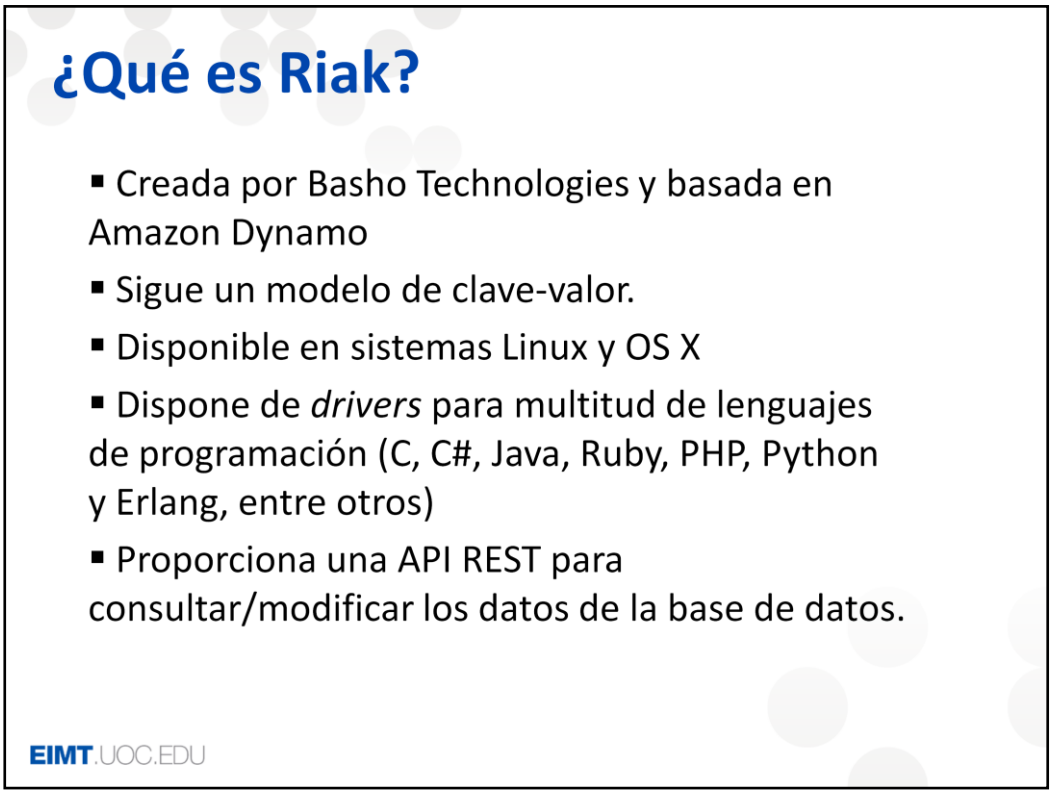
Bienvenidos. En esta presentación vamos a explicar brevemente una de las bases de datos NoSQL clave-valor más conocidas a día de hoy: Riak.

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 Riak y explicaremos sus principales características. Posteriormente veremos el modelo de datos que utiliza (el clave-valor con algún elemento añadido). En la tercera parte de la presentación, explicaremos cómo interactuar con Riak 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 Riak.

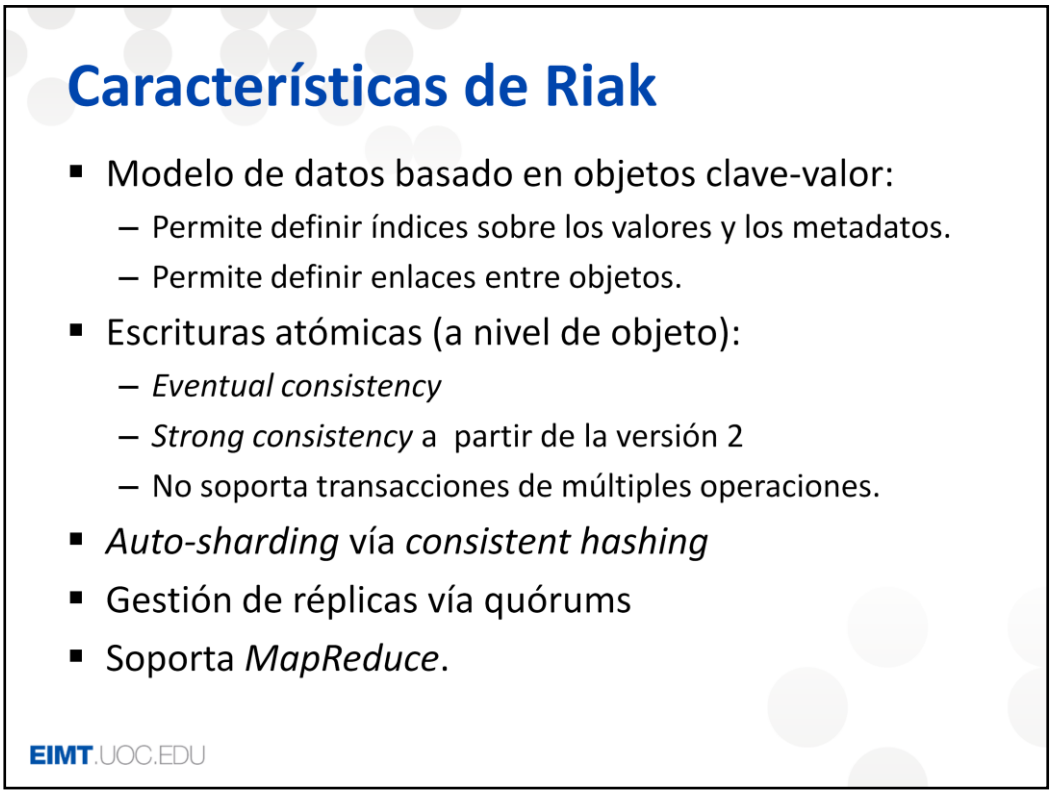


Riak es una base de datos clave-valor creada por Basho Technologies que se basa en el modelo propuesto por Amazon Dynamo. Actualmente es una de las más populares de su categoría según el *ranking* de db-engines.com (http://db-engines.com/en/ranking). De hecho, en 2014, es la tercera base de datos clave-valor más popular y la número 30 de 221 en el *ranking* general de bases de datos.

La unidad básica de procesamiento de Riak son los objetos, entendiendo un objeto como un par formado por los elementos <clave, valor>. Los valores son "opacos" para la base de datos. Es decir, el gestor de la base de datos no tiene por qué saber qué estructura contiene el valor de un objeto ni ser capaz de entender sus datos. Podríamos ver los valores de los objetos como cajas negras, aunque como veremos, hay algunas excepciones. Como base de datos de tipo *schemaless* no requiere crear a priori el esquema que siguen los datos que se van a almacenar en los valores de los objetos. Su gestión irá a cargo de los programas que acceden a la base de datos (y por lo tanto de las personas que crean dichos programas).

Es una base de datos escrita en Erlang, C, C++ y JavaScript que está disponible en sistemas operativos Linux y OS X.

Riak proporciona una API REST que permite consultar la base de datos realizando peticiones HTTP. REST permite mapear recursos a URLs e interactuar con ellos utilizando peticiones HTTP del tipo POST, GET, PUT y DELETE. Asimismo también proporciona *drivers* para una gran cantidad de lenguajes de programación, facilitando su uso sea cual sea el lenguaje que utilice el programa cliente que desea acceder a la base de datos.



Tal y como ya se ha comentado, la unidad básica de almacenamiento son los objetos, formados por los pares <clave, valor>. En el valor de un objeto se pueden almacenar datos en distintos formatos y que sigan distintas estructuras (*schemaless*). El modelo de datos utilizado por Riak extiende el modelo básico clave-valor para permitir definir metadatos y enlaces entre objetos. Además, Riak, como veremos más adelante, proporciona distintos tipos de índices, que permiten, por un lado, consultar la información de los valores de un objeto cuando éstos tienen un formato textual (XML o JSON), y por otro, consultar información sobre los metadatos asociados a un objeto (es decir al par <clave, valor>).

