Gestores NoSQL – Neo4j

Marta Zorrilla – Diego García-Saiz

Enero 2017

Tabla de contenidos

Bibliografía y documentación complementaria

- Libros:

  I. Robinson. **Graph Datasebases**. O'Reilly Media. 2015
  (disponible en: https://neo4j.com/graph-databases-book/?ref=home)

- Tutoriales:

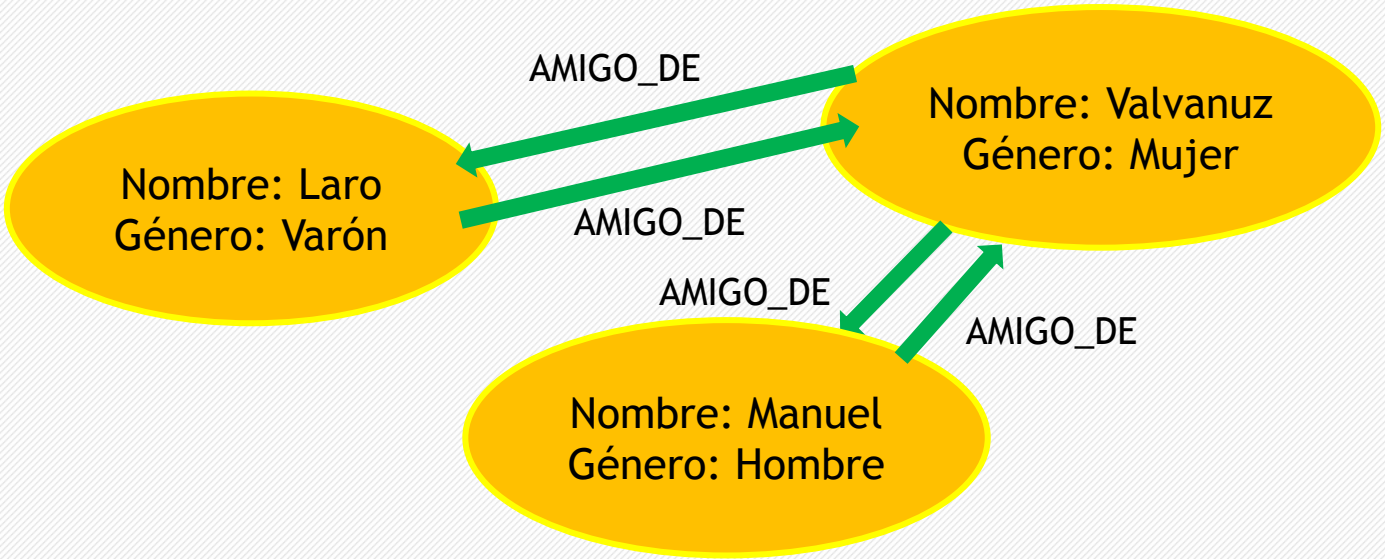
  https://neo4j.com/developer/get-started/ (oficial)

- Manuales:

  https://neo4j.com/graphacademy/ (oficial)

Introducción: Revisión histórica de Neo4j

- Base de datos NoSQL orientada a grafos. Almacena la información utilizando estructuras en grafo formadas por nodos y aristas.



- En vigor la versión 3.2.1, publicada en Junio de 2017.
- Dispone de tres versiones: *Community* (de uso libre), *Enterprise* y *Government*.

Introducción: ¿Quién y para qué se usa Neo4j?

- Diferentes empresas internacionales utilizan Neo4j con diferentes objetivos:
  - Ebay: para la logística de los servicios de entrega (planificación de itinerarios).
  - Airbnb: portal interno para que los empleados de la empresa puedan explorar los datos de clientes y reservas.
  - IBM: análisis de impacto o recomendaciones en tiempo real en base a los datos.
  - Otros usuarios: Microsoft, UBS, Walmart, NASA, etc.

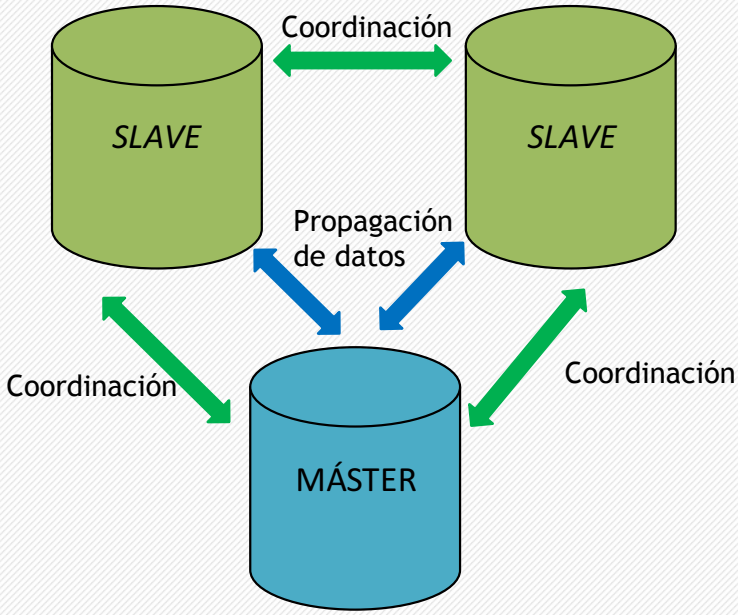
Puede consultarse más información acerca de casos de uso y usuarios de Neo4j en el siguiente enlace: https://neo4j.com/customers/

Arquitectura

- En la versión actual, existen dos posibles arquitecturas, diferentes e independientes, para desplegar una base de datos basada en Neo4j:
  - *High Available Cluster*: Es la arquitectura que ofrece desde sus inicios Neo4j, que trata de garantizar al máximo la accesibilidad a los datos aunque varios nodos caigan.
  - *Casual Cluster*: introducida por primera vez en la versión 3.1 de Neo4j, su principal objetivo es dotar de una mayor escalabilidad a la base de datos y una mayor eficiencia de las operaciones de lectura mediante la deslocalización de los nodos, que pueden estar distribuidos en diferentes espacios geográficos.

