la página web de dpbl. Se han descargado **dos archivos**:

* un zip que contiene las **publicaciones** en formato **XML**
* un **fichero** de tipo **dtd**

El archivo dtd contiene la definición del tipo de documento y describe el vocabulario que identifica los elementos y las entidades de los que consta el XML. Es decir, este archivo auxiliar expresa la estructura y es necesario porque permite que el programa de Python que parsea los documentos (que explicaremos más tarde) sea mucho más rápido. El fichero XML comprimido ocupa alrededor de 450 MB y descomprimido pesa acerca de **2 GB**.

## **2.- DEFINICION DEL ESQUEMA DE LA BBDD**

Antes de procesar todos los datos, analizamos cuales eran las consultas más frecuentes para organizar el esquema en función de estas consultas. De esta manera evitamos procesar datos o atributos que no sean necesarios y que no se vayan a usar.

Lo primero ha sido **especificar** qué ***tags*** queremos procesar y cuáles no son de nuestro interés puesto que, como se comentó, aunque hay 8 tipos de artículos, solo estamos interesados en 3: ***article****,* ***inproceedings*** *e* ***incollection***. Además, antes de procesar los datos, se han analizado las consultas para evitar procesar atributos que no se van a usar o que no van a ser consultados y para poder definir un esquema. Se han seleccionado finalmente los siguientes 5 **atributos**: **‘id’, ‘type’, ‘date’, ‘title’** y **‘authors’.**

Una vez que hemos seleccionado los campos que vamos a almacenar, se ha definido un esquema mirando las consultas que se tienen que realizar para poder entender cómo se usarán los datos. Si bien una base de datos documental es flexible y no exige que la estructura de todos los documentos sea idéntica, es importante definir un esquema porque la estructura de los documentos, aunque no sea idéntica, tienen que tener una estructura común. Para ello hay que definir cuantas colecciones vamos a definir en esta base de datos de MongoDB. Normalmente, se eligen el número de colecciones en función de cuántos tipos de documentos difieren en su estructura. En nuestro caso, como tenemos tres posibles tipos de documentos (*articles, inproceedings, incollection*) pero tienen una estructura muy similar, hemos decidido crear **una sola colección**. A esta colección la hemos llamado ***publications*** y contiene 5 campos:

* **\_id**: Identificador único auto incremental para cada publicacion presente la colección.
* **type**: Tipo de publicación. Puede tomar 3 valores: *article*, *inproceedings* e *incollections*.
* **date**: Fecha de publicación en formato %Y-%m-%d
* **authors**: A*rray* que recoge los autores que han participado en la publicación.
* **title**: título de la publicación

A continuación, se muestra un ejemplo de dos documentos insertados en la base de datos para visualizar la estructura que tienen los jsons:

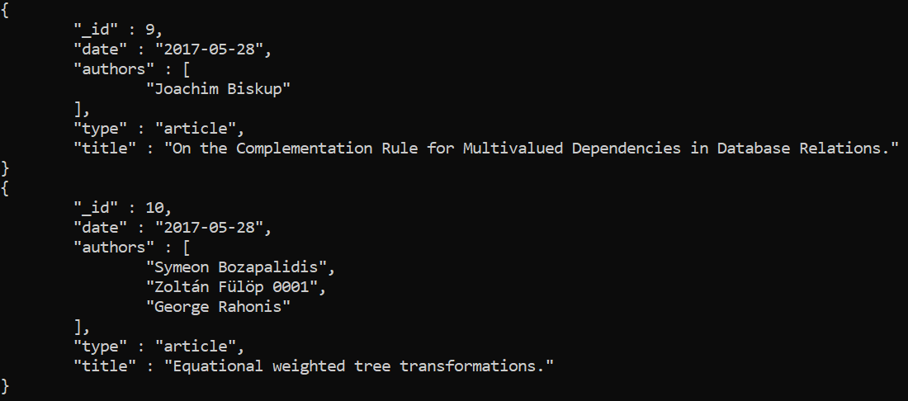


Ilustración 1 Ejemplo documentos insertados en MongoDB

El archivo que parsea el XML e inserta en una base de datos de MongoDB se llama **xml\_to\_mongo.py**, y utiliza para ello dos librerías: **etree** y **MongoClient**. Este código consta de **dos funciones**:

* ***get\_dict***: recibe el elemento del á
* rbol XML y el identificar auto incremental y construye un diccionario con los atributos anteriormente definidos. para cada publicación que acaba formando una lista de diccionarios, donde cada diccionario como se comentaba es una publicación o documento (en términos de la base de datos)
* ***write***: conecta con la el cliente de Mongo y con la base de datos y e inserta cada 500000 publicaciones en la base de datos.

Estas dos funciones se llaman dentro de un bucle que recorre todos los registros del XML. Al ser un archivo tan pesado, no se ha podido cargar directamente con Python. Por lo tanto, para procesar tantos datos se ha hecho uso de ***iterparse***, que construye el árbol, pero te permite eliminar partes del árbol que ya no se necesiten para dejar hueco y procesar el siguiente trozo. En otras palabras, el XML se procesa por trozos para no ir acumulando todos los datos y poder liberar espacio. En el *iterparse*, se ha especificado los *tags* que se quieren procesar (*article*, *inproceedings* e *incollection*).

La primera estrategia probada consistia en utilizar la funcion parse() librería xmltodict para convertir el fichero xml a diccinario. Sin embargo, esto resultó inviable dado el tamaño del mismo, asi que tuvimos que pensar en opciones alternativas. Finalmente, utilizamos la funcion iterparse de la clase etree (librería lxml), que permite procesar el xml elemento a elemento ayudandose del archivo “dblp.dtd”. Para cada elemento, llamamos a la función get\_dict para obtener el correspondiente diccionario y lo almacenamos en la lista “result”.

## **2.- ALMACENAMIENTO DE DATOS**

Una vez contamos con todas las publicaciones recogidas en la lista “*result*”, llamamos a la función *write()* para escribir los documentos en la colección “publications” de nuestra base de datos mongo.

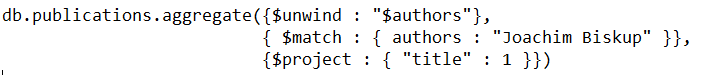
Debido al elevado número de publicaciones, nos vimos obligados a realizar escrituras parciales cada vez que la longitud de la lista superaba un límite establecido en 500.000 elementos.

El proceso completo de parseo y almacenamiento tarda en ejecutarse en torno a 3 minutos.

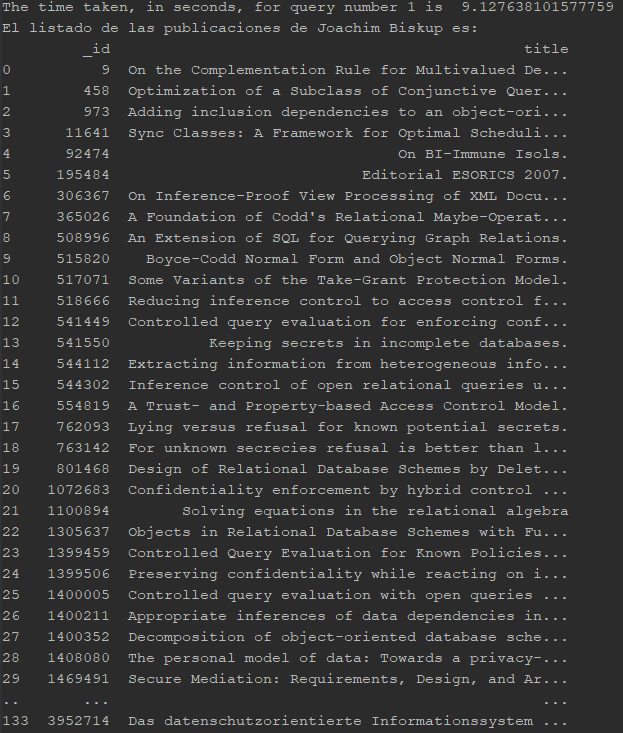
## **3.- ANÁLISIS DE DATOS**

Una vez que ya se han procesado todos los datos y se han almacenado en Mongo, la base de datos ya está lista para ser consultada. Se han realizado las 10 consultas propuestas y se han recogido en un archivo Python que conecta con la base de datos, realiza las consultas y recibe los resultados de éstas. Este archivo se llama **queries\_mongodb.py**. A parte también muestra el tiempo que tarda la *query*. Este tiempo se va a comparar antes de insertar índices en la colección y después, para ver cómo realmente los índices aceleran las consultas.

