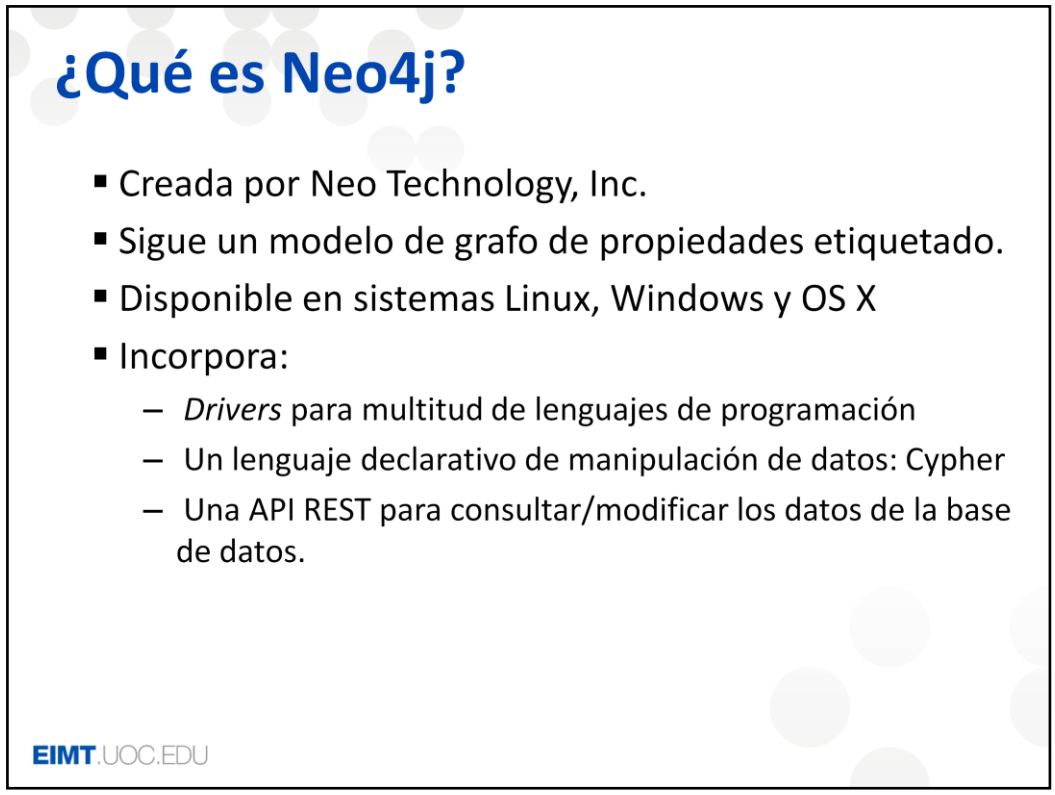
Bienvenidos. En esta presentación vamos a explicar brevemente una de las bases de datos NoSQL en grafo más conocidas actualmente: Neo4j.

Índice

- Introducción y características
- Modelo de datos
- Operaciones CRUD
- Estrategias de distribución
- Recursos y enlaces

EIMT.UOC.EDU

En la primera parte de la presentación comentaremos qué es Neo4j y explicaremos sus principales características. Posteriormente, hablaremos del modelo de datos que utiliza: un grafo de propiedades etiquetado. En la tercera parte de la presentación, explicaremos cómo usar Cypher para añadir, actualizar, borrar y consultar datos de la base de datos. Finalmente, veremos las posibilidades de replicación y distribución que ofrece esta base de datos y mostraremos un conjunto de enlaces a recursos relativos a Neo4j.



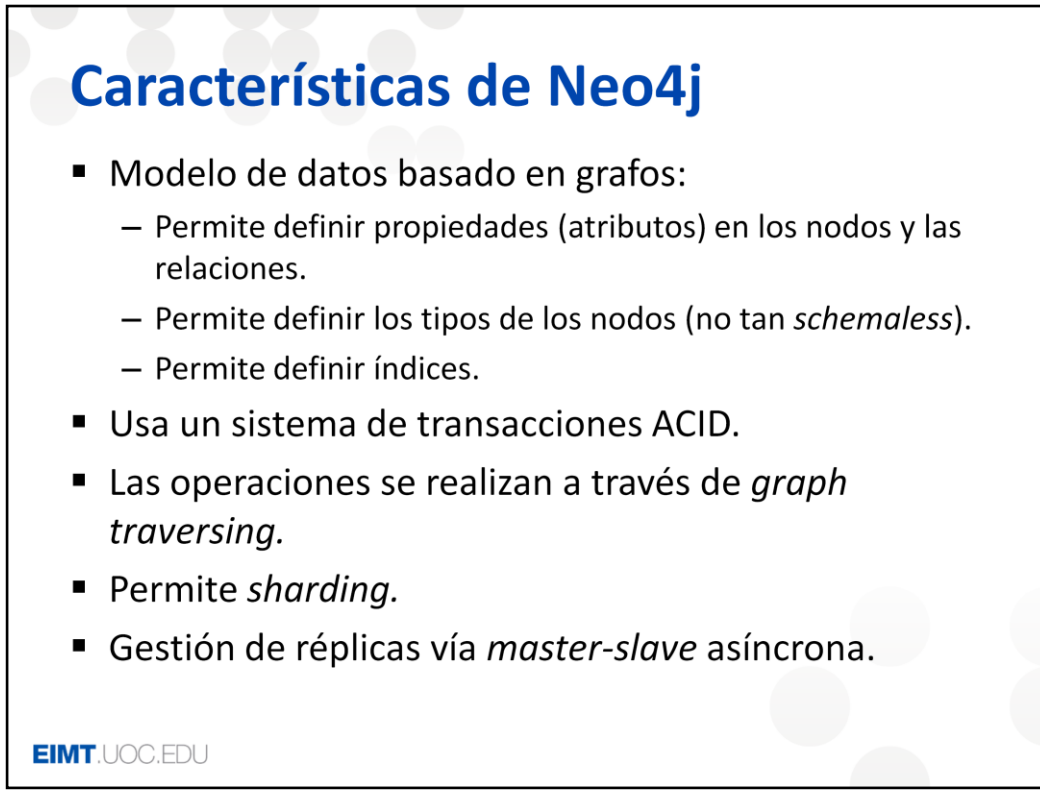
Neo4j es una base de datos en grafo creada por Neo Technology Inc. Actualmente es la base de datos en grafo más popular según el *ranking* de db-engines.com (http://db-engines.com/en/ranking). A fecha 2014, también es la número 30 de 221 en el *ranking* general de bases de datos.

El modelo de datos utilizado es un grafo de propiedades etiquetado. Las unidades básicas de procesamiento en Neo4j son: nodos, relaciones entre nodos, propiedades definidas sobre nodos o relaciones y etiquetas que permiten definir el tipo de los nodos y de las relaciones. Aunque es una base de datos *schemaless,* el uso de etiquetas permite asociar tipos a los elementos de los grafos y además permite definir algún tipo de restricción de integridad (actualmente sólo de unicidad sobre las propiedades).

Neo4j está disponible en sistemas operativos de las familias Windows, Linux y OS X.

Neo4j permite acceder a sus datos de diversas formas y usando distintos lenguajes de consulta. Sus datos pueden ser accedidos desde una consola de texto, un entorno web (con salida gráfica) y mediante APIs. Respecto a las API, Neo4j dispone de *drivers* para multitud de lenguajes de programación y una API REST que permite consultar y modificar los datos de la base de datos mediante peticiones HTTP. Respecto a sus lenguajes de consulta, destacamos Cypher, que es un lenguaje declarativo que permite consultar y manipular grafos, y Gremlin, que es un lenguaje específico de dominio para la gestión de grafos.

Existe una versión gratuita llamada Community Edition y una versión de suscripción llamada Enterprise Edition. La principal diferencia entre ellas es que la segunda incorpora soporte y provee funcionalidades de replicación, monitorización y alta disponibilidad. La versión Community no soporta fragmentación ni replicación.



Tal y como ya se ha comentado, los datos en Neo4j se almacenan utilizando grafos de propiedades etiquetados. Por lo tanto se pueden definir propiedades (o atributos) tanto sobre nodos como sobre sus relaciones. Al ser una base de datos de tipo *schemaless* se puede añadir información sin la necesidad de crear un esquema de los datos a priori. No obstante, en las últimas versiones de Neo4j, se permite añadir etiquetas a los nodos (y a las relaciones) para representar el tipo al que pertenecen. Esto posibilita operaciones de consulta/modificación más expresivas, pudiendo preguntar cosas, como por ejemplo, "Dame todos los nodos que sean de tipo pedido".

Aparte, como veremos más adelante, Neo4j permite definir índices sobre los datos para mejorar el rendimiento de las consultas. Estos índices básicamente permiten indexar las propiedades de los nodos.

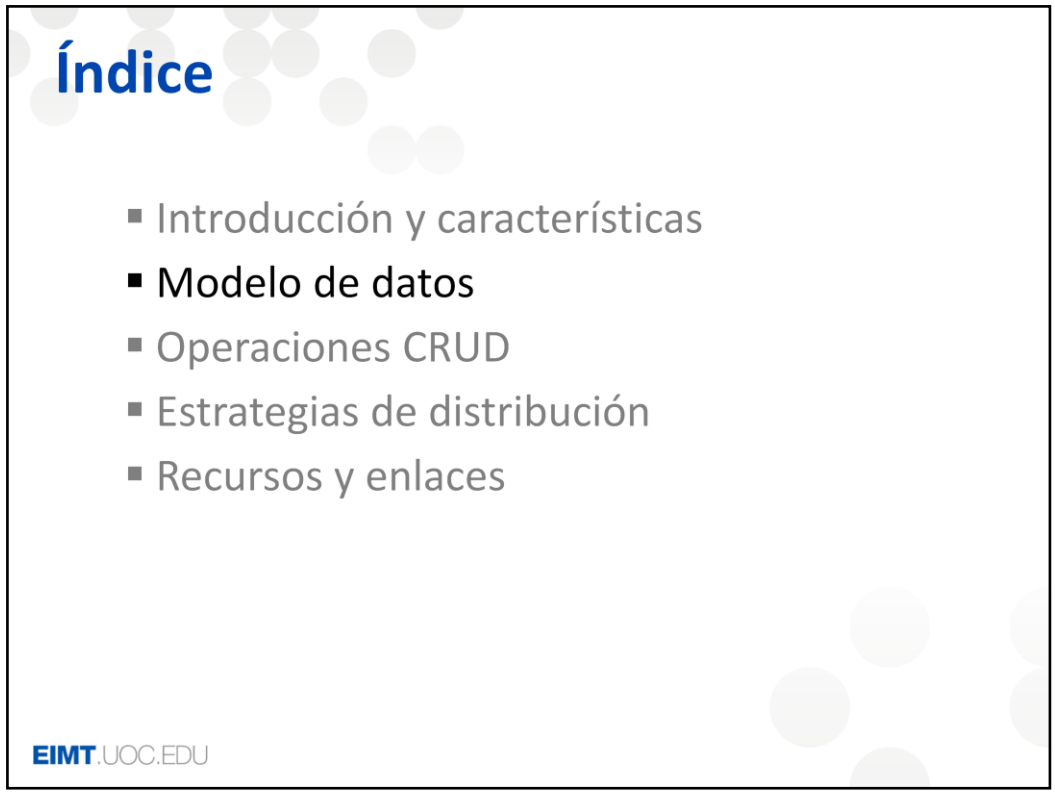
A diferencia de las bases de datos que hemos visto anteriormente, Neo4j utiliza un sistema de transacciones ACID. Las transacciones en Neo4j pueden contener diferentes operaciones. Aunque utilice un sistema de transacciones ACID, la gestión de réplicas que realiza puede causar que la consistencia que se ofrece en caso de distribución y/o replicación de datos, a diferencia del modo de trabajo preferido en una base de datos relacional, pueda no ser estricta (*strict consistency*).