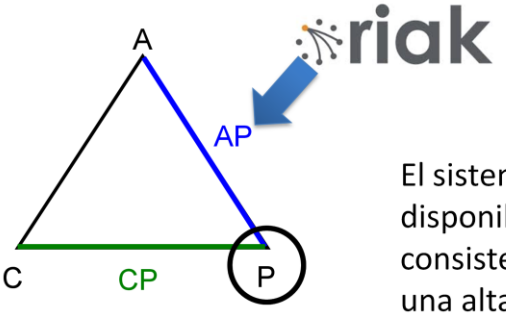
Las escrituras se realizan de forma atómica en la base de datos para cada objeto. No existen transacciones (en el sentido que este término tiene en bases de datos relacionales), por lo tanto, es imposible definir una transacción que incluya diferentes operaciones. Según la documentación de Riak, la replicación sigue una estrategia *masterless*. Es decir, no utiliza *master-slave* sino que utiliza quórums para la gestión de réplicas y *consistent hashing* para distribuir los objetos. La consistencia puede ser fuerte (*strong consistency*) o final en el tiempo (*eventual consistency*). Por defecto la consistencia es final en el tiempo, pero a partir de la versión 2.0 se permite aplicar consistencia fuerte en diferentes elementos de la base de datos identificados a priori por el diseñador de la base de datos.

Recordemos que la *strong consistency* garantiza que las aplicaciones (y en consecuencia los usuarios) tengan una visión consistente de la base de datos (bajo su punto de vista todas las réplicas de un mismo objeto tienen los mismos valores, aunque físicamente, en la base de datos, esto no sea así). Por su parte cuando se proporciona *eventual consistency* la falta de concordancia en los valores de unas mismas réplicas de un objeto puede ser visible para las aplicaciones y usuarios (es decir, las inconsistencias pueden ser visibles).

En Riak la base de datos se encarga de distribuir los objetos en distintos nodos de forma automática en función de su clave basándose en *consistent hashing*. Hay cierta controversia respecto a si Riak realiza *autosharding* o no. Depende de cómo se mire, podríamos entender que Riak ofrece *autosharding* si vemos los *shards* o fragmentos como objetos individuales. No obstante, si concibiéramos el *shard* como un conjunto de objetos, entonces no podríamos decir que Riak realice *autosharding*.

Riak integra el *framework* MapReduce para realizar operaciones sobre los datos de la base de datos de forma más eficiente.

Enfocado a proporcionar una alta tolerancia a fallos



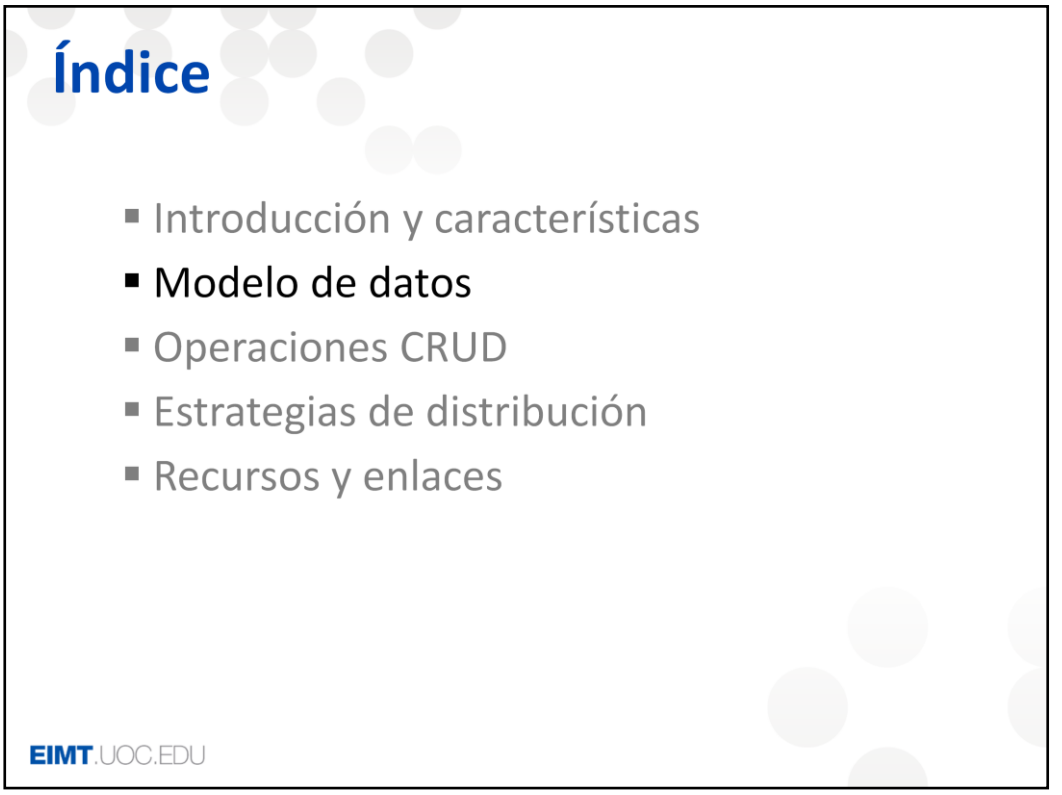
riak

El sistema apuesta por la disponibilidad a costa de la consistencia, proporcionando una alta tolerancia a fallos pero permitiendo que algunas lecturas puedan devolver datos obsoletos.

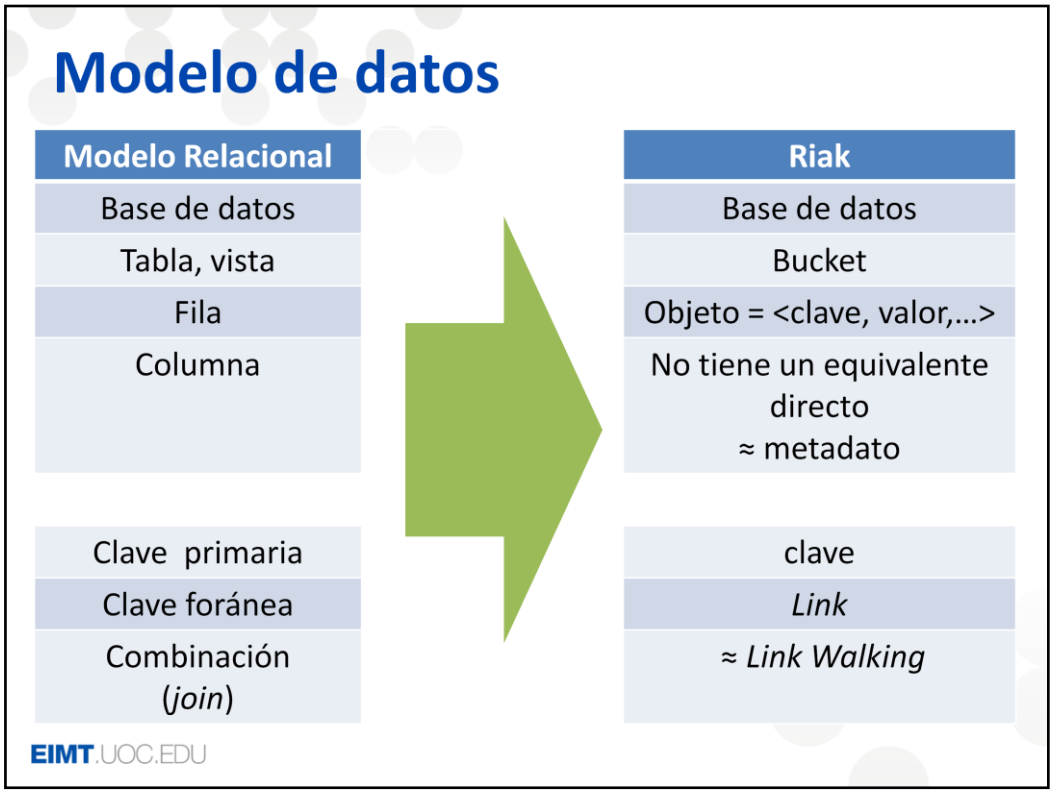
EIMT.UOC.EDU

Respecto a las tres propiedades del teorema CAP (consistencia, disponibilidad y tolerancia a particiones), Riak está enfocado a garantizar una alta tolerancia a fallos (AP). Esto quiere decir que para asegurar una alta disponibilidad de la base de datos, debe relajar la consistencia, permitiendo en algunas ocasiones que los usuarios/programas puedan leer valores obsoletos.