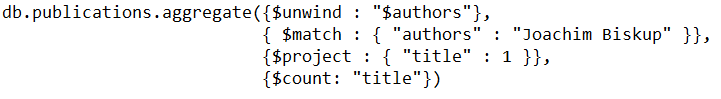
1. Listado de publicaciones de un autor determinado. En nuestro caso, el autor es *Joachim Biskup*.



Con esta consulta especificamos con *unwind* que queremos que convierta cada documento con *array* en tantos documentos como elementos tenga el *array*. Después filtramos los registros cuyo autor es *Joachim Biskup* y pedimos que nos devuelta el título.



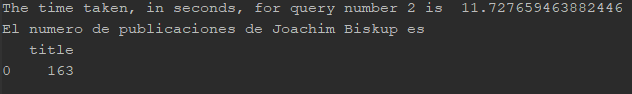
1. Número de publicaciones de un autor determinado. En nuestro caso es, una vez más, el autor *Joachim Biskup*.



Esta consulta, solo en este caso, es equivalente a la siguiente:



La razón es que ambas consultas nos dan el mismo número de publicaciones (probando con y sin *unwind*) porque da la casualidad de que el autor escogido tiene publicaciones donde solo él es el autor y, por lo tanto, no necesita hacer *unwind* para sacar el total de publicaciones que tiene. En casos en los que un autor colabore con otros, sí que es necesario usar *unwind* porque si no estaríamos cogiendo solo las publicaciones donde él es el autor único.



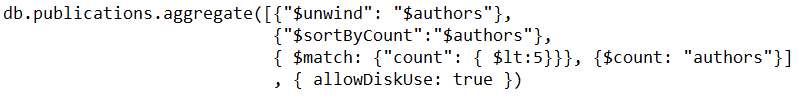
1. Número de artículos en revista para el año 2017



En este caso no hace falta hacer uso de *aggregate*, sino que vale con *find* puesto que ponemos las dos condiciones en un único elemento del pipeline. Se usa *regex* para *matchear* una expresión regular, aunque en este caso no es una expresión regular como tal, sino que es un substring del campo date.

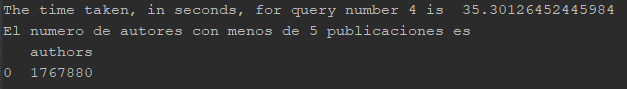
cid:image001.png@01D4EBB0.F5D00850

1. Número de autores ocasionales, es decir, que tengan menos de 5 publicaciones en total.

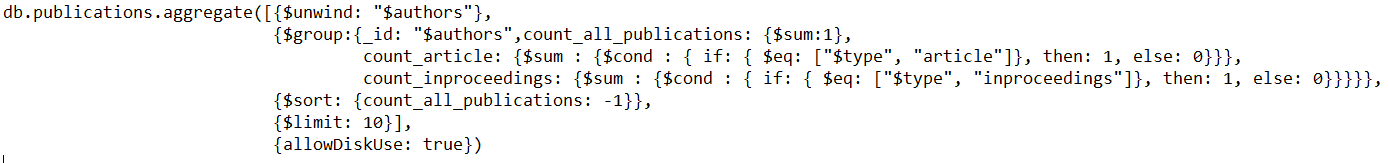


En este caso se usa *sortByCount* que ordena y cuenta el campo que le indiques.