La forma de trabajar con los datos de Neo4j es totalmente diferente a lo que hemos visto hasta ahora: en las bases de datos NoSQL basadas en modelos de agregación se accedía a los datos por medio de sus claves y en las bases de datos relacionales los datos se acceden a partir de sus tipos (relaciones), sus interrelaciones (claves foráneas), sus identificadores (claves primarias) y sus atributos. En Neo4j, para identificar un conjunto de datos sobre los que aplicar una operación, se realiza lo que se denomina *graph traversing*. Podríamos traducir *graph traversing* como "navegación por el grafo" y es la acción de navegar por el grafo, a partir de un punto de inicio (o un conjunto de puntos de inicio) para obtener los resultados deseados (es decir, obtener los elementos sobre los cuales realizar la operación).

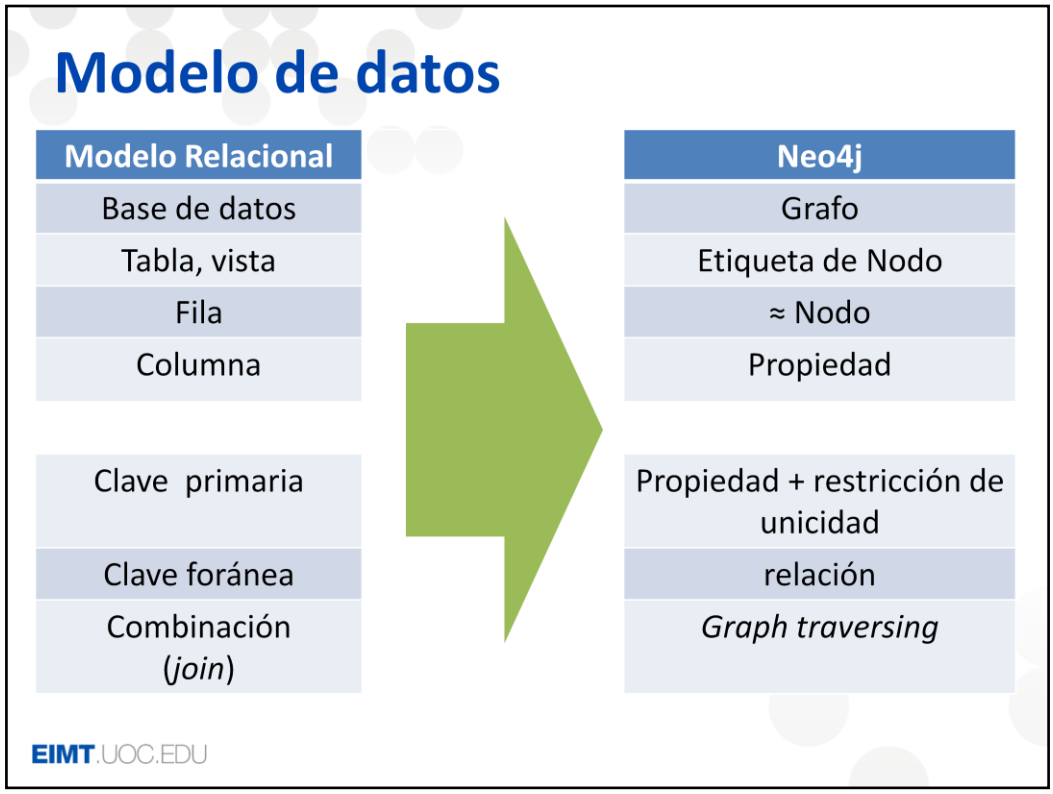
Neo4j permite *sharding*, es decir, permite la distribución de los grafos en distintos servidores. No obstante, el *sharding* no se realiza de forma automática, sino que debe definirse a la vez que se diseña la base de datos y requiere de un gran conocimiento sobre el dominio a modelar. Si se desea realizar *sharding*, es importante hacer un buen diseño de distribución. Si no fuese así, nos podemos encontrar con bajos rendimientos. El hecho es que, si separamos partes del grafo relacionadas que se consultan a menudo, deberemos acceder a diferentes nodos del grafo para navegar por el grafo. En el caso peor, puede darse el caso que la navegación entre N nodos del grafo implique consultar N servidores de forma secuencial (en el caso de que cada nodo del grafo estuviera en un servidor distinto). El hecho de que Neo4j no incorpore por defecto técnicas de MapReduce aún complica más la consulta de datos en bases de datos fragmentadas. Como el oyente puede intuir, las capacidades (y problemáticas) de escalabilidad horizontal de Neo4j son similares a las de las bases de datos relacionales.

La replicación en Neo4j se realiza mediante técnicas de *master-slave* asíncrona de forma similar a MongoDB.

Las peculiaridades de los grafos y el diseño de Neo4j hacen que esta base de datos soporte mejor la escalabilidad vertical que la horizontal. Neo4j está enfocada a garantizar la consistencia de los datos y la disponibilidad. Esto quiere decir que en algunas situaciones el sistema puede quedar parcialmente inoperativo en caso de particiones en la red.



Una vez introducida la base de datos Neo4j, vamos a ver más en detalle su modelo de datos, los índices que incorpora y las equivalencias entre su modelo de datos y el modelo relacional.

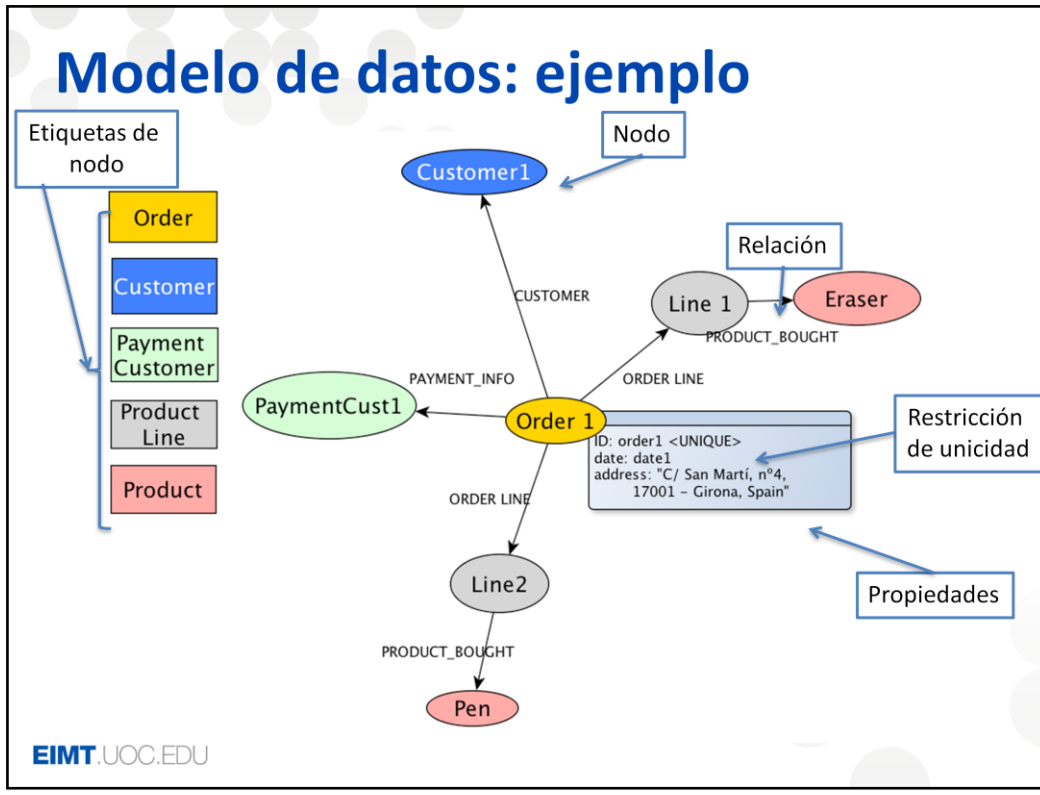


Para explicar el modelo de datos de Neo4j utilizaremos el modelo relacional como base. En Neo4j los datos se almacenan en un grafo de propiedades etiquetado. El grafo contiene un conjunto de nodos, relaciones, etiquetas y propiedades. Semánticamente, los nodos serían el equivalente a las filas del modelo relacional, aunque puede haber casos donde interesará definir una relación o un evento como un nodo. Los nodos pueden tener un conjunto de propiedades asociadas. Estas propiedades son pares *<nombre, valor>* y permiten definir un conjunto de propiedades para cada nodo y sus valores. Las propiedades serían el equivalente a las columnas en una base de datos relacional.

Continuando con las propiedades, desde la versión 2.0 de Neo4j, se permite (con ciertas limitaciones) definir restricciones de integridad de unicidad sobre las propiedades de la base de datos. Por lo tanto, se puede especificar que una propiedad determinada de la base de datos, en el contexto de un tipo de nodo, no puede contener valores duplicados. Al indicar que una propiedad es única, Neo4j crea un índice sobre dicha propiedad para poder garantizar la unicidad y mejorar la eficiencia de su consulta. Al definir este tipo de restricciones sobre las propiedades de los nodos, se pueden simular claves primarias.

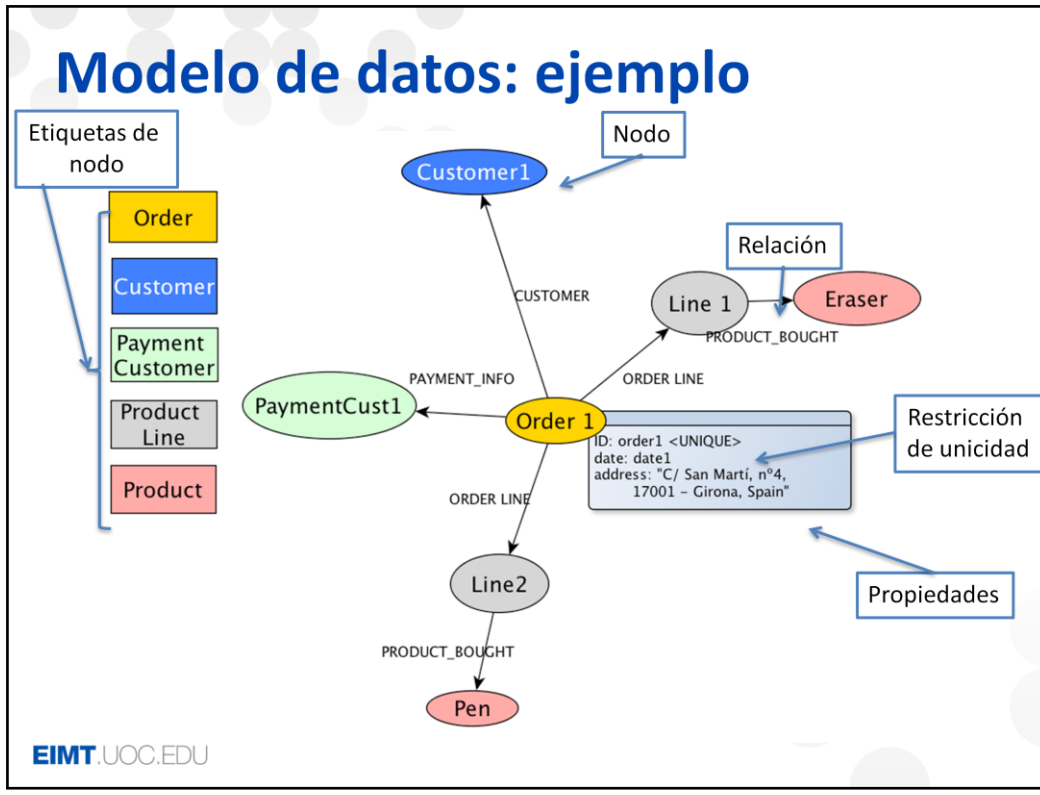
Para representar las relaciones entre los nodos de la base de datos se utilizan relaciones, que serían el equivalente a las claves foráneas del modelo relacional. Es importante comentar que también se pueden definir propiedades sobre las relaciones. Las relaciones en Neo4j son dirigidas y binarias.