Arquitectura: *High Available Cluster*

- En arquitectura *High Available Cluster*, un clúster está compuesto de una instancia principal (*máster*) y cero o más instancias secundarias (*slave*).
  - Todas las instancias, tanto la principal como las secundarias, tienen una copia completa de toda la base de datos creada con Neo4j.
  - Todas las instancias se comunican constantemente con el resto para coordinarse y conocer su estado (flechas verdes en la imagen).
  - Las instancias secundarias se comunican con la principal para obtener actualizaciones de los datos (flechas azules en la imagen).
  - Las lecturas se pueden hacer sobre cualquier instancia, garantizando la accesibilidad (disponibilidad).

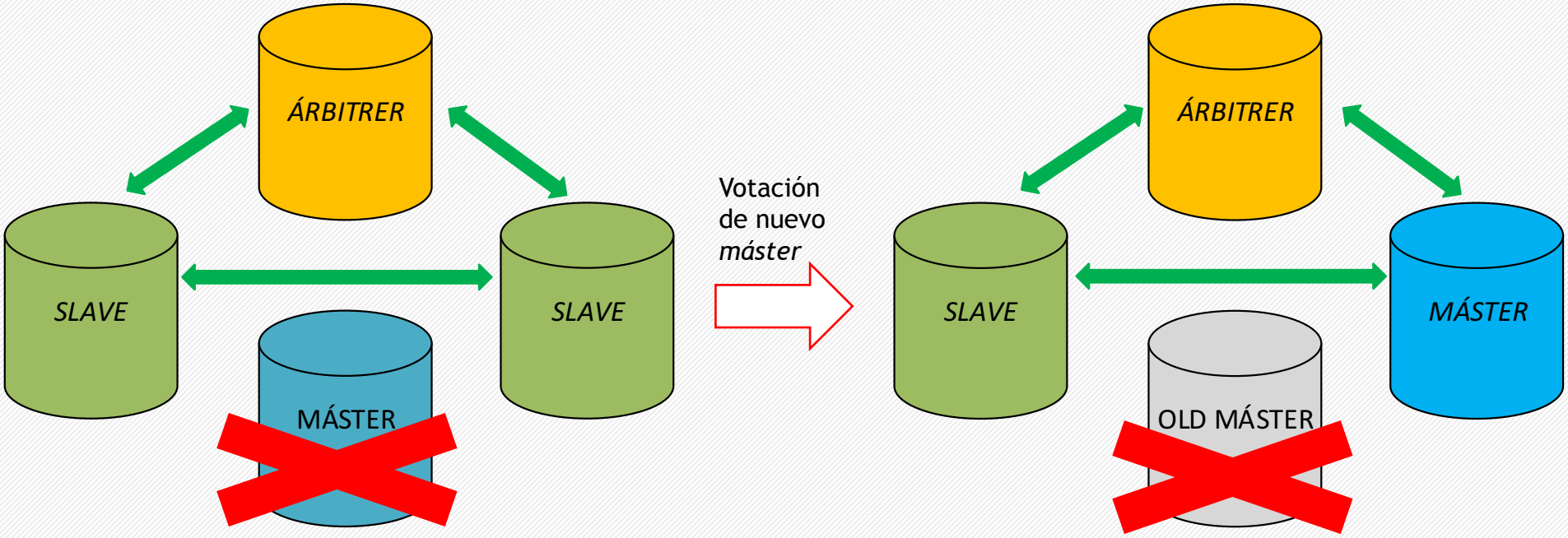


Arquitectura: *High Available Cluster*

- Las escrituras pueden hacerse directamente sobre la instancia principal o sobre una secundaria:
  - Si se realizan directamente sobre la principal, una vez terminada con éxito la escritura, esta es propagada a un número determinado de secundarias.
  - Si se realizan sobre una instancia secundaria, esta es propagada a la instancia principal. En este caso, la escritura no se considerará exitosa hasta que se escriba en el principal. Además, la instancia secundaria tiene que estar correctamente actualizada, esto es, tener los mismos datos que la principal en el momento de la escritura.