No obstante Riak, desde su versión 2.0, permite reforzar el nivel de consistencia (*strong consistency*) en detrimento de su disponibilidad. En estos casos, podríamos decir que el comportamiento de la base de datos estaría más cercano a un modelo CP.



Una vez introducida la base de datos Riak, 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 Riak utilizaremos el modelo relacional como base. En Riak los datos se almacenan en bases de datos y, dentro de éstas, se organizan en *buckets*. Los *buckets* son el equivalente a las tablas del modelo relacional, y se utilizan para agrupar elementos con semántica o características comunes como, por ejemplo, los pedidos de venta. Un *bucket* contiene un conjunto de objetos, donde cada objeto representa un individuo del *bucket*. Un objeto se compone básicamente de dos elementos: la clave que identifica el objeto y su valor. Siguiendo con nuestro ejemplo, cada objeto equivaldría a un pedido de venta concreto.

Podemos ver la clave de un objeto como un valor simple que permite identificar el objeto en el contexto de un *bucket* (sería el equivalente a la clave primaria en el modelo relacional). Su valor puede ser informado por el usuario o, en caso contrario, será asignado por el gestor de la base de datos de forma automática.

Una base de datos clave-valor pura no debería ser capaz de acceder al contenido de los valores de los objetos. No obstante, con el tiempo, Riak ha empezado a proveer funcionalidades adicionales a costa de acceder a su contenido. Un ejemplo de ello es la funcionalidad que pemite indexar los valores de los objetos cuando estos son documentos JSON.

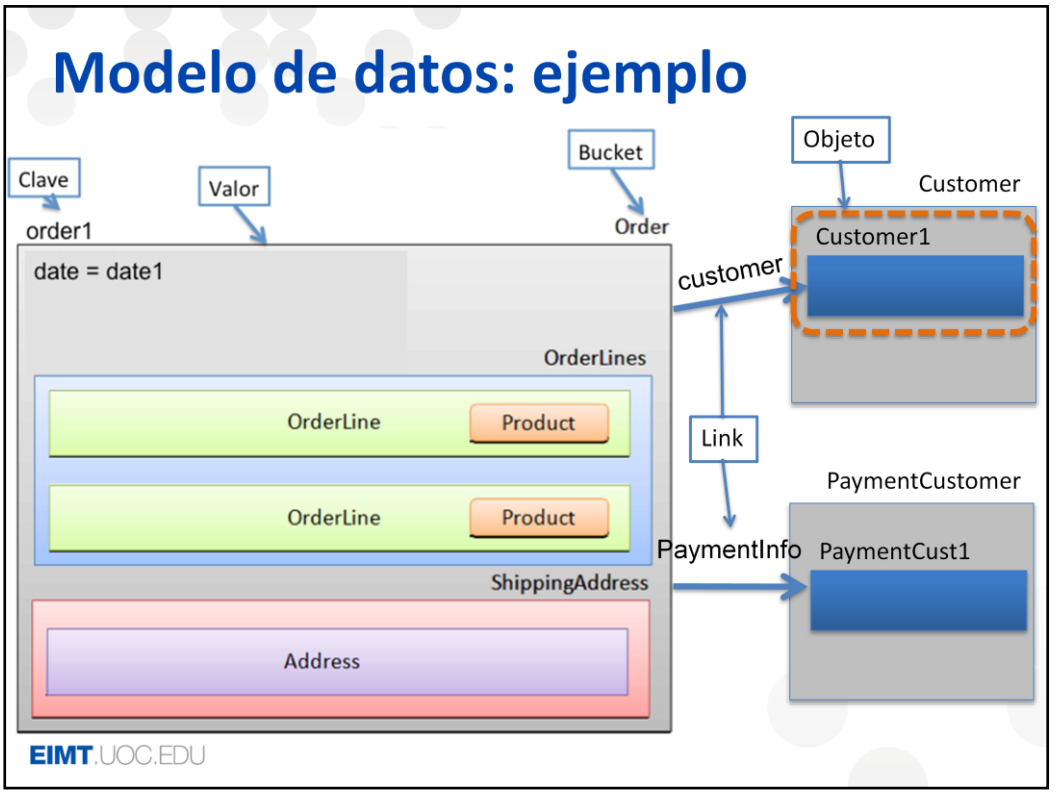
Además, en sus últimas versiones, Riak gestiona directamente subconjuntos del valor de los objetos cuando estos subconjuntos se definen mediante un conjunto de tipos de datos (*datatypes*) predefinidos. Estos tipos de datos predefinidos, a fecha 2014, son los siguientes: *flags* (equivalentes al tipo booleano), los registros (equivalente a un *string* binario), contadores (valores numéricos), conjuntos (que son colecciones de valores binarios) y mapas (que son equivalentes a un registro que agrupa un conjunto de variables de los 5 tipos que hemos comentado anteriormente). Los elementos que utilizan estos tipos de datos se pueden consultar y modificar directamente sin tener que consultar o modificar el valor al que pertenecen, permitiendo transferir parte de la gestión de datos de los programas al gestor de la base de datos.

Una vez presentada la semántica de la clave y del valor de un objeto, es importante mencionar que en Riak los objetos pueden contener más elementos: los metadatos y otros elementos para la gestión de los datos, como son la referencia al *bucket* del objeto y un vector de tiempo (*vector clock*) utilizado para gestionar los conflictos.

Los metadatos son atributos que pueden definirse sobre un objeto pero que no se almacenan dentro de su valor. Normalmente se usan para indicar datos sobre los datos que se almacenan (por ejemplo quién creó un pedido de venta), para informar de datos que pocos objetos del *bucket* comparten, o para el uso de funcionalidades añadidas, como por ejemplo, la creación de índices secundarios. Si bien los metadatos podrían verse como un tipo de columna del modelo relacional, no podemos decir que las columnas del modelo relacional tengan una contrapartida clara en Riak.

Riak tiene un elemento que podría verse como el equivalente a las claves foráneas del modelo relacional: el *link*. Un *link* es un metadato que establece una relación unidireccional entre dos objetos de Riak. El uso de *links* permite definir relaciones entre los datos. Por ejemplo, se podría utilizar un *link* para indicar qué cliente realizó un pedido. Riak no limita el número de *links* que se pueden definir por cada objeto. Dado un objeto, sus *links* se pueden utilizar para obtener los objetos con los que se relaciona. Esta operación se conoce como *link-walking* en Riak y viene a ser la operación de *join* de bases de datos relacionales.

Vamos a ver con un ejemplo los distintos elementos de Riak que acabamos de describir.



En esta transparencia podemos ver un ejemplo que representa un pedido de venta y los elementos relacionados. Los diferentes pedidos se almacenarían en un *bucket* denominado *Order*. Cada pedido estará representado por un objeto que contiene una clave (*order1* en el ejemplo) y un valor que contiene información sobre el pedido. La manera en que se distribuyen los datos dentro del valor del objeto en principio debería ser opaco para el gestor de la base de datos (a no ser que se utilicen tipos de datos – en este caso se podría utilizar un tipo de dato contador para la fecha y dos mapas uno con vectores para representar las líneas de pedido y uno con un conjunto de registros para representar la dirección de envío). Nótese también que se han definido dos *links* para relacionar el pedido con el cliente que lo ha realizado y con los datos de pago del pedido.

El ejemplo que utilizamos aquí es el mismo que usamos para ejemplificar los otros tipos de bases de datos que hemos visto. El objetivo es entender cómo se almacenan los mismos datos en distintos modelos de datos. No obstante, debido a las características del modelo clave-valor, es posible que este modelo no sea el más adecuado para almacenar los datos del ejemplo.