En Neo4j podemos navegar por el grafo para obtener datos o realizar operaciones sobre los mismos (lo que se conoce como *graph traversing*). Esta navegación nos permite, entre muchas cosas, simular la operación de combinación (*join*) relacional.



En esta transparencia podemos ver un ejemplo que representa un pedido de venta y sus elementos relacionados en una base de datos en grafo. Los pedidos se almacenan como nodos y cada pedido constituye un nodo distinto. En este caso, podemos ver como el nodo *Order 1* determina el pedido que hemos estado utilizando en el ejemplo. Véase también que se ha utilizado una etiqueta llamada *Order* para indicar que el nodo *Order 1* es de tipo *Order*. Una vez definido el nodo del pedido y su tipo, se pueden indicar sus datos mediante propiedades. En el ejemplo podemos ver que se han definido 3 propiedades: el identificador del pedido (*ID*), la fecha del pedido (*date*) y la dirección de envío (*address*). Además, en este caso, hemos decidido definir en la propiedad *ID* una restricción de unicidad, por lo tanto, no se permitirán dos propiedades *ID* con el mismo valor en nodos de tipo *Order*. Las líneas de pedido se han representado mediante un conjunto de nodos y relaciones, pero también podrían haberse representado mediante propiedades. Expresarlo de una forma u otra constituye una decisión de diseño de la base de datos y deberá escogerse en función del uso esperado de la base de datos.

El cliente que realizó el pedido se ha definido mediante un nodo *Customer 1* de tipo *Customer*. Para indicar el hecho de que *Customer 1* es el cliente que realizó el pedido, se ha creado una relación con nombre "CUSTOMER" entre los nodos *Order 1* y *Customer 1*. De forma parecida se ha creado un nodo para los datos de pago llamado *PaymentCust1* y una relación para indicar que dichos datos de pago son relativos al pedido *Order 1*. Notad que la nomenclatura utilizada para los nodos y las relaciones son diferentes, los primeros se indican con la primera letra en mayúscula y los segundos con todas las letras en mayúscula.



Respecto a las líneas de pedido, se ha creado un tipo de nodo llamado *Product Line* y se han definido dos nodos de este tipo para las líneas de pedido: *Line 1* y *Line 2*. Dichas líneas se han relacionado con el pedido mediante dos relaciones, ambas llamadas *ORDER LINE*. Fijaos que aunque diferentes nodos del mismo tipo contengan nombres distintos, las relaciones comparten nombre. Eso es porque el nombre de la relación se utiliza en Neo4j para identificar su tipo. Es decir, como las relaciones entre *Order 1* y *Line 1* y entre *Order 1* y *Line 2* tienen el mismo nombre (ese nombre es *ORDER LINE*), eso significa que son del mismo tipo: *ORDER LINE*. Esto nos resultará de utilidad más adelante a la hora de consultar el grafo, ya que nos permitirá realizar consultas que satisfagan un patrón, como por ejemplo "Dame todas las líneas de pedido del producto *Eraser*". El patrón detrás de esta consulta sería "los nodos de tipo *Product Line* relacionados mediante una relación *ORDER LINE* con un nodo cuyo nombre es *Eraser".* Los nodos del grafo que verifiquen este patrón satisfarán la consulta anterior.

Los productos que aparecen en cada línea de pedido se han definido mediante un nuevo tipo de nodo denominado *Product*. En particular, se han definido dos nodos de este tipo: *Eraser* y *Pen.* Para indicar que estos nodos aparecen en una línea de pedido se ha creado dos relaciones de *PRODUCT BOUGHT* que los relacionan con las líneas de pedido en las que aparecen.

En este ejemplo, en aras de la simplicidad, hemos creado una versión simplificada del posible grafo resultante, obviando algunos nodos y propiedades que podrían haberse añadido. Aún siendo incompleto, el número de nodos y relaciones creados en el ejemplo es elevado. En un caso real el número de elementos del grafo crece rápidamente. Por ese motivo, será necesario hacer un buen diseño del grafo para identificar qué conceptos deben ser representados mediante nodos, cuáles mediante relaciones y cuáles mediante propiedades. Asimismo, es importante hacer una buena gestión de índices para optimizar la consulta de los datos.

Índices

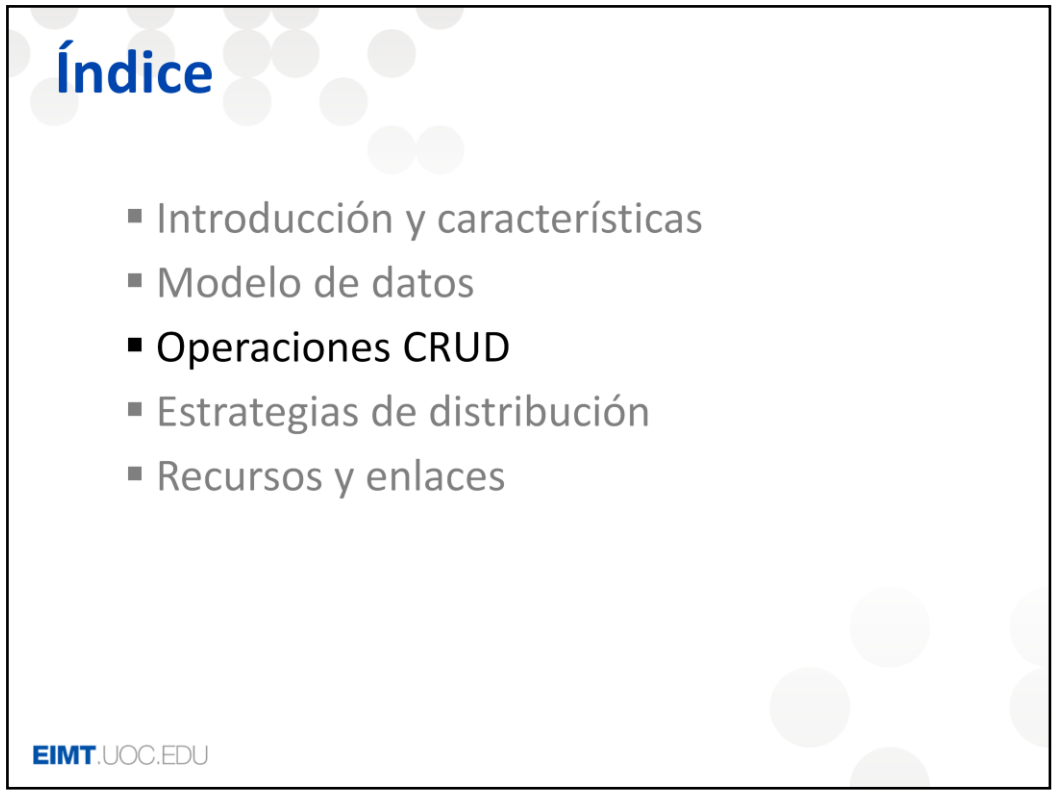
- Se indexan las propiedades de los grafos:
  - Permiten utilizar elementos del esquema de datos.

    Ejemplo: Crear un índice sobre la propiedad *country* de los nodos de tipo *Person*
  - Los índices se crean de forma asíncrona.
- Existe otro tipo de índices, denominados *Legacy Index*
  - Permite definir índices personalizados.
  - Actualmente no se pueden realizar mediante Cypher.

EIMT.UOC.EDU

En Neo4j hay dos tipos de índices, los índices que se definen sobre las propiedades de los grafos y los denominados *legacy index*. Los segundos utilizan el motor de indexación Lucene y permiten una indexación más personalizada y potente, pero deben ser gestionados vía API. Como en la documentación de Neo4j se recomienda utilizar el primer tipo de índice desde la versión 2.0 y son los únicos que pueden gestionarse mediante Cypher, éstos serán los que explicaremos aquí.

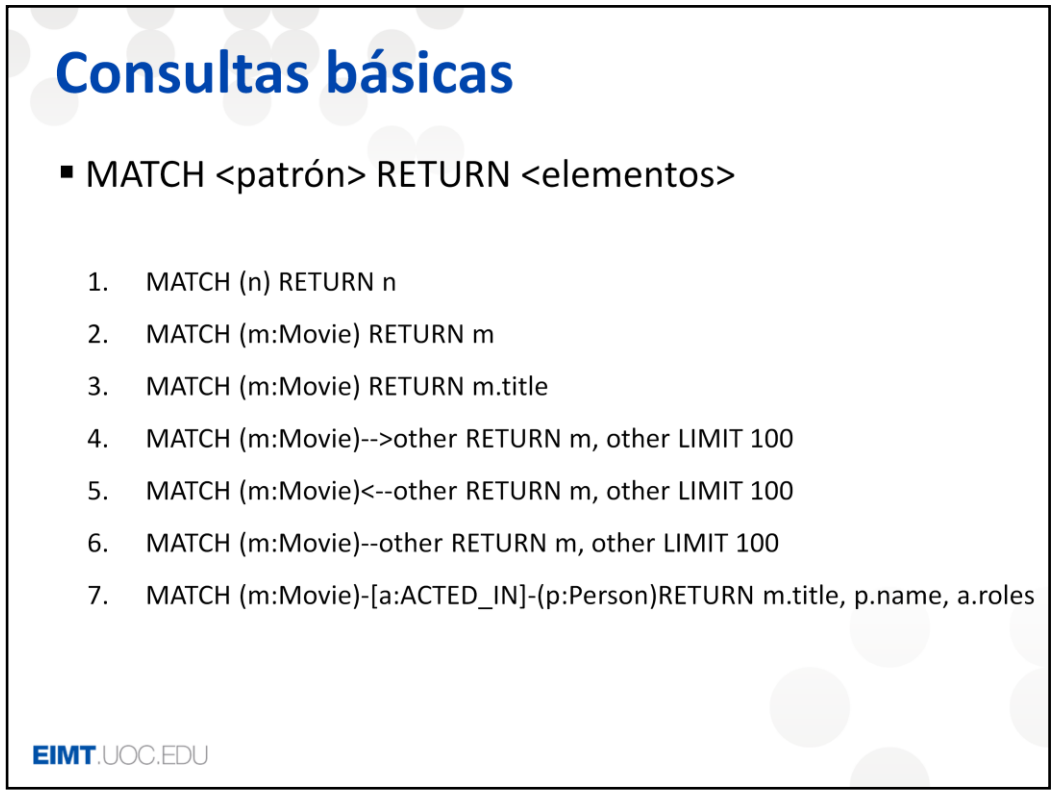
Los índices del primer tipo permiten indexar las propiedades de los nodos de un tipo determinado, es decir, permiten indexar una propiedad para un conjunto de nodos que comparten la misma etiqueta. Por ejemplo, podríamos definir un índice para la propiedad *country* de todos los nodos de tipo *Person*. Al crear un índice de este tipo hay que tener en cuenta que su creación se realiza en modo *batch*. Por lo tanto, entre que el usuario solicita la creación del índice al gestor de la base de datos y el índice está creado de forma efectiva puede pasar un lapso de tiempo (quizá considerable). Esto se deberá tener en cuenta al realizar consultas inmediatamente después de la creación de un índice. Más adelante vamos a ver con más detalle cómo crear y utilizar índices en Cypher.