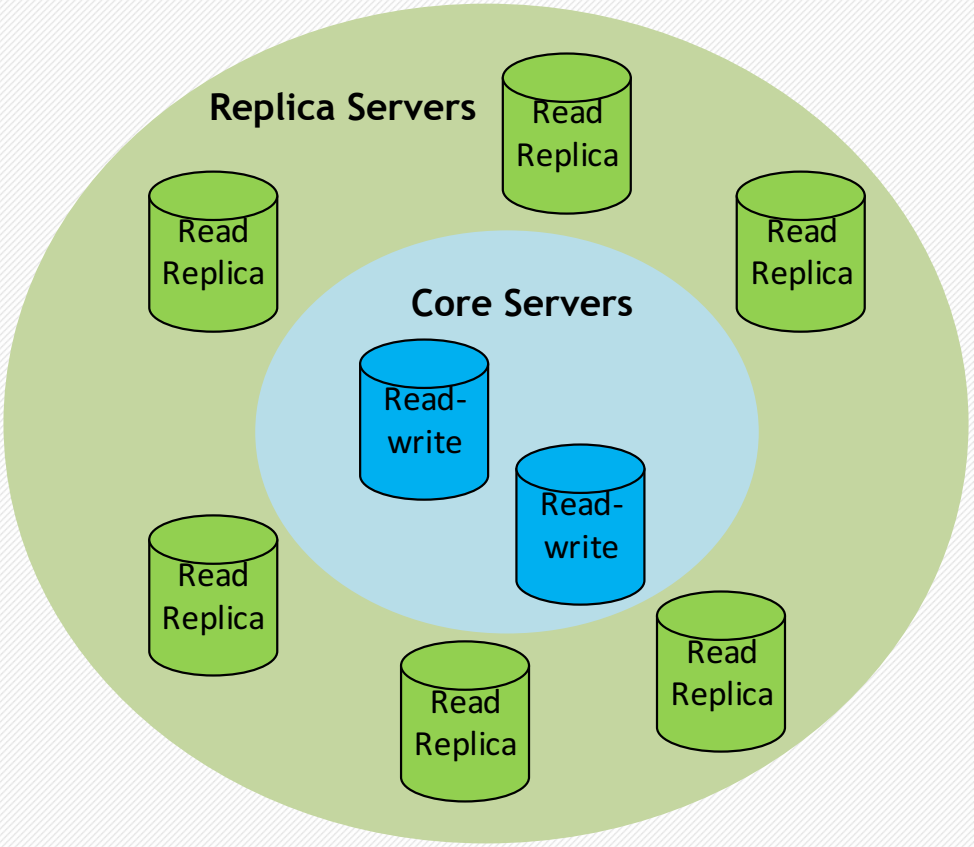
Arquitectura: *High Available Cluster*

- En caso de que la instancia principal deje de estar operativa, el resto de instancias votan para determinar cuál va a ser, de entre ellas, la nueva instancia principal.
  - En caso de que se prevean empates, pueden crearse instancias de desempate de votos (*arbitrer*). Esta instancia no contiene réplicas de los datos.



Arquitectura: *Casual Cluster*

- En la arquitectura *Casual Cluster* existen dos tipos de componentes:
  - *Core Servers*: su principal cometido es el de salvaguardar los datos, siguiendo el protocolo *Raft* (https://raft.github.io/). De esta forma, la escritura de datos recibida por parte de una aplicación cliente se propaga desde un *Core Server* al resto, y se considera exitosa si, al menos, la mitad más uno de los *Core Servers* la ha efectuado con éxito.
  - *Read replicas*: el cometido de estas réplicas es el de escalar la carga de trabajo de las operaciones de lectura. Contienen réplicas de los datos almacenados en los Core Servers, y pueden recibir peticiones de lectura por parte de los clientes. Cada determinado tiempo (usualmente, milisegundos), lanzan una petición asíncrona a un *Core Server* para recibir una copia actual de los datos.



Arquitectura: *Casual Cluster*

- *Casual Consistency*: la arquitectura garantiza que un cliente que haya efectuado determinadas operaciones de escritura, al realizar una operación de lectura este mismo cliente vea siempre los datos que el mismo ha escrito.
  - Al ejecutar una transacción, el cliente solicita una marca (*bookmark*) que luego podrá "presentar" como parámetro en subsecuentes transacciones.
  - Utilizando ese *bookmark*, el clúster puede garantizar que el cliente sólo efectuará lecturas sobre servidores que hayan procesado la transacción del cliente que generó ese *bookmark*.