Índices

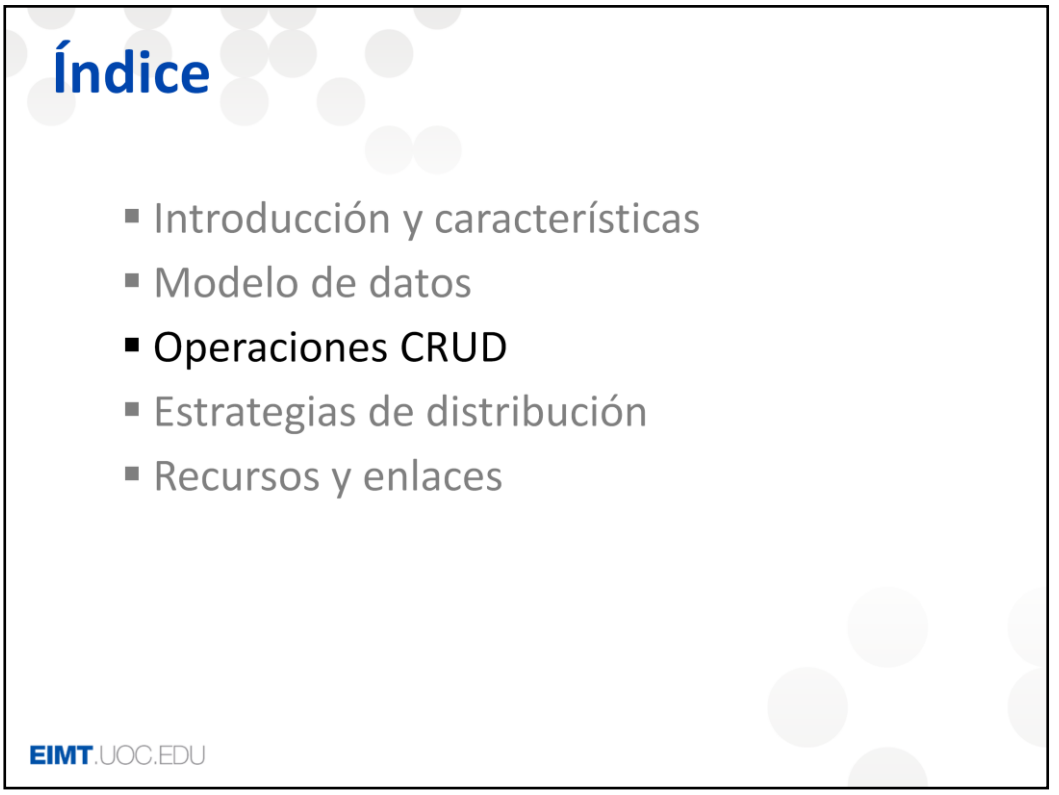
- Riak permite definir índices invertidos sobre los valores de los objetos: sólo disponible para valores en formato textual, XML, JSON y Erlang.
- Riak permite definir índices secundarios:
  - Los datos del valor son "opacos" y no siguen un esquema.
  - Los índices se definen sobre los metadatos.
  - Los valores indexados pueden no tener correspondencia con los datos del objeto.

EIMT.UOC.EDU

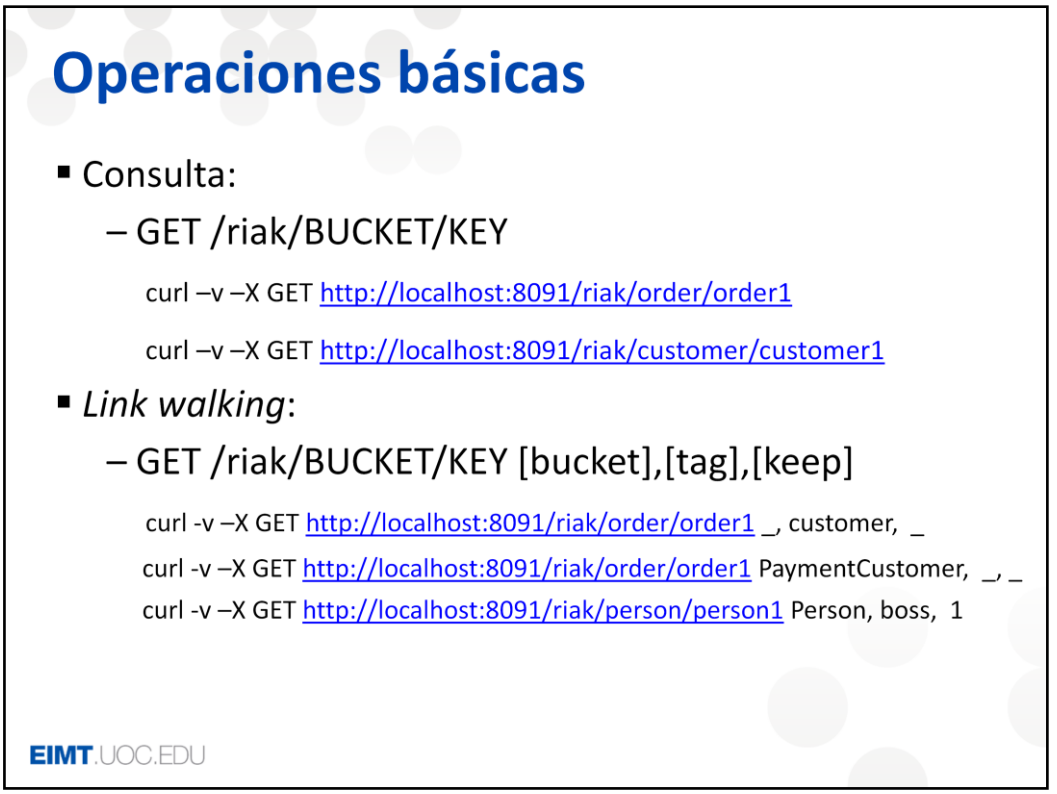
En Riak las consultas se realizan por defecto a partir de las claves de los objetos. No obstante, Riak ofrece diversos tipos de índices que permiten realizar búsquedas por contenido. Básicamente Riak permite dos tipos de índices: índices invertidos para indexar los valores de los objetos que se almacenan en un formato textual e índices secundarios sobre metadatos. Vamos a ver en más detalle estos dos tipos de índices.

Riak Search es una extensión que permite realizar búsquedas de texto completo sobre los valores y las claves de los objetos. Esta funcionalidad permite indexar las claves, pero también los valores que el gestor de la base de datos sea capaz de interpretar. Este tipo de índice se define a nivel de *bucket* y debe activarse y configurarse explícitamente. Sólo permite indexar datos textuales y ficheros XML, JSON y Erlang. Riak utiliza el tipo de valor (metadato *mime type*) para identificar si un valor se debe tener en cuenta en el índice o no.

Aparte de este sistema de indexación y búsqueda, Riak permite definir índices secundarios. Estos índices permiten indexar la información de ciertos datos sobre los objetos. No obstante, cómo los datos del valor de un objeto no son generalmente accesibles para el gestor de la base de datos, este tipo de índice se define sobre los metadatos del objeto. Cuando queramos utilizar este sistema para indexar un determinado dato para un objeto deberemos añadir dicho dato como metadato del objeto con un nombre del tipo "*x-riak-index-<name>-bin*" o "*x-riak-index-<name>-int*", donde *name* indica el nombre del metadato y el sufijo cambia en función de si el índice se realiza sobre un valor numérico o binario. Como este tipo de índices se calcula directamente sobre los metadatos, se podría usar para indexar datos que no estén almacenados en el valor del objeto. En el caso de los pedidos de venta podríamos indexar los siguientes datos del pedido *order1*: la fecha del pedido (indicando un metadato con nombre *x-riak-index-date-int* y valor *date1*) y si el pedido está pagado o no (mediante el metadato con nombre *x-riak-index-isPayed-int* y valor *0,* suponiendo que no está pagado). Como se puede ver en la transparencia anterior, la información de si el pedido está pagado o no, no forma parte de los datos del objeto. En consecuencia, los metadatos añadidos permitirían enriquecer la semántica del pedido, a la vez que podemos permitir una búsqueda más efectiva a partir del hecho de si un pedido está pagado o no.



Hasta aquí la introducción a Riak 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).



Las operaciones de escritura y lectura en Riak se ejecutan siempre en el ámbito de un *bucket* y permiten crear, modificar, borrar y consultar objetos. Recordad que, por norma general, las operaciones en Riak se realizan siempre a nivel de objeto (es decir, un par <clave, valor>).

En esta transparencia podemos ver una simplificación de la sintaxis REST propuesta. REST permite mapear recursos a URLs e interactuar con ellas utilizando peticiones HTTP (POST –Create–, GET –Read– , PUT –Create/Update– y DELETE –Delete–). Las llamadas a las API en algunos lenguajes pueden diferir ligeramente de esta sintaxis. Todo elemento de Riak podrá ser identificado mediante una URL. Dado un objeto que pertenece al *bucket* b con clave c, su URL sería /riak/b/c, donde "riak" es la URL del servidor donde se encuentra la base de datos de Riak ejecutándose.