Hasta aquí la introducción a Neo4j y a su modelo de datos. A continuación vamos a presentar algunas de sus operaciones para de creación (C), lectura (R), actualización (U) y borrado de datos (D). Tal y como hemos dicho, Neo4j se puede acceder usando distintos lenguajes, en esta presentación nos vamos a centrar en el uso de Cypher.

Para trabajar con Cypher es aconsejable descargar la *Neo4j Cypher Refcard* que se encuentra en la página de documentación de Neo4j (http://docs.Neo4j.org/). La sintaxis de Cypher es compleja y permite muchas posibilidades, en esta presentación vamos a efectuar un recorrido introductorio por sus principales características. Se recomienda a los oyentes que exploren sus posibilidades por su cuenta.

Las consultas de ejemplo que vamos a utilizar en esta parte de la presentación están basadas en una base de datos de películas que viene con Neo4j, llamada *The Movie Graph*. Se aconseja que tengáis una consola de Neo4j y la base de datos cargada para que podáis probar las operaciones que se van presentando. Para cargar esta base datos deben seguirse las instrucciones que encontraréis en la página principal del entorno web del *browser* de Neo4j, bajo el enlace *The movie graph*.

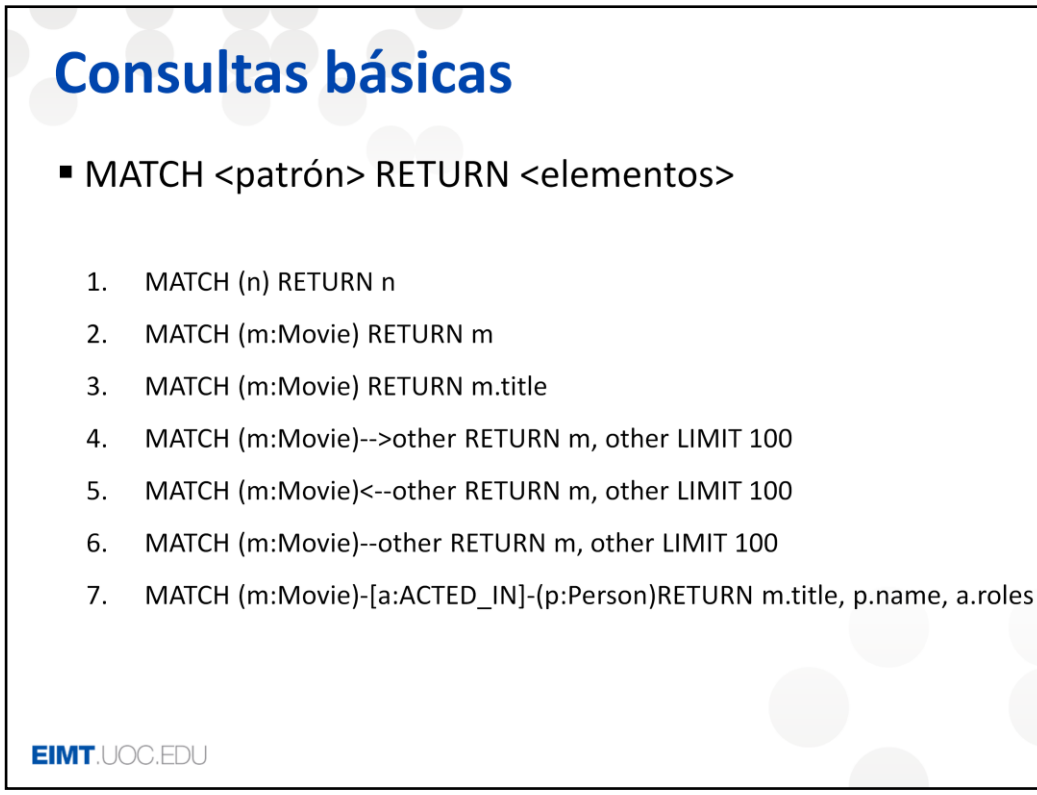


Empezaremos viendo cómo consultar datos utilizando Cypher. Las consultas se realizan mediante la instrucción *MATCH… RETURN*. La cláusula *MATCH* permite definir un patrón que identifica los elementos del grafo en los que estamos interesados. La cláusula *RETURN* permite indicar cuáles de los elementos de la cláusula *MATCH* deberá devolver la consulta.

Los patrones que definiremos en la cláusula *MATCH* se inician en uno o más nodos. A partir de un nodo inicial se puede definir el patrón (o patrones) que deben seguir los datos a devolver. Estos patrones podrán contener (entre otros) nodos, relaciones y etiquetas. Vamos a ver cómo funciona mediante algunos ejemplos.

La primera consulta devuelve todos los nodos de la base de datos. En esta consulta el patrón es *n*, siendo *n* un nodo. Por lo tanto el patrón sería “todos los nodos”. Como no hay más elementos en el patrón *MATCH,* no se aplica ninguna restricción, y en consecuencia, se identifican todos los nodos de la base de datos. La cláusula *RETURN n* indica que la consulta debe retornar los nodos *n* que satisfagan la consulta. Notad que el nombre de variable que se utiliza en el patrón (*n*) es el que se utiliza en la cláusula *RETURN* para identificar los valores a devolver.

En la segunda consulta se seleccionan un conjunto de nodos, pero con una restricción: los nodos deben ser de tipo *Movie* (al poner *nodo:etiqueta* indicamos que queremos seleccionar los nodos que sean de tipo *etiqueta*). En consecuencia, la consulta seleccionaría todos los nodos que representan películas. Al hacer un *RETURN m*, la consulta devolverá todos los nodos seleccionados. La tercera consulta es parecida a la segunda, pero en este caso, no se devuelven los nodos de las películas, sino el valor de la propiedad *title* de los mismos. Por lo tanto, esta consulta devolverá los títulos de las películas.

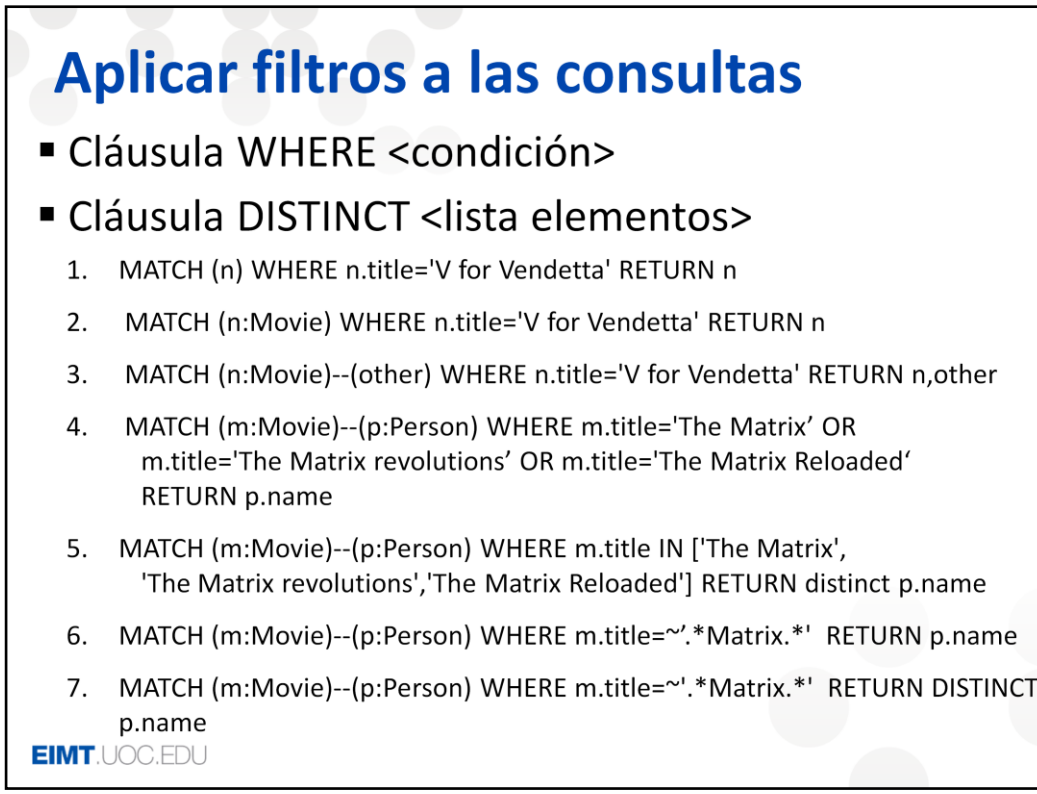


La cuarta consulta añade un nuevo elemento en el patrón *MATCH*: una relación. Las relaciones pueden indicarse añadiendo los caracteres "--" entre dos nodos. Adicionalmente, se puede añadir un signo "<" o ">" en los extremos de la relación para indicar la dirección de la relación. En consecuencia, si hay un patrón del tipo "(*a*)--(*b*)", Cypher lo interpreta como "seleccionar todos los nodos *a* y *b* que están relacionados" y si es del tipo "(*a*)-->(*b*)" se interpreta como "seleccionar todos los nodos *a* y *b* que están relacionados mediante una relación (o más) que va de *a* a *b*". En el caso de la cuarta consulta, se seleccionará el conjunto de parejas <*m*, *other*> tales que *m* es de tipo película, *other* es de cualquier tipo y existe como mínimo una relación que va de *m* a *other*. Finalmente, se retornarán (según podemos ver en la cláusula *RETURN*) los nodos *m* y *other* que satisfagan el patrón indicado. En este caso, el sistema no sólo mostraría los nodos, sino que también las relaciones entre ellos. La cláusula *LIMIT 100* limita el número de resultados que devuelve una consulta. En casos donde la consulta pueda retornar un gran número de resultados puede ser conveniente limitar la salida para no sobrecargar el sistema.

Si ejecutáis la cuarta consulta sobre la base de datos, podréis comprobar que no se devuelve ningún nodo. Esto es así porque las relaciones en las que participan los nodos de tipo *Movie* no son de salida sino de entrada. En el patrón de la consulta se piden los nodos de salida (mediante el símbolo "-->"), por eso no se devuelve ningún resultado. Si volvéis a ejecutar la consulta cambiando el sentido de la relación (consulta cinco) el resultado será el esperado.

La sexta consulta permite preguntar por datos relacionados pero sin tener en cuenta la dirección de las relaciones ("--"). En este caso, el gestor de la base de datos devolverá la unión de los resultados de las consultas 4 y 5.