Tareas de administración

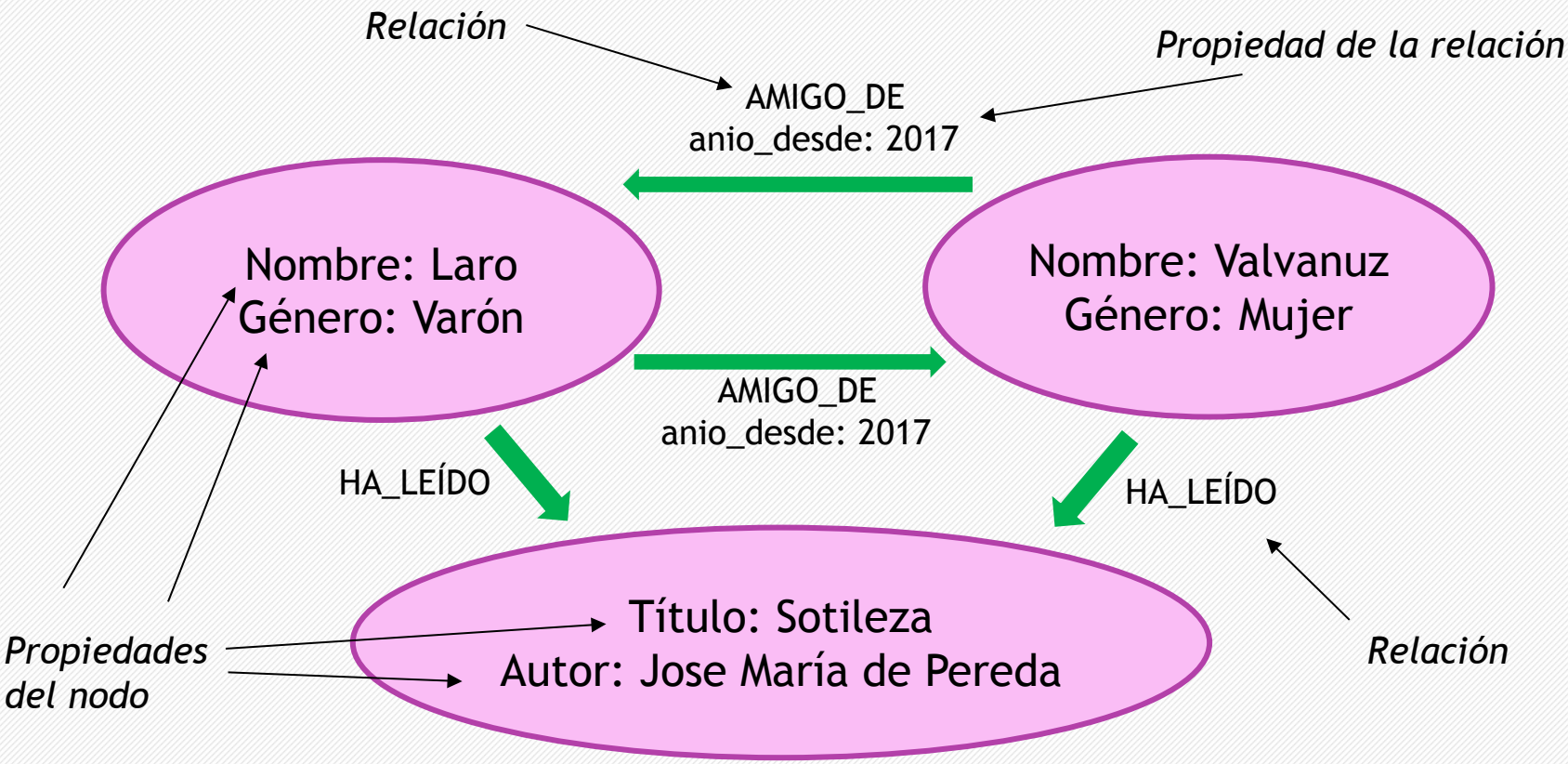
- Procesos de backup (https://neo4j.com/docs/operations-manual/current/backup/).
  - Ofrece planes de *backup* para los despliegues basados en *casual clustering* (https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/backup-planning/)
- Seguridad:
  - Ofrece procesos de autentificación y autorización, que incluye la creación y uso de roles (https://neo4j.com/docs/operations-manual/current/security/authentication-authorization/)
  - Provee de extensiones de seguridad: *Sandboxing* y *White listing* (https://neo4j.com/docs/operations-manual/current/security/securing-extensions/)
- También ofrece mecanismos de monitorización. (https://neo4j.com/docs/operations-manual/current/monitoring/)

Modelo de datos

- Neo4j es una base de datos orientada a grafos que se compone de dos elementos fundamentales:
  - **Nodos**, que representan entidades con un concepto de identidad único. Por ejemplo, pueden representar personas en la base de datos.
  - **Relaciones**, que representan conexiones o interacciones entre los diferentes nodos. Por ejemplo, una relación puede representar una conexión de amistad entre dos personas representadas como nodos.
- Tanto los nodos como las relaciones pueden contener **propiedades**, que son equivalentes a las columnas de las tablas en el modelo relacional:
  - Siguiendo el ejemplo anterior, un nodo que representa a una persona podría tener como propiedades el nombre de la misma o su fecha de nacimiento.
  - Una relación de amistad entre nodos que representan a personas podría tener como propiedades, por ejemplo, la fecha en la que comenzó dicha amistad.
  - Las relaciones son unidireccionales: por ejemplo, si una amistad entre dos personas es recíproca, habrán de existir dos relaciones entre los mismos nodos, una en cada sentido.