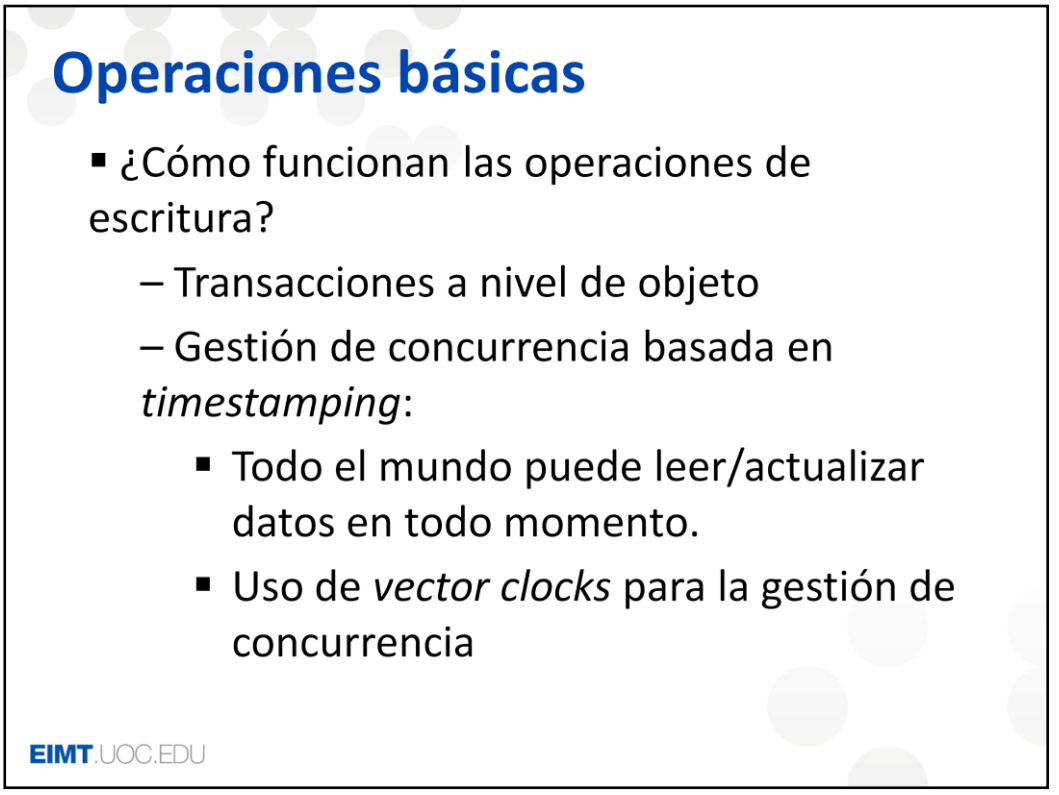
Empezaremos hablando de las operaciones de consulta (o lectura) de objetos. Podemos hacer una consulta de un objeto en Riak indicando la operación HTTP GET y pasando como parámetro la URL del objeto a consultar. En la transparencia podemos ver dos ejemplos, donde se consulta el pedido de venta con clave *order1* y el cliente con clave *customer1*.

El resultado de la consulta retornará el valor requerido y en la cabecera de la respuesta HTTP existirá un código que indicará el resultado de la operación en la base de datos: 200 – OK, 404 – *Not found*, etc.

A continuación podemos ver cómo realizar un *link walking* en Riak o, lo que es lo mismo, consultar las relaciones entre objetos. Para consultar las relaciones de un objeto en Riak (que llamaremos objeto origen) debemos preguntar por dicho objeto (tal y como hemos visto anteriormente) y luego indicar qué tipo de elementos relacionados esperamos obtener. Para ello tenemos tres parámetros: *bucket*, *tag* y *keep*. El parámetro *bucket* permite identificar el tipo esperado de los objetos relacionados que debemos obtener en la consulta (es decir de qué *bucket* son los objetos relacionados que queremos obtener). El parámetro *tag* permite definir qué relación queremos utilizar en la consulta. De indicar este parámetro, obtendríamos todos aquellos elementos relacionados con el objeto mediante la relación *tag*. Finalmente, el parámetro *keep* permite indicar si queremos obtener los elementos directamente relacionados con el elemento origen o si queremos obtener también los elementos indirectamente relacionados con él. En el caso de no querer informar uno de los tres parámetros podemos utilizar el elemento "_", que indica que cualquier elemento es válido.

En la transparencia se muestran 3 ejemplos. El primero obtendría todos los objetos que están relacionados con el objeto *order1* mediante la relación *customer* (*customer1* según el ejemplo). El segundo obtendría todos los objetos que pertenezcan al *bucket PaymentCustomer* relacionados con *order1* (*PaymentCust1* según el ejemplo presentado anteriormente). El tercer ejemplo devolvería todos los jefes de la persona *person1*, es decir, su jefe directo y los jefes de éste.

Antes de continuar con las siguientes operaciones creemos que es interesante hablar de cómo gestiona Riak las transacciones y la concurrencia.



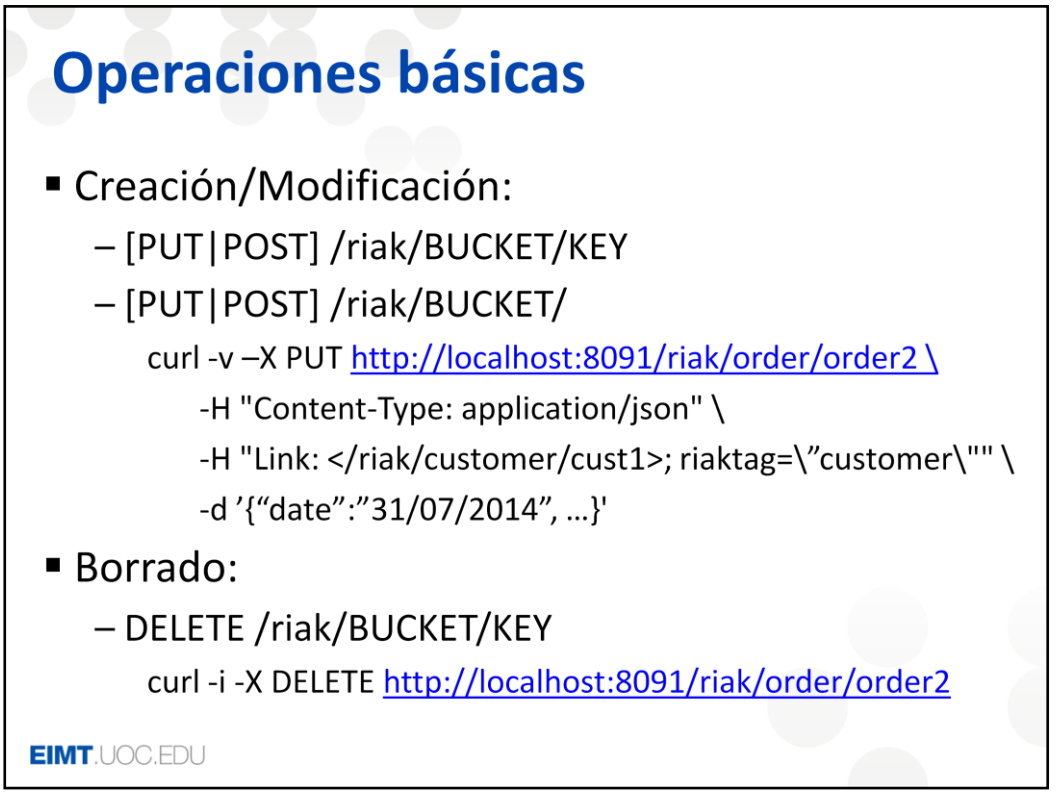
Las transacciones en Riak incorporan una única operación (y esta operación actúa sobre un objeto de un mismo *bucket*). Si queremos crear una transacción que incorpore múltiples operaciones deberemos gestionarlas desde nuestros programas.