A veces también puede interesar devolver los datos almacenados en las propiedades de las relaciones o aplicar filtros sobre las mismas. En la cláusula *MATCH,* para cada relación, podemos definir: 1) una variable que la identifique y que permita acceder a sus datos y 2) un tipo. Ambos elementos se deberán definir entre corchetes "[]" entre el primer y el segundo guión de la relación. En la séptima consulta podemos ver un ejemplo donde el tipo de las relaciones del grafo se utiliza para filtrar los datos devueltos. En este caso se restringe la consulta anterior para seleccionar sólo las relaciones de tipo *ACTED IN*. Por lo tanto, según este nuevo patrón, la consulta seleccionará el conjunto de tripletas del tipo <*m*, *a*, *p*> donde, *m* es un nodo de tipo película, *p* es un nodo de tipo persona y *a* es la relación de tipo *ACTED IN* que relaciona la persona *p* y la película *m*. Esta consulta devolverá, para cada película, su título (*m.title*) , el nombre de los actores que han participado en la película (*p.name*) y el nombre del personaje que han realizado (*a.roles*).



Hasta ahora hemos visto cómo consultar los datos de un grafo indicando cuál es el patrón del resultado esperado, es decir, cómo están relacionados los datos y qué tipos deben tener. Ahora vamos a ver cómo aplicar filtros a las consultas para devolver sólo aquellos datos que cumplan unas determinadas condiciones.

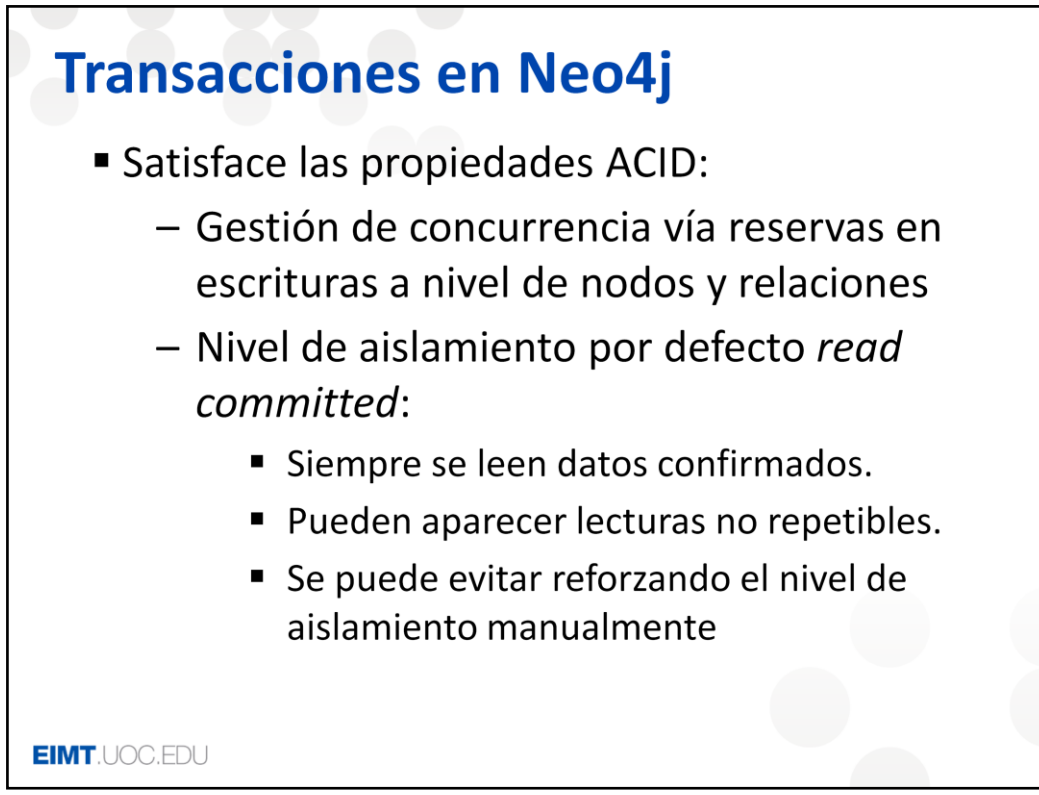
Al igual que en SQL, en Cypher se puede definir una cláusula *WHERE* para seleccionar los datos de interés. En la cláusula *WHERE* se pueden indicar un conjunto de condiciones que los datos de interés deberán satisfacer. Podemos ver un ejemplo del uso de esta cláusula en la primera consulta, que devuelve todos los nodos que tienen una propiedad con nombre *title* y valor '*V for Vendeta*'. Notad que al ser de tipo *schemaless*, en Neo4j no tenemos ninguna garantía que todos los nodos devueltos sean de tipo película. Podría darse el caso que una persona tuviera una propiedad con nombre *title* y valor '*V for Vendeta*'. En dicho caso, el gestor de la base de datos retornará también este último elemento. La segunda consulta realiza la misma consulta, pero garantiza que el resultado es de tipo película.

La tercera consulta nos permite obtener la película '*V de Vendeta'* y los otros nodos (*other*) relacionados con ella.

La cuarta consulta es un poco más complicada y retorna los nombres de los actores que han actuado en alguna de las tres películas de Matrix ("The Matrix", "The Matrix revolutions", "The Matrix Reloaded"). Notad que esta condición puede indicarse de distintas formas: concatenando un conjunto de comparaciones mediante operadores de tipo *OR* lógico (consulta 4), utilizando operaciones de conjunto (consulta 5) o utilizando expresiones regulares (consultas 6 y 7). Notad que el operador de expresión regular es el =~ y los caracteres .* indican cualquier secuencia de caracteres. Por lo tanto, la sexta consulta retornará los nombres de las personas que han actuado en alguna película que contiene el texto "Matrix" en su título.

La última consulta realiza lo mismo que las tres anteriores pero utiliza la cláusula *DISTINCT* para eliminar los valores repetidos.

Antes de continuar con las operaciones de creación, borrado y modificación vamos a hablar de cómo gestiona Neo4j las transacciones y la concurrencia.

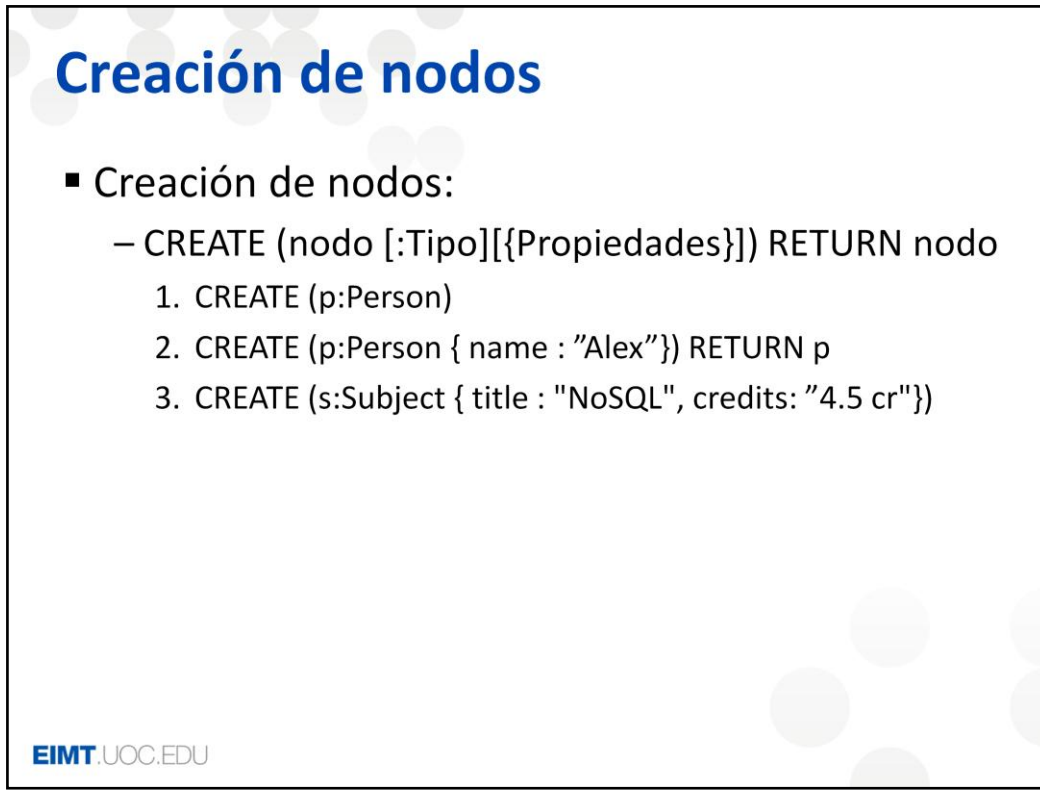


Según la documentación, el sistema transaccional de Neo4j satisface las propiedades ACID.

En Neo4j cada operación que accede al grafo, sus índices o a su esquema deben ser ejecutados dentro de una transacción. Las transacciones pueden incorporar múltiples operaciones. Al utilizar la API de Neo4j, el usuario puede definir el inicio y el final de cada transacción. Al utilizar Cypher desde el intérprete de comandos web (*web browser*) la definición de transacciones es más limitada. De hecho, parece que en dicho entorno las transacciones se definen a nivel de sentencia, es decir cada sentencia de Cypher constituye en sí misma una transacción. Hay que tener en cuenta que una sentencia en Cypher puede modificar, desde un solo nodo del grafo, a todos sus nodos, y que todo ello se ejecutará en una sola transacción. Es un aspecto a tener en cuenta al utilizar sentencias de actualización en Cypher. Si utilizamos Cypher desde el intérprete de comandos textual o en versiones anteriores del entorno web se puede indicar el inicio y el fin de una transacción con las sentencias *BEGIN* y *COMMIT*.

Respecto a la gestión de concurrencia, Neo4j utiliza el típico sistema pesimista de bloqueos (o reservas S, X), donde las operaciones de escritura bloquean los datos hasta su finalización para que no se pierdan cambios. Según la documentación, el bloqueo se realiza a nivel de nodos y relaciones de la base de datos. En particular, según la documentación, al modificar (añadir o borrar) un nodo o sus propiedades, el bloqueo se realiza únicamente sobre el nodo afectado. No obstante, al modificar (añadir o borrar) una relación, el bloqueo afecta tanto a la relación como a los nodos que participan en la relación.

Es importante mencionar que Neo4j, por defecto, garantiza un nivel de aislamiento (esto se relaciona con la propiedad I del acrónimo ACID) de *read committed*. Esto quiere decir que el sistema garantiza que siempre que se realice una operación de lectura, se obtiene el último valor confirmado. Esto, en transacciones de larga duración, puede dar lugar a lecturas no repetibles. Por ejemplo, cuando en una transacción (T1) se realizan dos lecturas de un mismo nodo (*n*) en dos momentos diferentes y otra transacción (T2) ha modificado el nodo (*n*) entre la primera y segunda lectura. En algunos casos, este comportamiento puede ser problemático. Neo4j permite que los usuarios realicen bloqueos directamente. Esto puede ser útil en caso de que se quiera reforzar el nivel de aislamiento del sistema para evitar interferencias, como por ejemplo, lecturas no repetibles.