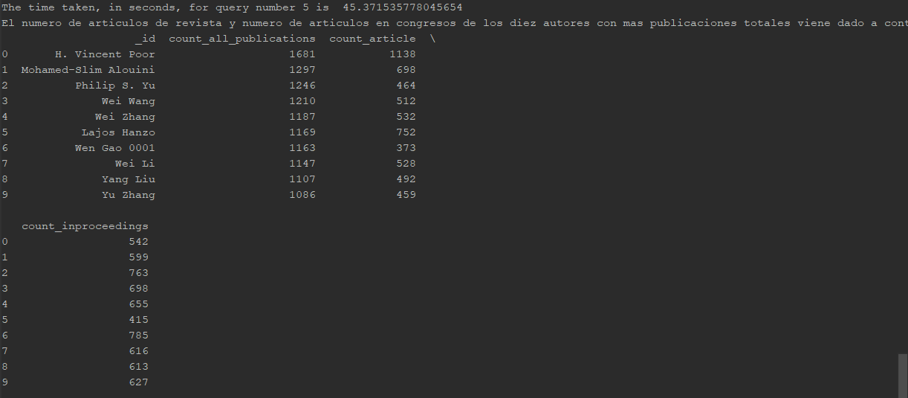
Cabe mencionar que hemos necesitado añadir la variable *allowDiskUse: true* específicamente para que no se quede sin memoria al hacer la consulta. Esto nos pasará más adelante con otras consultas.



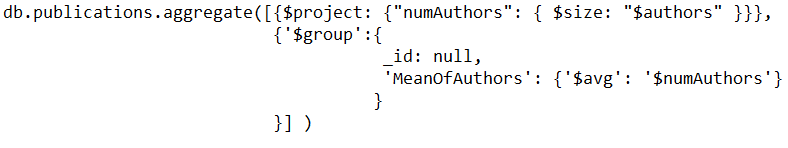
1. Número de artículos de revista (article) y número de artículos en congresos (inproceedings) de los diez autores con más publicaciones totales.



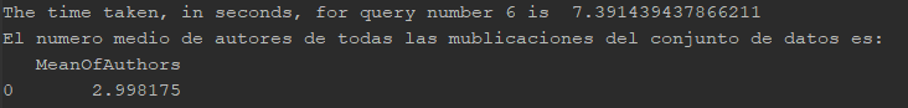
En este caso no hemos podido usar *sortByCount* como en la anterior consulta porque este operador solo te devuelve dos columnas y después quieres hacer más operaciones no te lo permite. Por ello, se ha usado *group* y luego *sort*.



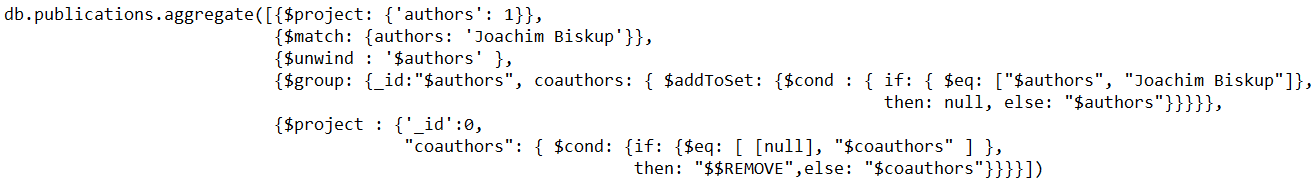
1. Número medio de autores de todas las publicaciones que tenga en su conjunto de datos.