Modelo de datos

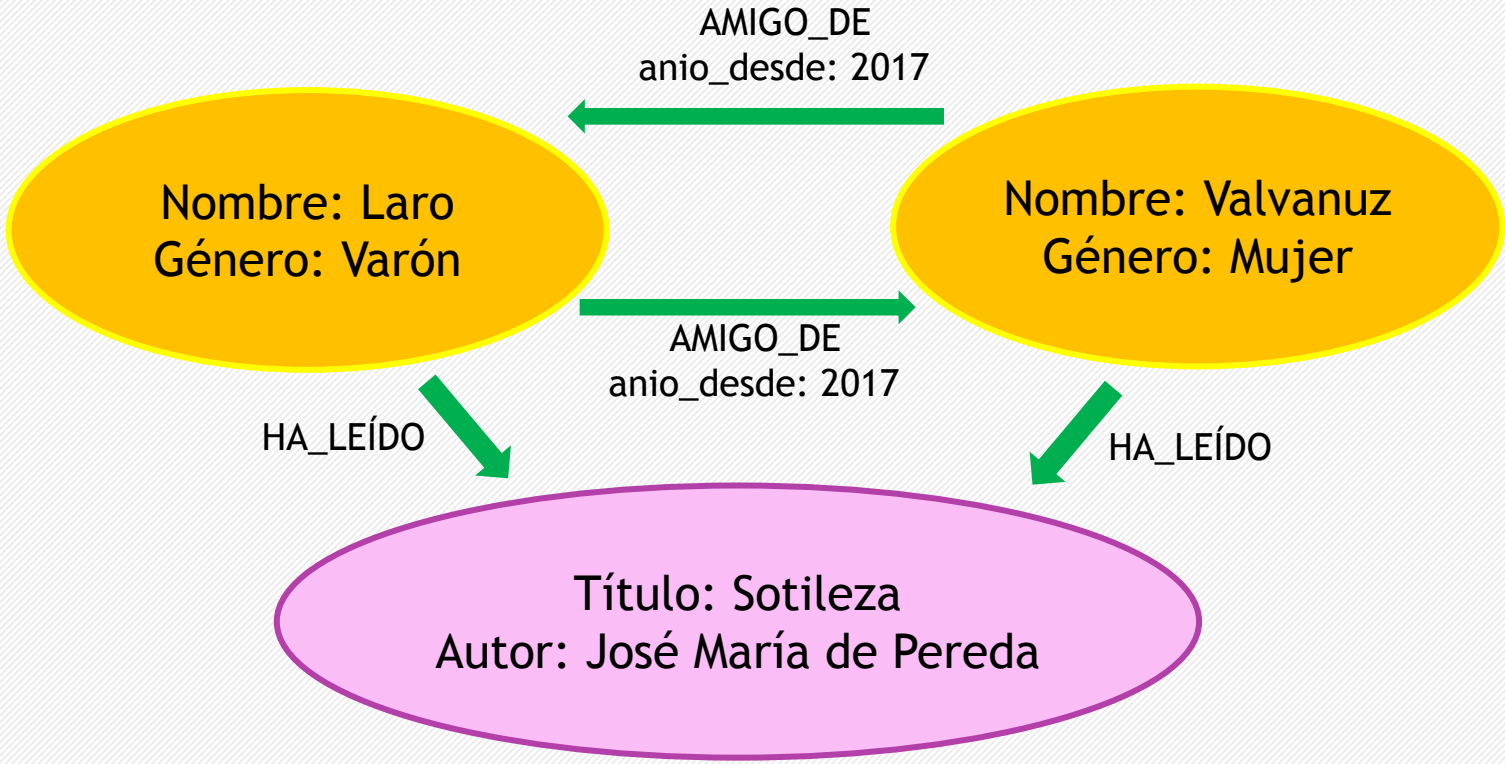
- La siguiente imagen muestra un ejemplo* de base de datos en Neo4j:



*Nota: este ejemplo se basa en el mostrado en la guía de modelado oficial de Neo4j, accesible en el siguiente enlace: https://neo4j.com/developer/guide-data-modeling/

Modelo de datos

- Los nodos puede tener **etiquetas** (ninguna o varias), que se utilizan para agruparlos. En el ejemplo anterior, podríamos definir dos etiquetas: una llamada “personas” (en naranja), que asignaríamos a Laro y Valvanuz, y otra llamada “libros” (en morado), que asignaríamos al nodo restante:



Criterios de diseño

- El proceso de modelado consiste en crear una estructura gráfica que exprese las preguntas que queremos hacer de nuestro dominio, por ello se debe:
  - 1. Identificar las preguntas del dominio.
  - 2. Identificar las entidades y las relaciones que aparecen en estas preguntas.
  - 3. Traducirlas en expresiones Cypher.
- En general,
  - Los nombres comunes se convierten en etiquetas: "usuario" y “libro”
  - Los verbos se convierten en nombres de relación: “leido“. Estas pueden contener propiedades que indiquen la intensidad de la relación.
  - Un nombre propio (nombre de una persona, p.ej) se refiere a una instancia (nodo) y sus características se definirán como propiedades.
  - Los nombres comunes que se pueden generalizar (personas incluye usuarios, administradores y supervisores) se les asignará una etiqueta.

Lenguaje de consulta y manipulación de datos: Cypher

- Cypher es el lenguaje desarrollado por Neo4j para la creación de bases de datos así como para realizar operaciones de inserción, modificado, borrado y consulta de los datos.
  - Es un lenguaje declarativo, inspirado en SQL.
  - Orientado a la descripción de patrones en grafos.

- Los ejemplos de uso de Cypher que se muestran a continuación se basan en la guía oficial, que puede ser consultada en el siguiente enlace:
  - https://neo4j.com/developer/cypher-query-language/

Cypher: tipos de datos

- En Cypher, los tipos de datos se dividen en dos grupos:
  - Básicos:
    - Booleano: puede tomar como valores verdadero o falso.
    - Integer: entero de 64 bits (equivalente al tipo long en Java).
    - Float: coma flotante de 64 bits (equivalente al tipo double en Java).
    - String: valor de tipo texto.
    - List: lista de elementos ordenados (equivalente al tipo list en Java).
    - Map: mapa de pares clave/valor (equivalente al tipo map en Java).
  - De estructura:
    - Node: almacena un nodo del grafo con sus propiedades.
    - Relationship: almacena una relación del grafo con sus propiedades.
    - Path: almacena una ruta en el grafo.