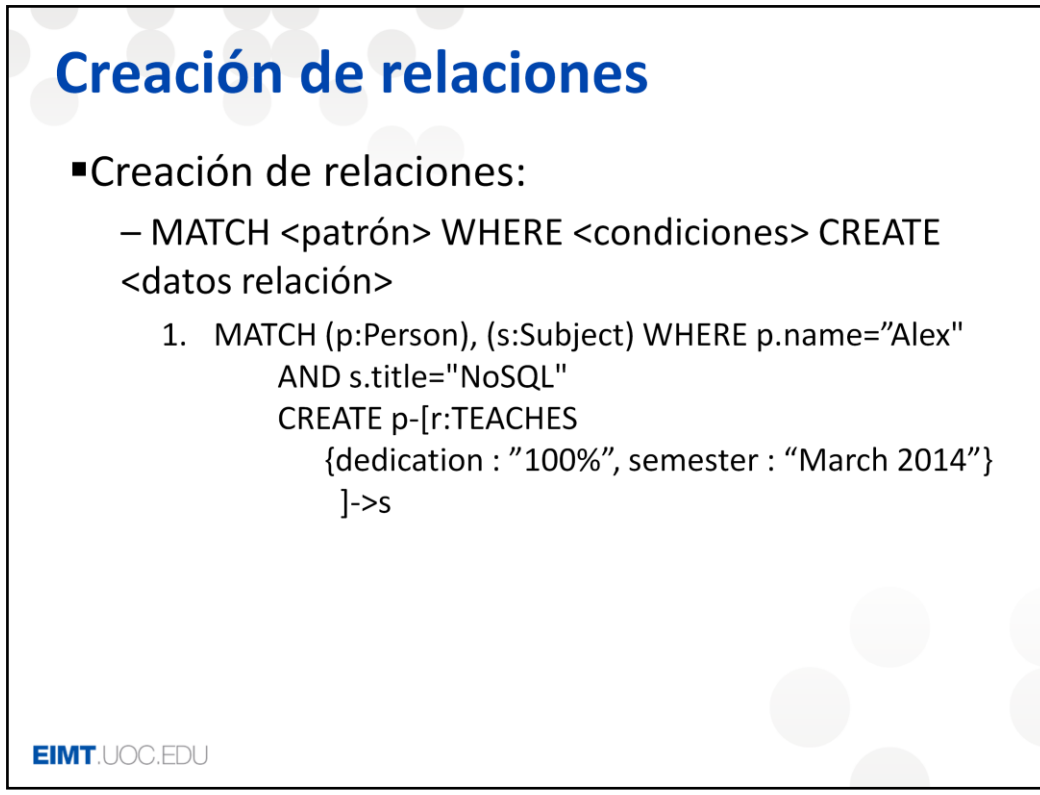


Una vez visto cómo funcionan las transacciones en Neo4j vamos a introducir cómo crear nodos y relaciones en Neo4j.

Para crear un nodo (o más de uno, si se separan por comas) en Neo4j se usa la sentencia *CREATE*. Para cada nodo a crear, el usuario podrá indicar sus tipos y propiedades. Las propiedades se definen en formato JSON.

A continuación podemos ver tres ejemplos de sentencias de creación de nodos. La primera permite crear un nodo de tipo persona sin ninguna propiedad ni relación asociada. La segunda operación permite crear un nodo de tipo persona que tiene una propiedad con nombre *name* y valor *Alex*. Notad que en este caso hay un *RETURN* al final de la operación. Con dicho *RETURN* estamos pidiendo a Cypher que nos devuelva el elemento creado como consecuencia de ejecutar la sentencia.

La tercera operación permite crear un nodo de tipo asignatura con dos propiedades que denotan, respectivamente, el nombre (o título) de la asignatura (con valor *NoSQL*) y el número de créditos de la asignatura (con valor de *4.5* créditos). Supongamos que el tipo de nodo *Subject* no existe en la base de datos. En este caso, como Neo4j no tenía definido este tipo de nodo, lo creará al ejecutar la operación. Este comportamiento puede ser peligroso porque si asignamos nodos a etiquetas erróneas (*Peson* en vez de *Person,* por ejemplo) nos podemos encontrar con el hecho de no encontrar los datos por estar mal clasificados.



En Neo4j los nodos pueden estar aislados, es decir, no tienen por qué participar en relaciones. No obstante, las relaciones deben crearse entre nodos. Por lo tanto, antes de crear una relación, se deberá identificar qué nodos relaciona. Para hacerlo, se utilizará la sentencia *MATCH … WHERE* como hemos visto anteriormente. A pesar de ello, en este caso, la sentencia terminará con una cláusula *CREATE* que permitirá especificar qué relación se debe crear y entre qué nodos.

El ejemplo utiliza esta sintaxis para crear una relación entre los dos nodos que acabamos de crear. En la sentencia, primero se identifican los nodos a relacionar (la persona con nombre *Alex* y la asignatura con título *NoSQL*). Posteriormente, en la cláusula *CREATE*, indicamos que queremos crear una relación entre estos dos nodos y le asignamos un conjunto de propiedades, que representan la dedicación del profesor y el semestre en que se ha realizado la docencia. Podemos ver que también se ha asociado la relación a una nueva etiqueta: *TEACHES*. Al igual que en el caso anterior, si el tipo de relación *TEACHES* no existiera en la base de datos, se creará como consecuencia de la ejecución de esta operación.

Para acabar, comentar que hay una operación denominada *MERGE* que permite crear elementos en el grafo siempre y cuando éstos no existan. Esta operación es similar a las operaciones de *upsert* de MongoDB.

Eliminación de nodos y relaciones

- No se puede eliminar un nodo si tiene relaciones asociadas:
  - MATCH <patrón> DELETE <lista elementos>
    1. MATCH (p:Person) WHERE p.name="Alex" DELETE p
    2. MATCH (p:Person)-[r]-(s) WHERE p.name="Alex" DELETE p, r

EIMT.UOC.EDU

En Neo4j no se puede borrar un nodo de un grafo a no ser que el grado de entrada y de salida del nodo sea cero. En otras palabras, sólo se podrán eliminar nodos que no participen en relaciones.

Para eliminar un nodo o una relación de Neo4j, primero se utiliza la sentencia *MATCH* para identificar los datos de interés y después una cláusula *DELETE* para indicar qué elemento hay que eliminar. En las cláusulas *MATCH* y *WHERE* se indicarán los patrones y condiciones que permitan identificar los elementos a eliminar. La cláusula DELETE indicará qué elementos se deben eliminar.

Aquí podemos ver un par de ejemplos para eliminar los datos insertados en la transparencia anterior. La primera sentencia intenta borrar la persona con nombre *Alex*. Para ello, indicamos que el elemento a eliminar es un nodo *p* (cláusula *DELETE*), de tipo *Person* (cláusula *MATCH*) y cuyo nombre es *Alex* (cláusula *WHERE*). Si ejecutamos esta operación veremos que Neo4j nos dará un error. Eso es debido a que el nodo a eliminar aún mantiene relaciones con otros nodos. Podríamos eliminar el nodo *Alex* de dos formas distintas: 1) eliminar primero las relaciones en las que participa el nodo y después eliminar el nodo, o 2) eliminar a la vez (con la misma operación) el nodo y sus relaciones. En esta transparencia vamos a utilizar la segunda opción (operación número 2).

La segunda sentencia selecciona un conjunto de tripletas del tipo <*p*, *r*, *s*> donde *p* es el nodo que representa a *Alex*, *r* son las relaciones que relacionan el nodo que representa a *Alex* con otros nodos y *s* son los nodos relacionados. La sentencia indica que deben borrarse los elementos *p* y *r*, es decir, el nodo que representa a *Alex* y todas sus relaciones.

Modificación de propiedades

- Añadir/modificar/borrar propiedades:
  - MATCH … [SET propiedad=valor] [REMOVE propiedad]
    1. MATCH (p:Person) WHERE p.name="Robin Williams"
       SET p.rating=5
    2. MATCH (p:Person) WHERE p.name="Robin Williams"
       REMOVE p.rating
    3. MATCH (p:Person) WHERE p.name="Robin Williams"
       SET p.rating=3, p.oscarsOwned=1

EIMT.UOC.EDU

Otras operaciones de modificación que se pueden realizar sobre los grafos de Neo4j son, por un lado, añadir y eliminar propiedades, y el por otro, modificar los tipos de los nodos y de las relaciones.

La adición (o modificación) de propiedades se realiza añadiendo la cláusula *SET* al final de una sentencia *MATCH*. La cláusula *SET* indicará el nombre y la ubicación de la propiedad y el nuevo valor que debe tener. En el caso de que la propiedad no existiera, se crearía. Por otro lado, para eliminar una propiedad de uno o más nodos, se utiliza la cláusula *REMOVE*. Es necesario indicar el conjunto de propiedades a eliminar.

A continuación presentamos tres ejemplos. En el primer caso, se añade una propiedad con nombre *rating* y valor *5* a la persona con nombre *Robin Williams*.

La segunda sentencia elimina la propiedad *rating* de los nodos de tipo persona con nombre *Robin* Williams.

La tercera sentencia permite añadir dos propiedades a la misma persona, la valoración del actor (*rating*) y el número de premios Oscars que ha ganado (*oscarsOwned*). En caso de que la propiedad *rating* ya existiera (imaginemos que no la hubiéramos borrado en la operación anterior) se modificará su valor en vez de añadir la propiedad.

Modificación de etiquetas (tipos)

- Añadir/borrar la etiqueta de un nodo o una relación:
  - MATCH … {SET|REMOVE} {nodo|relación} :Tipo1:…:TipoN

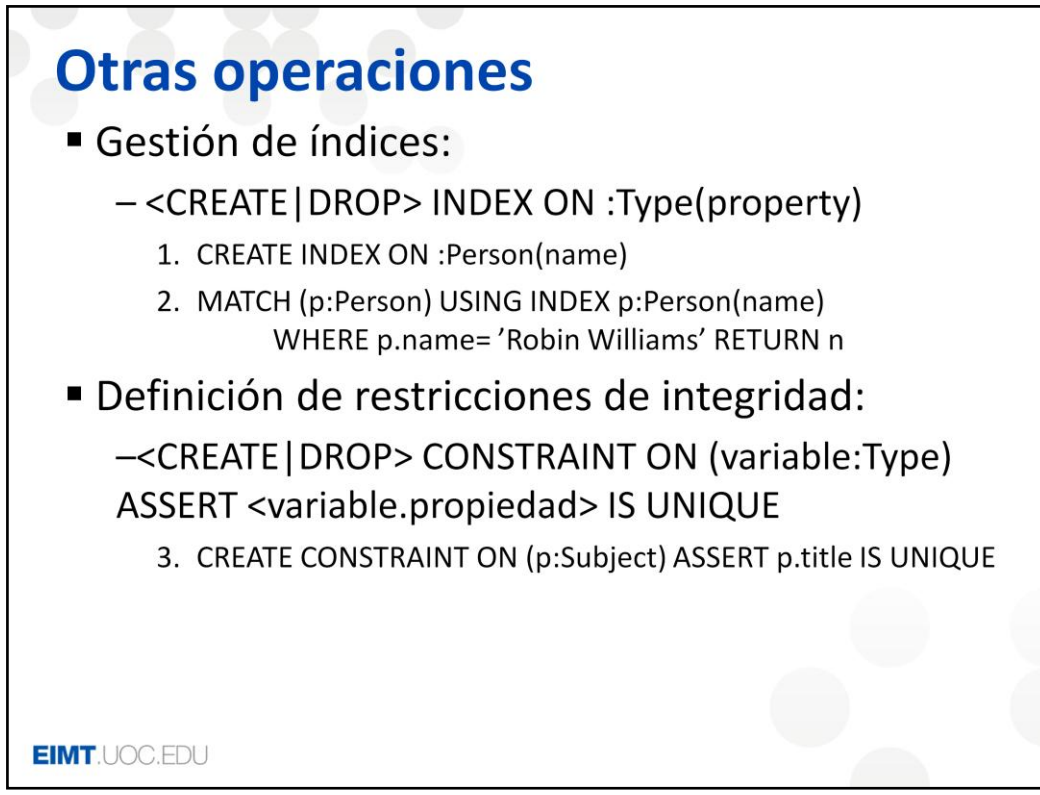
1. MATCH (p:Person)-[r:ACTED_IN]->(f) SET p :Actor
2. MATCH (p:Actor) WHERE p.name="Robin Williams"
   REMOVE p :Actor

EIMT.UOC.EDU

Como hemos comentado, también se pueden añadir o eliminar etiquetas a los nodos o relaciones de un grafo. Para hacerlo se utilizan también las cláusulas *SET* o *REMOVE*, que identifica el nodo o la relación a modificar y el conjunto de tipos a añadir o eliminar. Tal y como podéis ver en la transparencia, el número de las etiquetas que pueden asignarse (o borrarse) a un elemento del grafo puede ser superior a una. Esto permite simular que un elemento del grafo pertenezca a más de un tipo, lo que sería útil para simular modelos de clasificación múltiple o de herencia múltiple.