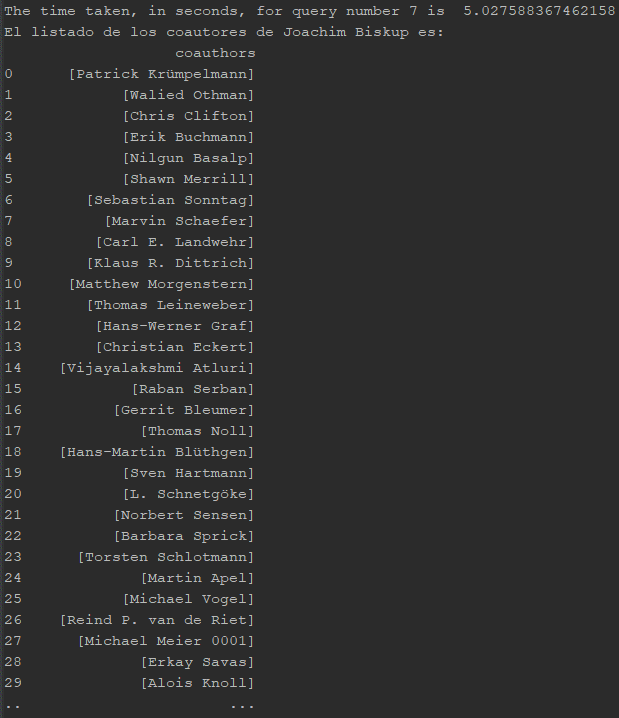
Para responder esta consulta, se ha creado una variable que calcule el tamaño del array que contiene al autor o autores para cada publicación y después se ha calculado la media, obteniendo así que, en promedio, las publicaciones tienen casi 3 autores.



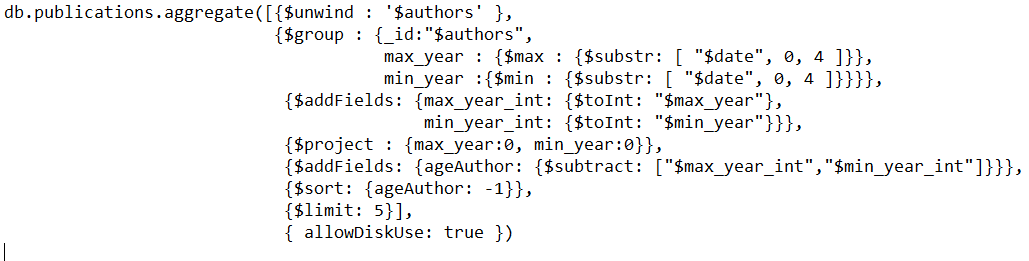
1. Listado de coautores de un autor (se denomina coautor a cualquier persona que haya firmado una publicación). En nuestro caso es, una vez más, el autor *Joachim Biskup*.



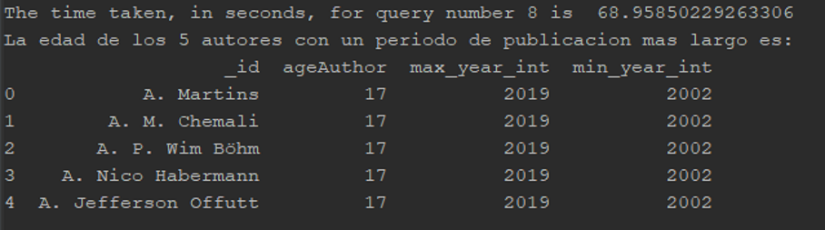
Aquí filtramos los documentos por aquellos donde aparece *Joachim Biskup,* y después se hace *unwind* y devolvemos aquellos que no son *Joachim Biskup*.



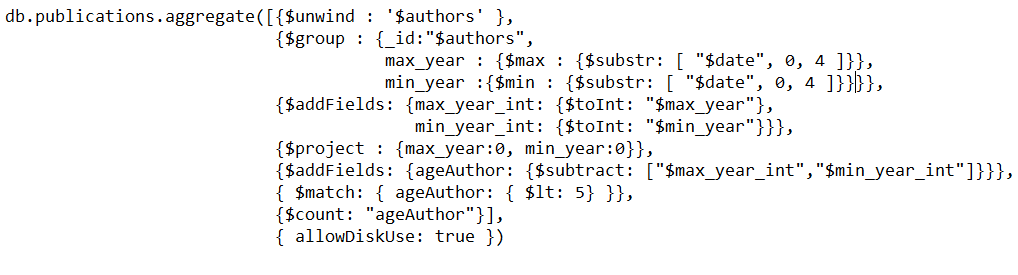
1. Edad de los 5 autores con un periodo de publicación más largo (se considera la edad de un autor al número de años transcurridos desde la fecha de su primera publicación hasta la última registrada)



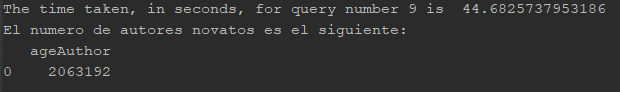
Agrupamos por autor y construimos dos variables (una que hace referencia al año máximo del autor y otra al año mínimo del autor) utilizando *substr* para extraer del campo *date* los 4 primeros dígitos como *string*. Después, creamos otras dos variables que son resultado de convertir las dos anteriores de *string* a *integer*. Por último, se calcula la diferencia de estas dos variables (que es lo que consideramos como edad del autor), se ordenan de manera decreciente y limitamos a 5 los resultados.