Más información en:
https://neo4j.com/docs/developer-manual/current/drivers/cypher-values/

Cypher: crear un nodo con CREATE

- La operación CREATE permite crear nuevos nodos.
- Ejemplo de creación de un nodo con etiqueta y propiedades:

Variable temporal en la que se almacena el nodo. Puede servir para seguir trabajando con él en la creación de relaciones (ver siguiente trasparencia).

```
CREATE(LaroNodo:persona {nombre: 'Laro', anio_nac: 1988})
```

Operación para crear nodos

Etiqueta

Propiedades

Cypher: crear relaciones con CREATE y MATCH

- La operación CREATE permite también crear relaciones entre nodos. Las diferentes operaciones CREATE han de ejecutarse juntas para poder acceder a las variables temporales que apuntan a los diferentes nodos creados:

```
CREATE(LaroNodo:persona {nombre: 'Laro', anio_nac: 1988})

CREATE(ValvanuzNodo:persona {nombre: 'Valvanuz', anio_nac: 1993})

CREATE (ValvanuzNodo)-[:AMIGO_DE {anio_desde:[2017]}]->(LaroNodo),
       (LaroNodo)-[:AMIGO_DE {anio_desde:[2017]}]->(ValvanuzNodo)
```

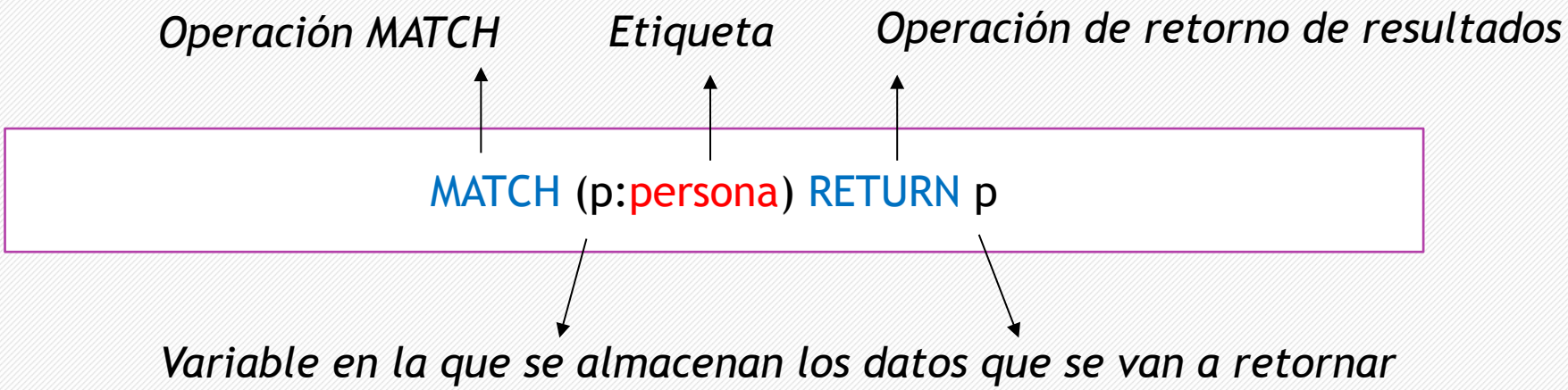
Dos relaciones, una en cada dirección

Nodo origen *Relación* *Propiedades de la relación* *Nodo destino*

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- El operador MATCH permite hacer consultas con condiciones sobre los datos de los nodos y sus relaciones.
- La siguiente consulta devolvería todos los nodos etiquetados como “persona”:



Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- ¡Ojo!. En las consultas es donde se muestra la importancia de las etiquetas en los nodos.
- La siguiente consulta devolverá los datos de todos los nodos de la base de datos ya que no indica que los nodos pertenezcan a una etiqueta:

```
MATCH (p) RETURN p
```

Al no indicar ninguna etiqueta, en la variable p se almacenan todos los nodos

Cypher: consultas con el operador MATCH

- Al igual que en el SELECT de SQL, la operación MATCH en Cypher permite indicar qué propiedades se quieren mostrar (proyección).
- La siguiente consulta mostrará solamente el campo nombre de los nodos con etiqueta “persona”:

```
MATCH (p:persona) RETURN p.nombre, p.anio_nac
```

Se muestran los campos “nombre” y “anio_nac” de las personas retornadas

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Al igual que en el SELECT de SQL, la operación MATCH en Cypher tiene una cláusula WHERE para expresar condiciones.
- La siguiente consulta sólo devolverán aquellas personas con nombre “Laro”:

```
MATCH (p:persona) WHERE p.nombre='Laro' RETURN p.nombre, p.anio_nac
```

Condición WHERE

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Se pueden utilizar los operadores AND, OR y XOR para concatenar condiciones en el WHERE:
  - Retorna a las personas que se llamen Laro o hayan nacido posteriormente al año 1992:

```
MATCH (p:persona) WHERE p.nombre='Laro' OR p.anio_nac>1992 RETURN p
```

  - Retorna a las personas que se llamen Laro y además hayan nacido posteriormente al año 1992:

```
MATCH (p:persona) WHERE p.nombre='Laro' AND p.anio_nac>1992 RETURN p
```

  - Retorna a las personas que se llamen Laro o hayan nacido posteriormente al año 1992, sin retornar aquellas que cumplan simultáneamente ambas condiciones:

```
MATCH (p:persona) WHERE p.nombre='Laro' XOR p.anio_nac>1992 RETURN p
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Para comparar valores y campos, se tienen los siguientes operadores: = (igual que), <> (distinto de), > (mayor que), >= (mayor o igual que), < (menor que), <= (menor o igual que).
- La siguiente consulta devuelve todas las personas cuyos años de nacimiento estén comprendidos entre 1980 y 1990, que se apelliden “García” y no se llamen “Manuel”:

```
MATCH (p:persona) WHERE p.anio_nac>=1980 AND p.anio_nac<1991 AND
p.nombre='García' AND p.nombre<>'Manuel 'RETURN p
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Otros operadores:
  - IN: similar a SQL, retorna aquellos nodos cuyo valor en la propiedad esté entre un conjunto de valores dados. En el ejemplo siguiente se retorna a todas las personas nacidas en 1993 o 1995:

```
MATCH (p:persona) WHERE p.anio_nac IN [1993,1995] RETURN p
```