El primer ejemplo añade la etiqueta *Actor* a los actores, entendiendo como actores aquellas personas que participan en una relación de tipo *ACTED_IN* como nodo origen. Como resultado, después de la ejecución de esta operación, los actores tendrán dos etiquetas asociadas: *Person* y *Actor*.

El último ejemplo permite modificar el nodo del actor *Robin Williams* , desasignándole la etiqueta *Actor*. Por lo tanto, después de la ejecución de esta operación, *Robin Williams* dejaría de ser considerado como un actor en la base de datos y pasará a ser considerado simplemente como una persona.



Además de las operaciones básicas que acabamos de ver, hay otras operaciones que se pueden ejecutar en Cypher. Algunas de ellas son las relativas a la creación y gestión de índices, otras a la gestión de restricciones de integridad, la gestión de agregados, etc. A continuación vamos a ver ejemplos de algunos de estos tipos de operaciones.

La creación y eliminación de índices en Cypher se realiza mediante las sentencias *CREATE INDEX ON* y *DROP INDEX ON*. En dichas sentencias se debe indicar la propiedad que se quiere indexar y el tipo de los nodos que contienen dicha propiedad. En la transparencia podemos ver un ejemplo donde se crea un índice sobre la propiedad *name* de los nodos de tipo *Person*. Una vez creado el índice, el optimizador de Neo4j lo utilizará automáticamente para encontrar los nodos que utilizan dicha propiedad en las cláusulas *MATCH* y *WHERE*. No obstante, si es necesario, también se puede indicar a Neo4j que utilice un determinado índice en una consulta de forma explícita, tal y como muestra la segunda sentencia de ejemplo. En ella, se indica que debe usarse el índice definido sobre la propiedad *name* de los nodos de tipo *Person*.

Otro tipo de operaciones que se pueden realizar en Neo4j es la definición de restricciones de integridad de unicidad. Este tipo de restricciones se pueden definir sobre propiedades para indicar que la propiedad no puede tomar valores repetidos en el contexto de un tipo de nodo. En la transparencia podemos ver un ejemplo donde se crea una restricción de unicidad sobre la propiedad *title* que contienen los nodos de tipo *Subject*.

Otras operaciones

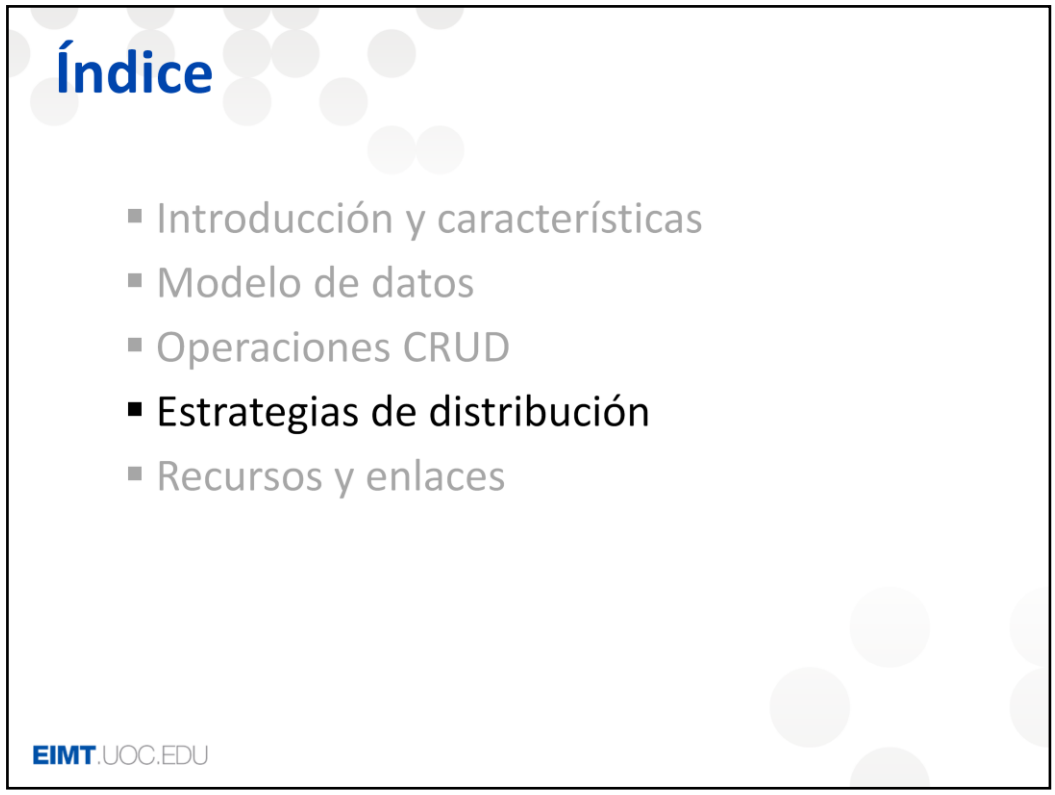
- Cálculo de agregados:
  1. MATCH (p:Person) return p.born, count(*)
  2. MATCH (p:Person)--(f:Movie) return distinct f.title, count(*)

EIMT.UOC.EDU

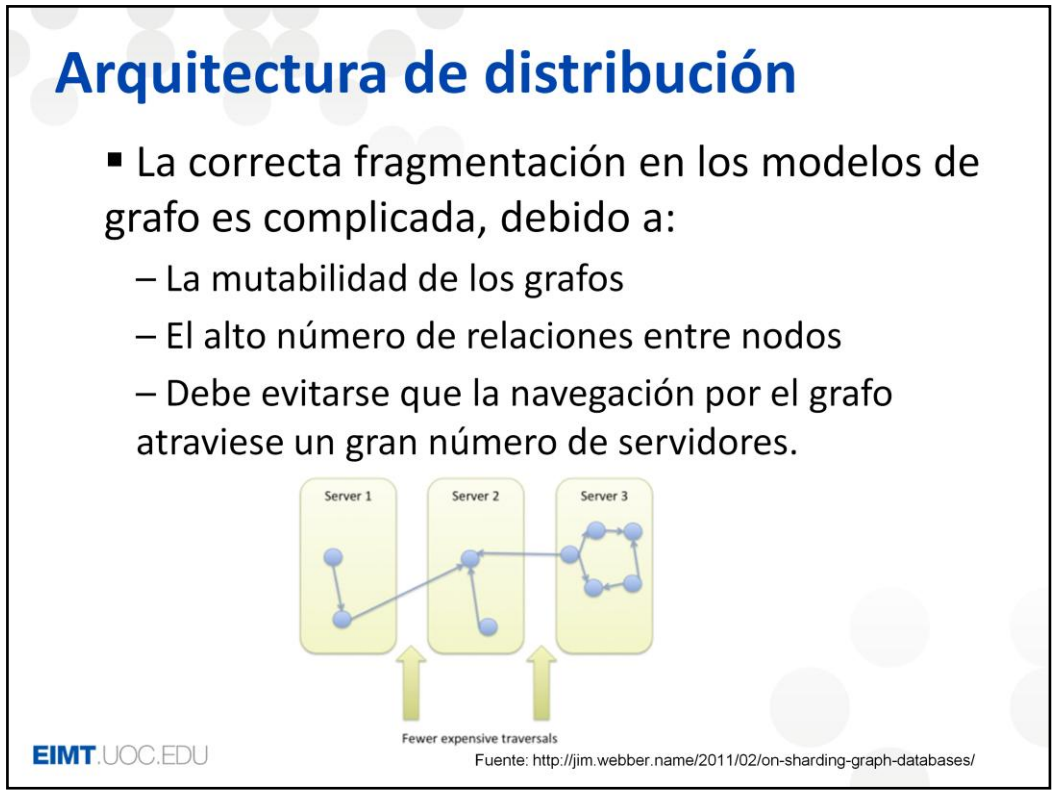
Cypher ofrece también algunas funciones para el cálculo de agregados. A continuación mostramos un par de ejemplos. El primero permite devolver los años en que ha nacido alguna persona y, para cada año, el número de personas que han nacido en ese año. Notad que el *count* en este caso nos permite contar el número de personas que han nacido en el mismo año.

En el segundo ejemplo, se muestra el número de personas que han trabajado en cada película. Para hacerlo, se obtienen todas las personas y películas relacionadas. Por el conocimiento de la base de datos que tenemos, podemos asumir que si una persona está relacionada con una película es porque ha trabajado en ella. Después, simplemente hay que indicar que queremos retornar el nombre de cada película y el número de personas que han trabajado en la misma.

A diferencia de los sistemas gestores de bases de datos vistos hasta ahora, Neo4j no integra funcionalidades de MapReduce. Esto, que puede resultar extraño en el contexto NoSQL, tiene sentido en Neo4j, ya que es una base de datos que permite realizar operaciones de agregados de forma nativa y no está pensada para ofrecer una alta distribución de datos.



Hasta aquí hemos visto las características básicas de Neo4j y cómo interactuar con esta base de datos. En este apartado vamos a estudiar cómo funciona su mecanismo de distribución y replicación de datos.



La fragmentación de datos en un grafo es una tarea altamente compleja. Algunos de los motivos son:

- La gran mutabilidad de los grafos
- El gran número de relaciones entre nodos
- La dificultad de encontrar subgrafos disjuntos o subgrafos poco interrelacionados,

Estos factores hacen realmente complejo identificar distintos fragmentos del grafo que puedan distribuirse en los distintos servidores de manera que 1) no generen servidores sobrecargados, 2) que el balanceo de carga se pueda ajustar fácilmente a medida que el grafo crezca y 3) que las consultas realizadas no requieran, por norma general, navegación a través de distintos servidores porque los datos de interés están distribuidos.