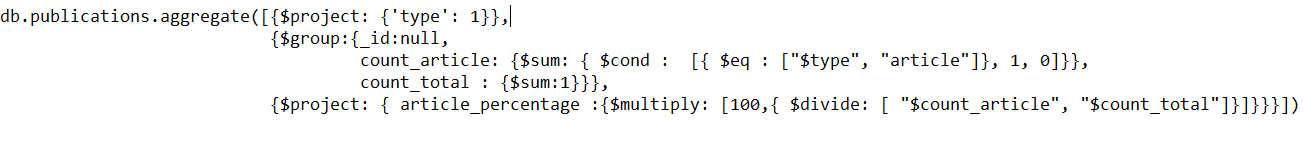
1. Número de autores novatos, es decir, que tengan una edad menor de 5 años. Se considera la edad de un autor al número de años transcurridos desde la fecha de su primera publicación hasta la última registrada.



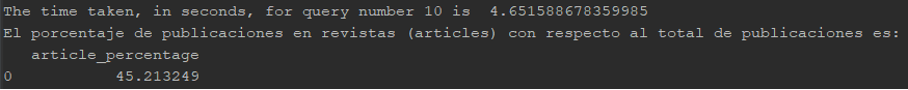
Calculamos la edad de los autores de la misma manera que hicimos en la anterior consulta y después filtramos para quedarnos con aquellos registros cuya edad de autores es menor que 5 y hacemos un *count*.



1. Porcentaje de publicaciones en revistas con respecto al total de publicaciones.



Para obtener el porcentaje de publicaciones en revistas (*articles*) con respecto al total de publicaciones, se ha construido una variable llamada *count\_article* que es binaria. Toma valor 1 cuando el *type* es igual a *article* y el valor 0 cuando no. También construimos otra variable que sume el total de registros. Por último dividimos el número de publicaciones en revistas entre el total y mulitplicamos por 100.



Después de hacer estas consultas, se han insertado una serie de **índices** para ver si mejora el tiempo de ejecución de estas consultas. La creación de índices se encuentra en el archivo **xml\_to\_mongo.py. Estos índices han sido añadidos a las consultas (en el archivo** **queries\_mongodb.py) como *hint()* para acelerar el tiempo de respuesta.**

**Los índices creados son los siguientes:**

* **Índices simple sobre el campo autor**
* **Índice simple sobre el campo fecha**
* **Índice compuesto sobre los campos autor y fecha**

A continuación se muestra una tabla **comparativa** de los **tiempos** de ejecución de las consultas desde Python **con y sin índices**.

|  |  |  |
| --- | --- | --- |
| Question | Execution time without indexes (in seconds) | Execution time with indexes (in seconds) |
| 1 | 9.11 | 8.43 |
| 2 | 11.70 | 11.27 |
| 3 | 2.08 | 20.13 |
| 4 | 31.63 | 31.88 |
| 5 | 39.98 | 40.30 |
| 6 | 8.24 | 7.56 |
| 7 | 2.58 | 2.47 |
| 8 | 37.29 | 37.75 |
| 9 | 37.59 | 39.37 |
| 10 | 4.59 | 4.80 |

# **PARTE II: NEO4J**

Esta segunda parte de la práctica va a utilizar los **mismos datos** pero, a diferencia de la parte I, la base de datos que se va a usar es una **base de datos orientada a grafos (Neo4J)** en vez de una base de datos orientada a documentos (MongoDB). Las bases de datos orientadas a grafos representan la información como **nodos** que se relacionan mediante **aristas**. Cada nodo tiene un identificador y puede tener varios atributos que describen este nodo. Uno de los puntos fuertes de estos tipos de bases de datos es que su rendimiento permanece constante aun cuando crece el conjunto de datos. Además, son bastante flexibles, lo cual se ajusta perfectamente con los entornos empresariales cambiantes que van de la mano de las metodologías ágiles.