Respecto a la gestión de concurrencia, Riak utiliza una variante del sistema pesimista de *timestamping* (marcas de tiempo en castellano) gestionado mediante *vector clocks*. En dicho sistema, no se realizan bloqueos y se permite que todo el mundo realice actualizaciones (y lecturas) sobre la base de datos en todo momento. Es en las operaciones de lectura cuando se comprueba que las versiones de los datos son correctas. De no ser así, se realizan las acciones correctoras pertinentes.

Los *vector clock* son elementos asignados a los valores de los objetos y tienen el objetivo de determinar el orden causal (que no el cronológico) de las diferentes operaciones realizadas sobre un objeto. Estas relaciones causales permitirán definir un orden parcial de los eventos que se han realizado sobre cada valor de la base de datos y, por lo tanto, permitirán identificar qué valores se han actualizado a partir de otros y por lo tanto como ha evolucionado el valor de un objeto en el tiempo.

El *vector clock* de un valor evoluciona con cada operación de modificación realizada sobre el mismo. Dados dos *vector clocks* pertenecientes a dos versiones de un mismo objeto, Riak es capaz de obtener la siguiente información a partir de su comparación: 1) si un valor se ha obtenido mediante la modificación del otro valor (uno es descendiente del otro), 2) si el valor de los dos objetos se ha modificado a partir del mismo valor (es decir, si ambos han utilizado el mismo valor origen), o 3) los valores no están relacionados (es decir, uno no es la modificación del otro y no tienen ninguna versión anterior en común). Riak usa esta información para reparar automáticamente los conflictos entre los datos o para proporcionar información al cliente (usuario o programa) para que sea él quien reconcilie los datos. Por ejemplo, en el caso de que Riak detecte que hay dos posibles valores (*v1* y *v2*) para un mismo objeto, si el análisis de sus *vector clocks* demuestra que uno de los valores (supongamos que es *v2*) se ha realizado a partir del valor del otro (v1), entonces puede concluir que *v2* es posterior a *v1* y por lo tanto consolidar *v2* como valor válido.

En caso de que Riak no pueda resolver un conflicto en un objeto (por ejemplo, en el caso de que dos escrituras se hayan realizado a partir del mismo valor) se pueden crear diferentes versiones del objeto con los posibles valores, denominados *siblings* en la nomenclatura de Riak. Se pueden consolidar distintas versiones de un objeto (*siblings*) determinando cuál es la versión válida y cuál se debe descartar. Esto pueden hacerlo los clientes (aplicaciones o usuarios) o el servidor de la base de datos de forma automática. La manera en que se resuelven los conflictos puede configurarse para cada *bucket* y su tratamiento por defecto puede variar en función de la versión de Riak con la que trabajemos.



Dar de alta o modificar objetos en Riak se realiza mediante una operación HTTP de PUT o POST e indicando la URL de la clave del objeto y su valor. Cuando la clave no exista, Riak interpretará que queremos realizar una inserción en la base de datos. Cuando la clave exista, la operación ejecutada será una actualización.

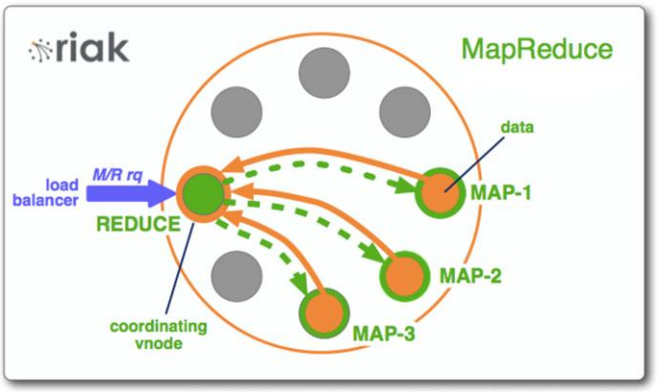
También es posible añadir el valor sin indicar una clave, es decir, facilitando sólo la URL del *bucket* donde debe añadirse el objeto. En este caso, Riak creará el objeto en el *bucket* indicado y asignará una clave automáticamente al mismo. En estas operaciones se puede utilizar el parámetro –H para indicar metadatos o *links* del objeto.

En el ejemplo de la transparencia podemos ver cómo se hace una llamada a la operación PUT para crear, en el *bucket* llamado *order,* un nuevo objeto con clave *order2*. El valor del objeto es un documento JSON que describe el pedido de venta. Asimismo, podemos ver cómo se añade una relación entre el objeto *order2* y el objeto identificado con la clave *cust1* perteneciente al *bucket customer*. El nombre de la relación es *customer* y en este caso sirve para indicar el cliente que efectúa el pedido.

Para borrar objetos se utiliza la operación DELETE indicando la clave del objeto a eliminar. En la transparencia podemos ver un ejemplo donde se eliminaría el objeto identificado con la clave *order2* que se acaba de crear anteriormente.

Otras operaciones

- Gestión de *buckets*
- Búsquedas por índice
- MapReduce



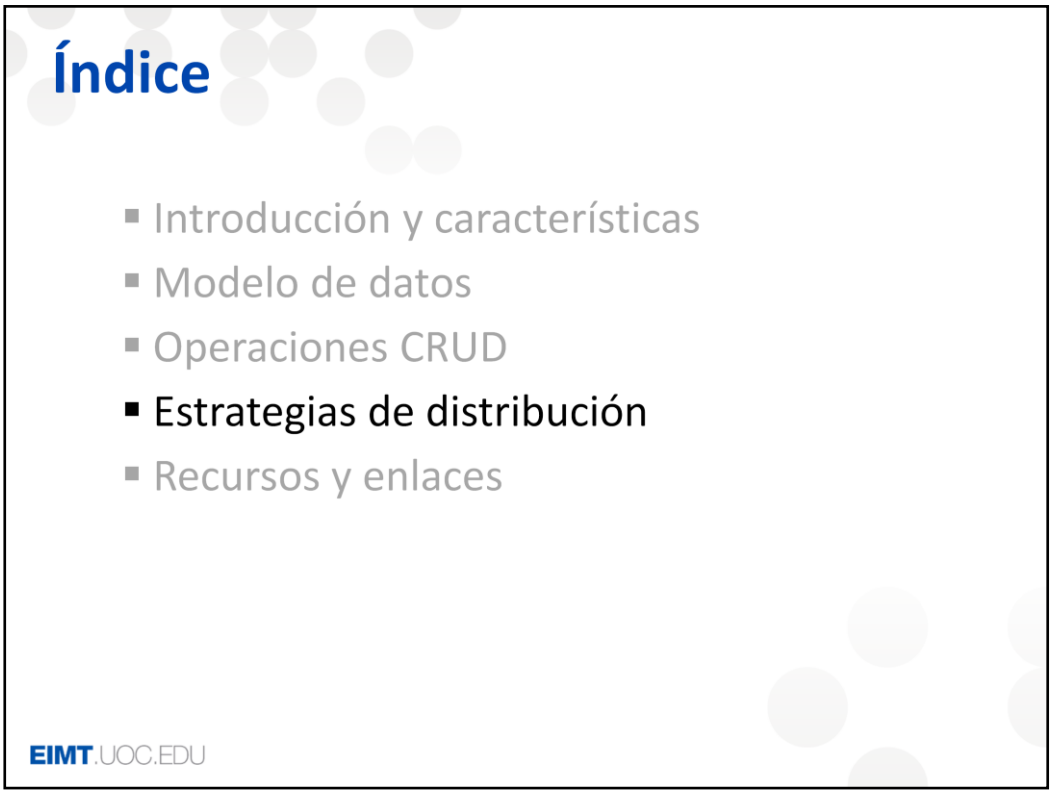
Fuente: http://docs.basho.com/riak/latest/dev/using/mapreduce/