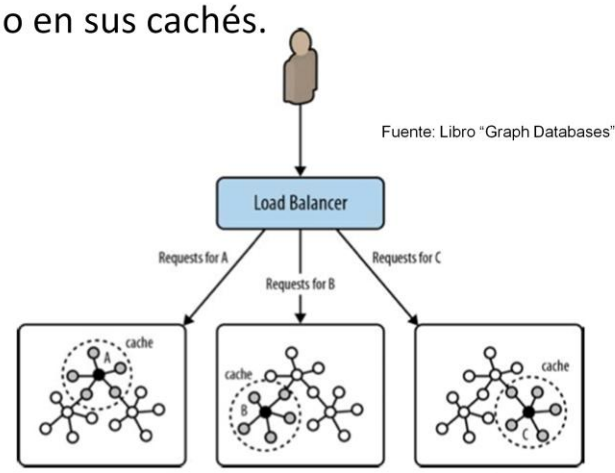
Por todos estos motivos, la fragmentación no es uno de los objetivos de Neo4j. De hecho, en algunos *blogs,* se comenta que realizar *sharding* en una base de datos en grafo es un antipatrón, al igual que el uso de combinaciones (*joins*) son también un antipatrón en las bases de datos NoSQL de agregación (clave-valor y orientadas a documentos). Está claro que la fragmentación en una base de datos en grafo no es tan natural como en las bases de datos NoSQL de agregación, pero creemos que en algunos casos también puede ser viable. En Neo4j el *sharding* se deberá efectuar de forma manual por parte del equipo que diseña la base de datos.

Para proveer fragmentación en una base de datos en grafo debe utilizarse información del dominio representado para identificar qué subgrafos deben distribuirse en cada servidor. Uno de los factores más importantes en este proceso será identificar el conjunto de subgrafos que están menos relacionados, con el objetivo de minimizar las navegaciones (*graph traversals*) que impliquen navegar entre nodos del grafo almacenados en distintos servidores. Esto es importante, porque en el caso de que el diseño de fragmentación esté mal realizado, nos podemos encontrar con casos donde consultar un conjunto de *N* nodos relacionados del grafo implique obtener datos de forma secuencial de *N* servidores distintos, de tal manera que obtengamos un pésimo rendimiento.

No obstante, también es cierto que en algunos casos el *sharding* podría ser útil y viable en una base de datos en grafo. Un ejemplo donde podría ser viable sería para representar la red de carreteras de distintos países. En este caso, podría fragmentarse la red de carreteras en función del país dónde se encuentran. Como las consultas de rutas normalmente se realizan en el contexto de un país, las consultas podrían realizarse localmente en el servidor que contiene los datos, evitando los *graph traversals* entre servidores. Por otro lado, el número de interrelaciones entre los nodos de los grafos ubicados en los diferentes servidores sería mínimo, ya que sólo contendría las carreteras que atraviesan la frontera.

Arquitectura de distribución

- *Graph sharding*:
  - Facilita la escalabilidad horizontal.
  - No se fragmentan los datos en distintos servidores, sino en sus cachés.



EIMT.UOC.EDU

En Neo4j los datos de los grafos se cargan en memoria para hacer la navegación por los mismos. En consecuencia, cuanta más memoria disponga el servidor de la base de datos más veloces serán las operaciones sobre los grafos. En el caso hipotético de que pudiéramos tener todo el grafo en memoria, las lecturas y escrituras sobre la base de datos serian muy eficientes. En condiciones normales esto es inviable, pero Neo4j ofrece una funcionalidad para simular esta situación, y por lo tanto, aumentar la escalabilidad horizontal. Esta funcionalidad se denomina *caché sharding*.

Esta técnica busca solucionar la imposibilidad de mantener todo el grafo en memoria para acelerar su consulta. Esta técnica no fragmenta los datos en distintos servidores sino en las cachés de los distintos servidores. En este caso, los datos del grafo están replicados en diferentes servidores pero las cachés de los servidores contienen subgrafos disjuntos del grafo general. Por lo tanto, en este caso, se puede "simular" una gran memoria que permite acceder a todos los datos del grafo (o a los más comunes). En este caso podríamos ver cada caché como una vista que contiene una parte del grafo, así se podrá acceder a cualquier parte del grafo sólo accediendo a las cachés de los distintos servidores de la base de datos.

Replicación de datos

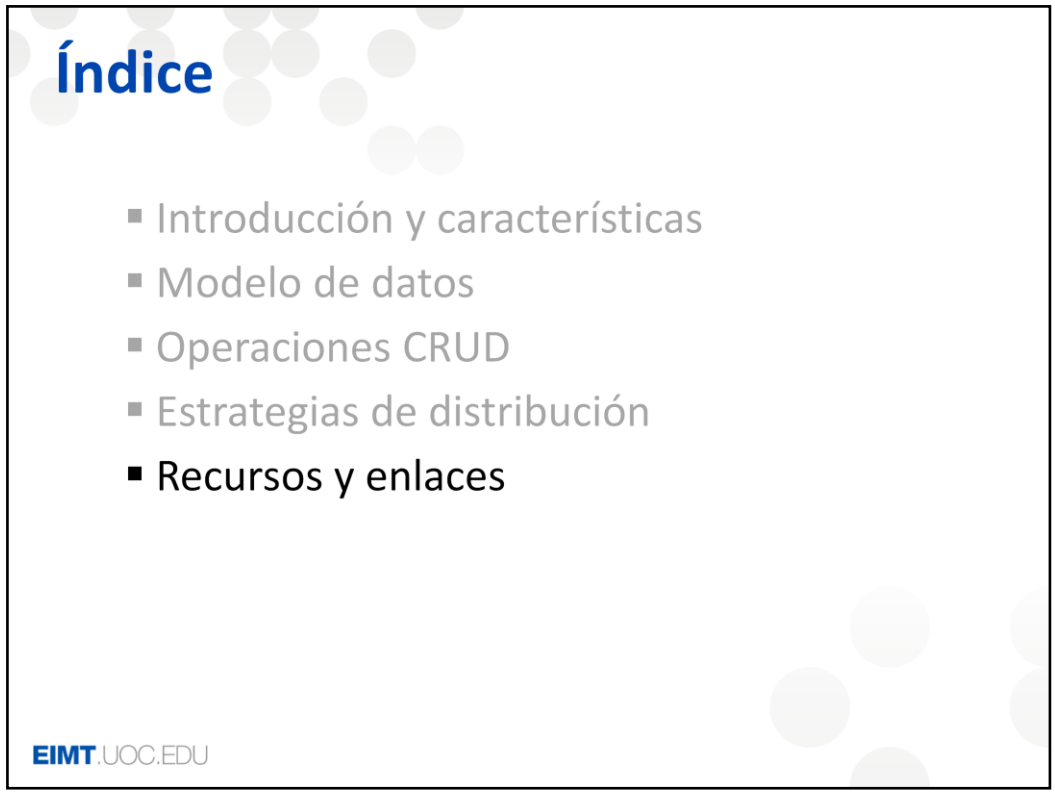
- *Clusters* de alta disponibilidad: conjunto de servidores que almacenan réplicas de unos mismos datos.
- Utilizan una gestión de réplicas de tipo *master-slave* asíncrona:
  - Las operaciones de escritura se realizan en el servidor maestro (*master*):
    - Se puede configurar para propagar los cambios a los servidores secundarios (*slave*) de manera optimista.
    - Extraordinariamente, se puede escribir en servidores secundarios (*slave*).
    - Las operaciones de lectura se pueden realizar en cualquier servidor (*eventual consistency*).
  - Se pueden definir servidores de tipo árbitro

Neo4j permite la replicación de datos para proveer de copias de seguridad y garantizar la recuperación automática en caso de caídas de servidores. En Neo4j, actualmente, esta funcionalidad se incluye en las denominadas características de *high availavility* que únicamente están disponibles en la versión Enterprise Edition. La replicación en Neo4j se gestiona mediante una arquitectura *master-slave* asíncrona parecida a la que hemos visto para MongoDB, donde:

• Existe un solo *master* (o copia primaria).
• Las operaciones de escritura se realizan, por norma general, sobre la copia primaria (hay una excepción que veremos más adelante).
• Las operaciones de lectura se pueden realizar tanto en la copia primaria como en las secundarias (que están en nodos *slave*).
• Los datos (o valores) de las copias (o réplicas) secundarias se sincronizan a partir de las operaciones realizadas en la copia primaria de forma asíncrona.
• En caso de que el nodo que almacena la copia primaria caiga, se promociona una copia secundaria como nueva copia primaria.
• Opcionalmente, pueden añadirse nodos de tipo árbitro a un *cluster*. Estos árbitros permitirán desempatar las votaciones que puedan surgir al escoger una nueva copia primaria.

Las anteriores características hacen que las réplicas permitan a Neo4j incrementar la disponibilidad de los datos mientras mantienen consistencia final en el tiempo, esto es debido a que la propagación de los cambios se realizan de forma asíncrona y que las operaciones de consulta se pueden realizar también en las copias secundarias. Por lo tanto, entre que se realiza un cambio en la copia primaria, hasta que se propaga a las copias secundarias, éstas contendrán valores distintos. No obstante, es posible configurar Neo4j para que los cambios en la copia primaria se propaguen de forma automática a las secundarias antes de terminar la transacción. No obstante, esta propagación se realiza de forma optimista, es decir, se acepta la confirmación de la transacción aunque no se haya podido propagar el valor a alguna de las réplicas. Por ese motivo, aunque esta operación aumenta la consistencia de los datos en el caso general, el sistema continua proporcionando *eventual consistency* en el caso peor.

Otra peculiaridad en la gestión de réplicas de Neo4j es que se permite que los nodos que almacenan copias secundarias (nodos *slave*) acepten operaciones de escritura. Cuando esto pasa, la escritura se realiza en el nodo *master* (que almacena la copia primaria) y en el nodo *slave* a la vez (en una sola transacción), garantizando que la copia primaria tiene el valor actualizado en todo momento. Antes de ejecutar una operación de este tipo, el sistema deberá comprobar que los datos afectados en los nodos *master* y *slave* son los mismos. Si no es así, se descartará la operación. Este modo de operación aumenta el tiempo de ejecución de la operación (y en consecuencia, disminuye el tiempo de respuesta). Además bloquea los datos de dos servidores, reduciendo la disponibilidad del sistema. No obstante, puede ser aconsejable realizar este tipo de operación cuando se quiera garantizar que un nodo *slave* tenga los datos actualizados en todo momento.



En esta presentación hemos examinado las principales características de Neo4j y de su modelo de datos. También hemos visto cómo utilizar Cypher para acceder y manipular los datos almacenados en Neo4j. Finalmente, hemos hablado de distribución y replicación de datos en Neo4j.

La presentación tiene por objetivo ser eminentemente introductoria. Para cualquier desarrollo real en Neo4j aconsejamos consultar su documentación para obtener información actualizada (y detallada) de sus funcionalidades, operaciones y arquitectura.

Ya para finalizar presentaremos un conjunto de enlaces y referencias de interés.

Referencias

Página oficial de Neo4j: http://www.Neo4j.org/

Documentación oficial: http://docs.Neo4j.org/

Reference Card de Cypher: http://docs.Neo4j.org/refcard/2.1.1/

Ian Robinson, Jim Webber & Emli Eifrem (2013). *Graph Databases. O'Reilly.*

E. Redmond, J. Wilson (2012). *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.* Pragmatic Bookshelf.

P.J. Sadalage & M. Fowler. (2013). *NoSQL Distilled. A brief Guide to the Emerging World of Polyglot Persistence*, Pearson Education.

Esperamos que hayáis disfrutado y aprendido con este vídeo. A continuación encontraréis algunas referencias que os permitirán profundizar más en los conceptos que hemos tratado.

Que tengáis un buen día.