Por lo general, se usarán este tipo de grafos cuando queramos modelar relaciones.

## **1 .- CAPTURA Y PROCESAMIENTO DE DATOS**

Para capturar los datos, procesarlos y después insertarlos, inicialmente se planteó la opción de exportar los datos de MongoDB en formato csv. Esta opción se descartó porque el comando de exportación en formato csv de mongo, añadía comillas dentro del array de autores que complicaba el proceso. Además, como el formato en el que se insertan los datos en Neo4J es diferente de Mongo (insertamos por un lado los nodos y por otro lado las relaciones), se construyó un parseador que generase **3 ficheros csv**:

* **2** ficheros con dos **tipos de nodos: autor\_nodes.csv** y **publication\_nodes.csv**
* **1** fichero que contiene las **relaciones** entre estos dos tipos de nodos: **relationships.csv**

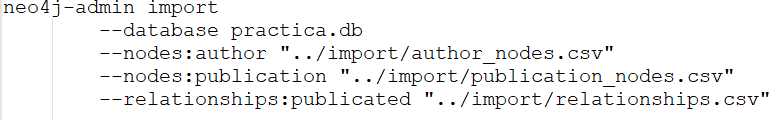
El archivo Python que crea estos tres documentos se llama **csv\_parser.py**. Este código recorre el XML haciendo uso de ***iterparse***, como se hacía en la parte 1.

En total se generan **6.699.232 nodos**. Como comentábamos, un archivo recoge los **nodos de tipo autor** y el otro recoger los **nodos de tipo publicación**, que a su vez tiene tres tipos: *article*, *inproceedings* e *incollections*.

En cuanto a las **relaciones**, se han generado **13.253.583** de **un solo tipo**: **publicated**. (Puede haber varios tipo de relaciones entre nodos, pero no es nuestro caso)

## **2.- ALMACENAMIENTO DE DATOS**

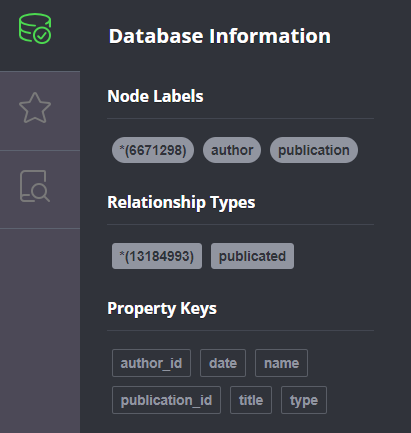
Como los datos ya han sido parseados y se han recogido en 3 archivos en formato CSV, para almacenar los datos en Neo4J se ha ejecutado el siguiente comando en la terminal:



Es importante que los ficheros **csv** estén en el **directorio** de **Neo4J**, en la **carpeta import**.

Cabe mencionar también, que con el fin de cambiar de base de datos que se visualiza en el browser, hay que modificar el archivo **neo4j.conf** y des comentar la línea donde aparece la variable **dbms.active\_database** e igualarla al nombre de la base de datos que queramos.

A continuación se muestra la información de la base de datos creada:



Como vemos, la diferencia entre Mongo y Neo4J reside en la manera de almacenar la información. Esto repercute en las consultas, el esquema, etc. Mientras que MongoDB guarda cada registro como si fuera un documento en formato json, Neo4J lo almacena como nodos que tienen atributos y se relacionan entre ellos.

## **3.- ANÁLISIS DE DATOS**

Las consultas más apropiadas para una base de datos orientada a grafos son aquellas que implican la “navegación” por relaciones. Es decir, las consultas basadas en relaciones.

A continuación se muestran las consultas que se han realizado en Neo4J:

¿Qué autor tiene más colaboraciones?