EIMT.UOC.EDU

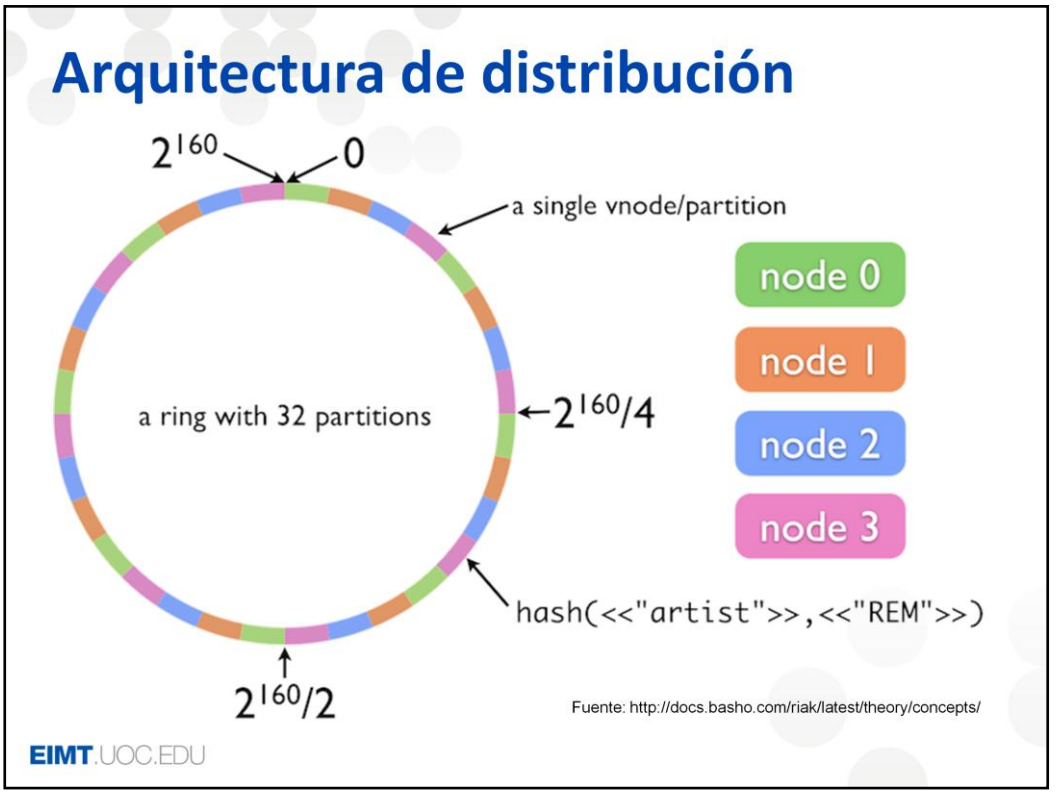
Además de las operaciones básicas que acabamos de ver, hay otras operaciones que se pueden ejecutar en Riak. Algunas de ellas son las relativas a la creación y gestión de *buckets*, al uso de índices para realizar búsquedas por contenido y a la definición de tareas MapReduce.

En Riak las tareas MapReduce no están pensadas para para realizar consultas de agregados en tiempo real, sino para ejecutar operaciones en modo *batch*. Las consultas MapReduce en Riak están formadas por 2 elementos: un conjunto de entradas y un conjunto de operaciones *map* y *reduce* a realizar. El conjunto de entradas es una lista de pares clave/*bucket* que indican los objetos que participarán en el proceso MapReduce.

En la figura podemos ver cómo funciona el proceso MapReduce en Riak. El cliente hace una petición MapReduce que llega a un nodo de la base de datos. El nodo que recibe la petición será el encargado de coordinar todo el proceso de MapReduce. Este nodo utiliza las claves y los *buckets* de entrada para determinar a qué nodos debe enviar las funciones *map* para que sean ejecutadas localmente (normalmente se trata del nodo que almacena los datos a procesar). Los nodos elegidos ejecutarán la función *map* sobre los objetos identificados con la clave recibida y retornará sus resultados al nodo origen. El nodo origen concatenará los resultados obtenidos en una lista y ejecutará la función *reduce* (en el caso de haberse definido) sobre dicha lista.



Hasta aquí hemos visto las características y operaciones de Riak. En este apartado vamos a estudiar cómo funciona su mecanismo de distribución y replicación de datos.



Riak está diseñada para hacer eficiente la distribución de datos a gran escala. Normalmente los objetos en Riak se distribuyen entre distintos servidores interconectados. Estos servidores se denominan *nodos* según la nomenclatura de Riak.

La interfaz de Riak trabaja con *buckets* y claves. Internamente, y para cada objeto, Riak calcula una función de *hash* consistente (*consistent hashing*) sobre la pareja <clave, *bucket*>. La función de *hash* devuelve como resultado un valor binario de 160 bits que se utilizará para determinar dónde se almacenarán los objetos. El espacio de los posibles valores de la función de *hash* puede verse como un espacio circular, donde el valor siguiente al último valor posible es el primer valor posible. Este espacio de valores circular se denomina anillo de Riak.

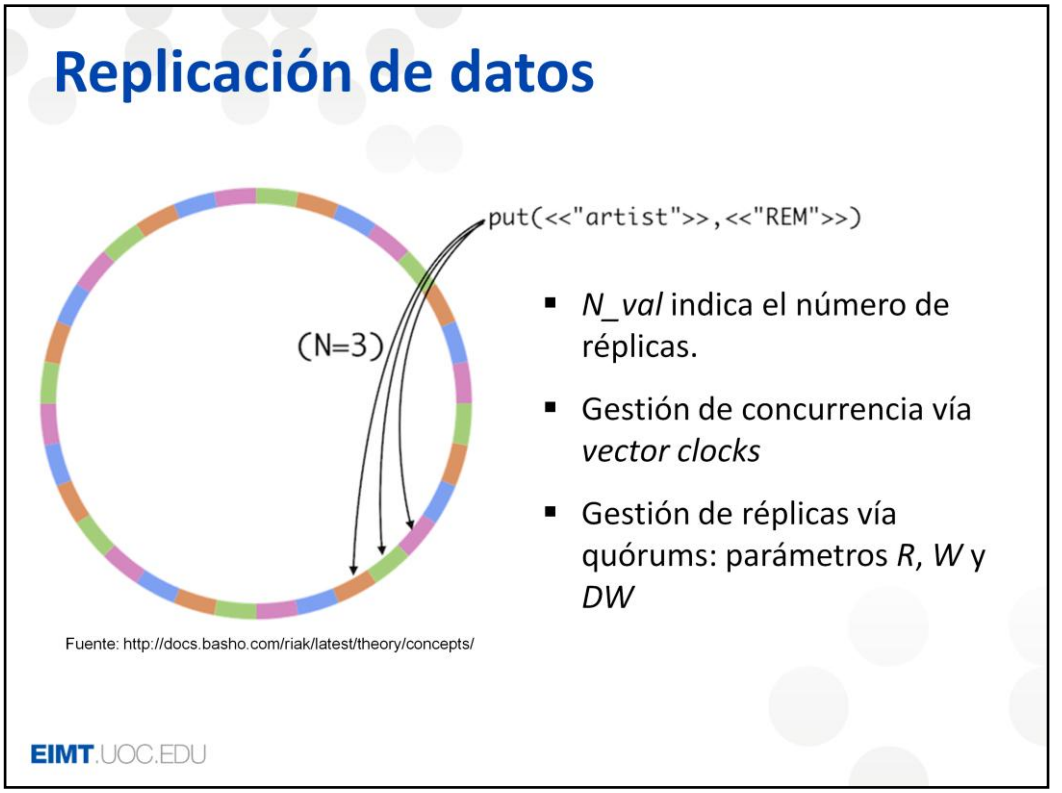
Tal y como podemos ver, el anillo de Riak comprende el espacio de valores que va desde 0 a 2 elevado a 160. Este espacio contiene todos los posibles valores que devuelve la función de *hash* que se aplica sobre la pareja <clave, *bucket*>. El anillo se divide en distintas particiones de igual tamaño. Cada una de estas particiones se gestiona desde un nodo virtual (o *vnode*). En la figura podemos ver cómo se ha distribuido el anillo en 32 particiones que se han pintado de distintos colores para diferenciarlas. Cada *vnode* gestiona una partición y se hará cargo de almacenar los datos de los objetos cuyos valores aplicados a la función de *hash* den como resultado un valor dentro de la partición que gestiona el *vnode*.

El número de *vnodes* es independiente del número de servidores, que son denominados nodos (o *nodes*) en la nomenclatura de Riak. Los distintos nodos de la base de datos se encargarán de repartirse los *vnodes* de forma equitativa. Por lo tanto, el número de *vnodes* por cada nodo será igual al número de particiones dividido por el número de nodos. En el caso de ejemplo, existen 4 nodos. En consecuencia, el número de *vnodes* (o particiones) por nodo son 8. En el ejemplo podemos ver como la distribución de particiones por cada nodo se realiza de forma intercalada, de forma que el primer nodo se encargará de las particiones número 1, 5, 9, 13, 17, 21, 25 y 29, el segundo nodo se encargará de las particiones 2, 6, 10, 14, 18, 22, 26 y 20, y así sucesivamente.