  - STARTS/ENDS WITH: comprueba si el string comienza o termina por una secuencia de caracteres. En el siguiente ejemplo, devuelve las personas cuyo nombre comience por L:

```
MATCH (p:persona) WHERE p.nombre STARTS WITH "L" RETURN p
```

  - CONTAINS: comprueba si el string contiene una cadena concreta de caracteres. En el siguiente ejemplo, devuelve las personas cuyo nombre contiene la secuencia "ar":

```
MATCH (p:persona) WHERE p.nombre CONTAINS "ar" RETURN p
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Al igual que en SQL, en Cypher existen los siguientes operadores:
  - ORDER BY: ordena de forma ascendente (ASC) o descendente (DESC) los resultados:

```
MATCH (p:persona) RETURN p ORDER BY p.nombre ASC, p.anio_nac DESC
```

  - LIMIT: limita el número de nodos retornados:

```
MATCH (p:persona) RETURN p LIMIT 1
```

  - DISTINCT: si un resultado se repite, sólo lo devuelve una vez:

```
MATCH (p:persona) RETURN DISTINCT p
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Cypher ofrece operaciones de agregación como count, max, sum, avg, etc., al igual que SQL.
  - La siguiente consulta devuelve el número de personas que han nacido posteriormente al año 1980:

```
MATCH (p:persona) WHERE p.anio_nac>1980 RETURN count(*)
```

- En el caso de Cypher, no existe cláusula GROUP BY, la agregación es implícita. La siguiente consulta devuelve el número de personas nacidas posteriormente al año 1980, agrupadas por género:

```
MATCH (p:persona) WHERE p.anio_nac>1980 RETURN p.genero, count(*)
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- La gran potencialidad en una base de datos orientada a grafos está en poder acceder a las relaciones entre nodos y a sus propiedades.
- Las operaciones de MATCH que usan relaciones son conceptualmente similares a los JOIN en SQL.
  - Con el operador MATCH, se pueden realizar este tipo de consultas. Siguiendo con el ejemplo mostrado anteriormente, la siguiente consulta retorna todas las relaciones de amistad. En este caso, retornará las dos relaciones creadas con anterioridad: Valvanuz como amiga de Laro, y Laro como amigo de Valvanuz (ojo a la dirección de la flecha "->", que indica el sentido de la relación que se consulta):

```
MATCH relation=(p1:persona)-[r:AMIGO_DE]->(p2:persona) RETURN relation
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Las consultas sobre relaciones también pueden tener condiciones. Podríamos limitar la consulta anterior para sólo devolver las relaciones de amistad cuya persona de origen haya nacido en 1988:

```
MATCH relation=(p1:persona)-[r:AMIGO_DE]->(p2:persona) WHERE
p1.anio_nac=1988 RETURN relation
```

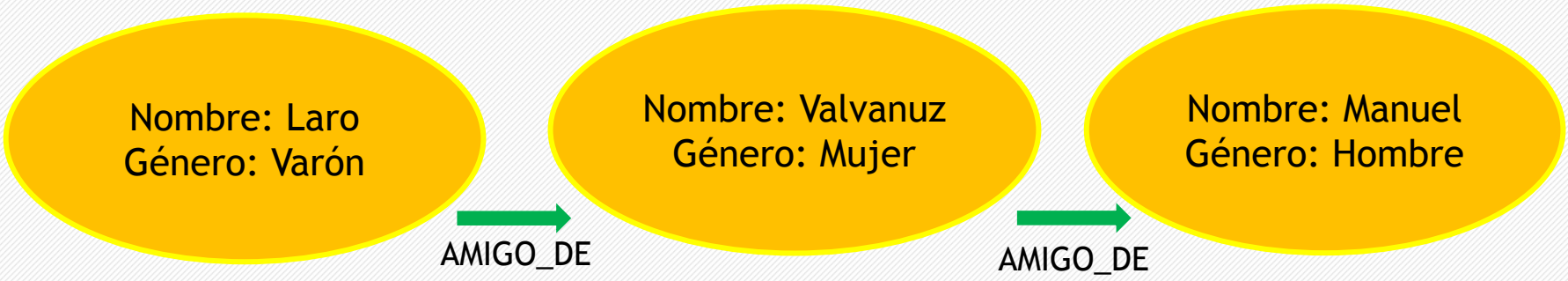
- Igualmente, se pueden utilizar los valores de las propiedades de las relaciones en el WHERE:

```
MATCH relation=(p1:persona)-[r:AMIGO_DE]->(p2:persona) WHERE r.anio_desde
=2017 RETURN relation
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- También se pueden definir consultas que incluyan varios niveles de una relación. Veámoslo con el siguiente ejemplo: supongamos que tenemos el siguiente esquema, al que se añade un nuevo usuario del que Valvanuz es amiga:



- La siguiente consulta devolvería solamente el nombre de Valvanuz, la única que tiene relación de amistad directa con Laro:

```
MATCH relation=(p1:persona)-[r:AMIGO_DE]->(p2:persona) WHERE
p1.nombre='Laro' RETURN p2.nombre
```

Cypher: consultas con el operador MATCH

M.Zorrilla - D.García

- Podríamos retornar el nombre de todos los amigos que sean amigos de los de Laro. Es decir, los amigos de Valvanuz. Para ello, añadiríamos la distancia a la relación de la siguiente forma:

```
MATCH relation=(p1:persona)-[r:AMIGO_DE*2]
->(p2:persona) WHERE p1.nombre='Laro' RETURN p2.nombre
```

Distancia 2, sólo retornaría a Manuel

- Podríamos por otra parte retornar nodos a diferentes distancias. Por ejemplo, si quisiésemos retornar a todos los amigos de Laro (distancia 1) o a los amigos de sus amigos (distancia 2), ejecutaríamos lo siguiente:

```
MATCH relation=(p1:persona)-[r:AMIGO_DE*1..2]
->(p2:persona) WHERE p1.nombre='Laro' RETURN p2.nombre
```

Distancia 1 o 2

Cypher: el operador SET

- El operador SET puede utilizarse para añadir nuevas propiedades a un nodo.
- En la siguiente instrucción, se añadiría la propiedad apellido1 al nodo con etiqueta persona y cuyo valor en la propiedad nombre sea Laro. Si la propiedad ya existe en el nodo, simplemente se actualizaría con el nuevo valor:

```
MATCH (p:persona) WHERE p.nombre='Laro'
SET p.apellido1 = 'Ceballos'
RETURN p
```

- También puede usarse para eliminar propiedades:

```
MATCH (p:persona) WHERE p.nombre='Laro'
SET p.apellido1 = NULL
RETURN p
```

Cypher: el operador REMOVE

- El operador REMOVE puede usarse, al igual que el operador SET, para eliminar propiedades:

```
MATCH (p:persona) WHERE p.nombre='Laro'
REMOVE p.apellido1
RETURN p
```

- También puede usarse para eliminar etiquetas de un nodo. La siguiente instrucción elimina la etiqueta persona de aquellos nodos que tengan una propiedad nombre con valor Laro:

```
MATCH (p) WHERE p.nombre='Laro'
REMOVE p:persona
RETURN p
```

Cypher: el operador DELETE

- El operador DELETE puede usarse para eliminar nodos. La siguiente instrucción elimina los nodos con etiqueta persona con nombre Laro:

```
MATCH (p:persona) WHERE p.nombre='Laro' DELETE p
```

- ¡Ojo!. Para eliminar un nodo, es necesario eliminar antes sus relaciones. Éstas pueden eliminarse también con el operador DELETE. La siguiente instrucción elimina todas las relaciones de amistad cuyo origen es Laro:

```
MATCH relation=(p1:persona)-[r:AMIGO_DE]
->(p2:persona) WHERE p1.nombre='Laro' DELETE r
```

- Pueden eliminarse en una sola operación un nodo y sus relaciones usando DETACH. La siguiente instrucción elimina al nodo Laro y todas sus relaciones tanto de origen como de destino:

```
MATCH (p:persona) WHERE p.nombre='Laro'  DETACH DELETE p
```

Cypher: *constraints*

- Cypher permite definir *constraints*:
  - De unicidad: los valores de una propiedad o de un conjunto de ellas no pueden repetirse en nodos que compartan una etiqueta. En el ejemplo siguiente, la propiedad "nif" en personas sea única:

```
CREATE CONSTRAINT ON (p:persona) ASSERT p.nif IS UNIQUE
```

  - De existencia: obliga a que la propiedad exista en todos los nodos con la misma etiqueta. En el siguiente ejemplo, todos los nodos con etiqueta de persona deben tener nombre (puede ser utilizada también para obligar a la existencia de propiedades en relaciones). IMPORTANTE: esta restricción sólo está disponible en la edición Enterprise de Neo4j:

```
CREATE CONSTRAINT ON (p:persona) ASSERT exists (p.nombre)
```

  - De clave (*Node Key*): obliga a que una propiedad o un conjunto de estas existan y sean únicas. Concepto similar al de *Primary Key* en el modelo relacional. En el ejemplo siguiente, la propiedad "id" de las personas pasa a ser su clave. IMPORTANTE: esta restricción sólo está disponible en la edición Enterprise de Neo4j:

```
CREATE CONSTRAINT ON (p:persona) ASSERT (p.id) IS NODE KEY
```

Más sobre *constraints* en: http://neo4j.com/docs/developer-manual/current/cypher/schema/constraints/

Cypher: índices

- Cypher puede ser utilizado para definir índices que agilicen las consultas:
  - Simples: sólo una propiedad es parte del índice. En el siguiente ejemplo, se indiza la propiedad "dni" de las persona:

```
CREATE INDEX ON :persona(dni)
```

  - Compuesto: varias propiedades forman parte del índice. En el siguiente ejemplo, se indizan conjuntamente el nombre y el año de nacimiento de las personas:

```
CREATE INDEX ON :persona(nombre, fecha_nac)
```

Más sobre índices en: http://neo4j.com/docs/developer-manual/current/cypher/schema/index/

Ventajas de Neo4j

- Modelo muy natural para representar datos conectados. Directamente leíble y fácil de interrogar.
- Implementa consultas referidas a la estructura en grafo (recorridos, adyacencia, etc.), gracias al uso de algoritmos basados en grafos (A*, Dijsktra, etc.).
- No O/R mismatch – mapeo simple del grafo al lenguajes orientado a objetos como Ruby, Java, C#, phyton.
- Es altamente disponible (HA) y tolerante a la partición (AP), lo que puede hacer que devuelva datos fuera de sync.

Desventajas de Neo4j

- El modelo de datos no está estandarizado → dificultad de cambio de gestor.
- Cypher no es un lenguaje estandarizado.
- Falta de herramientas para ETL, modelado,…
- Replicación de grafos completos, no de subgrafos (sharding).