El uso del anillo de Riak y de la función de *consistent hashing* para distribuir los objetos en el mismo hace que se puedan añadir o borrar nodos de forma dinámica sin una grave repercusión en el rendimiento del sistema.

Tal y como podemos ver, el tipo de distribución que realiza Riak permite distribuir los objetos de forma equitativa entre las distintas particiones (asumimos que la función de *hash* distribuye uniformemente los objetos) y la distribución de dichas particiones en distintos nodos permite distribuir los objetos de forma equitativa entre los nodos disponibles.

Una vez hemos visto cómo funciona la distribución de datos en Riak, vamos a ver como se realiza su replicación.



Riak maximiza la disponibilidad del sistema. Por lo tanto, debe tener en cuenta aspectos de replicación. La replicación de los objetos se realiza de acuerdo al valor de la variable N_val. El valor de esta variable es un número natural que indica el número de réplicas que se deben realizar de los objetos. Su valor puede ser configurado por el usuario a nivel de *bucket*. Por defecto, en Riak, el valor de N_val es 3, es decir, la base de datos guarda tres copias de los objetos. El número de réplicas debe ser mayor que 0 y menor o igual que el número de nodos. Por defecto, las réplicas en Riak se almacenan en *vnodes* consecutivos, tal y como podemos ver en la figura. Según la documentación, no hay ninguna garantía de que las diferentes réplicas acaben almacenándose en diferentes nodos de la base de datos.

En la transparencia podemos ver un ejemplo de cómo se podría realizar la replicación del objeto con clave *REM* del *bucket Artist*. Podemos ver que el número de réplicas del sistema es 3. En consecuencia, el objeto se almacenará en tres particiones consecutivas. En el caso de que el nodo que alberga una de las réplicas caiga o quede desconectado de la red, el sistema sería capaz de obtener los datos de las otras dos réplicas, evitando así un fallo en la operación.

Los *vnodes* siguen una organización de tipo P2P. El nodo virtual que recibe una petición es el encargado de resolverla realizando las peticiones a los otros *vnodes* implicados. De igual forma, todos los *vnodes* pueden recibir operaciones de consulta y modificación. Esto permite una alta disponibilidad, pero puede causar problemas de consistencia.

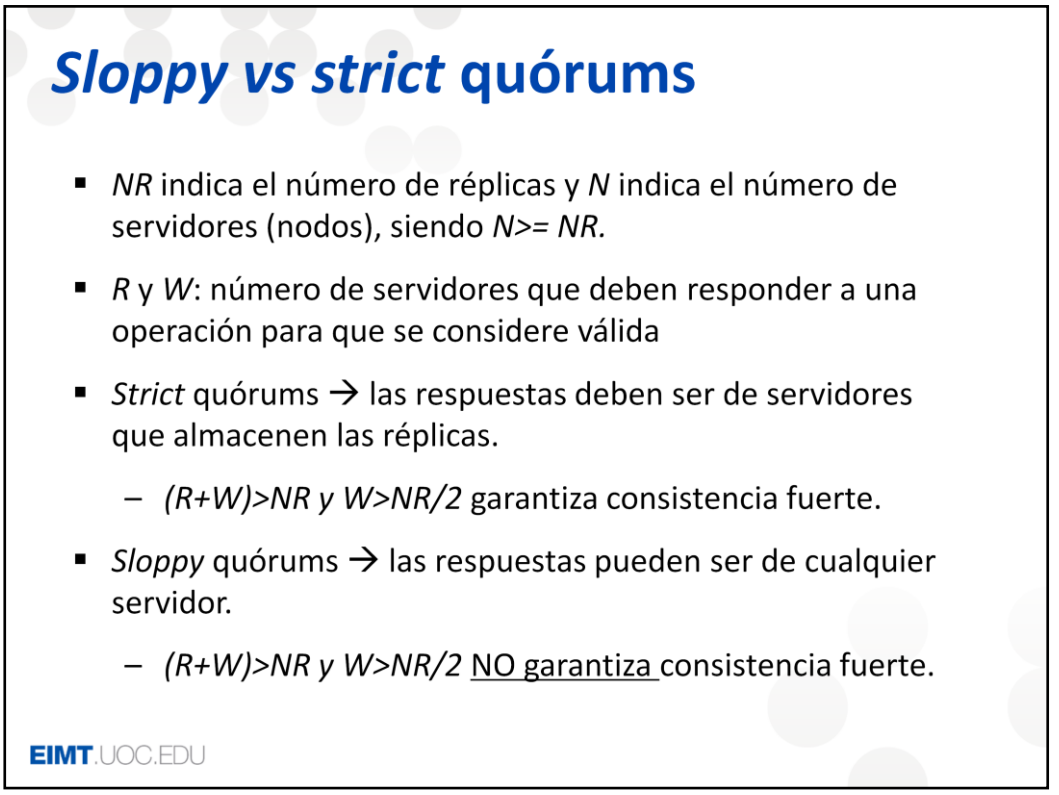
Tal y como hemos comentado antes, la gestión de la concurrencia en Riak se realiza mediante *vector clocks*. La gestión de réplicas se realiza mediante quórums, donde los usuarios indican, para cada operación, cuántos *vnodes* deben responder antes considerar una respuesta como válida. Vamos a explicar con más detalle el funcionamiento de los quórums.

En una operación de escritura, se envía la petición a los *N vnodes* que contienen las réplicas de los datos a modificar/añadir. La petición también puede definir dos parámetros adicionales, el parámetro *W* y el parámetro *DW*. El parámetro *W* indica el número de *vnodes* que deben contestar satisfactoriamente para que la operación termine con éxito. Por otro lado, con el parámetro *DW,* no se espera que los nodos virtuales indiquen que la operación se ha ejecutado satisfactoriamente, sino que la operación ha sido ejecutada satisfactoriamente y ha sido almacenada en la base de datos (es decir, garantiza su durabilidad en el sistema). El uso de *W*, por defecto, podría no garantizar la durabilidad de los objetos porque los nodos que almacenan el valor podrían "caer" antes de realizar su escritura en disco.

Las operaciones de lectura funcionan de manera parecida. Los usuarios pueden indicar un parámetro *R* que indica el número de *vnodes* que deben responder antes de que el valor consultado sea devuelto al cliente.

Los valores de *W* y *R* se pueden definir a nivel *bucket* pero pueden indicarse también para cada operación realizada.

Tal y como se ha comentado anteriormente en la asignatura, escoger los valores adecuados de *R* y *W* puede permitir aumentar o disminuir la disponibilidad de la base de datos y la consistencia de sus datos. Riak (según su documentación) por defecto ofrece consistencia final en el tiempo (*eventual consistency*) definiendo el valor de las variables *W* y *R* en (*(N_val*/2) + 1). Es decir, Riak espera a que más de la mitad de las réplicas contesten antes de confirmar una operación de lectura o escritura. Este hecho puede parecer extraño porque, cuando presentamos el modelo BASE, dijimos que bajo estas condiciones se garantizaba la consistencia fuerte. No se trata de un error, lo que pasa es que Riak utiliza *sloppy* quórums, que son un tipo de quórums menos estrictos que los *strict* quórums*,* que son los que presentamos en la parte de BASE.



A continuación vamos a explicar las principales diferencias entre *sloppy* y *strict* quórums.

Supongamos una base de datos distribuida y replicada con *N* servidores. La base de datos guarda *NR* réplicas de todos sus datos. Si se quiere hacer una gestión de réplicas mediante quórums se pueden definir valores de R y W para ajustar la disponibilidad/consistencia de los datos. Estos parámetros indican el número de servidores que deben responder a una operación para que se considere válida.

Cuando la base de datos recibe una operación debe identificar en qué nodos se encuentran los datos replicados y enviar la operación a dichos nodos. El número de nodos donde se envía la petición no podrá superar el valor de *NR*. Ahora bien ¿Qué pasa si no contesta un número suficiente de nodos a la operación? Tenemos dos opciones:

1) Denegar la operación, reduciendo la disponibilidad de la base de datos.

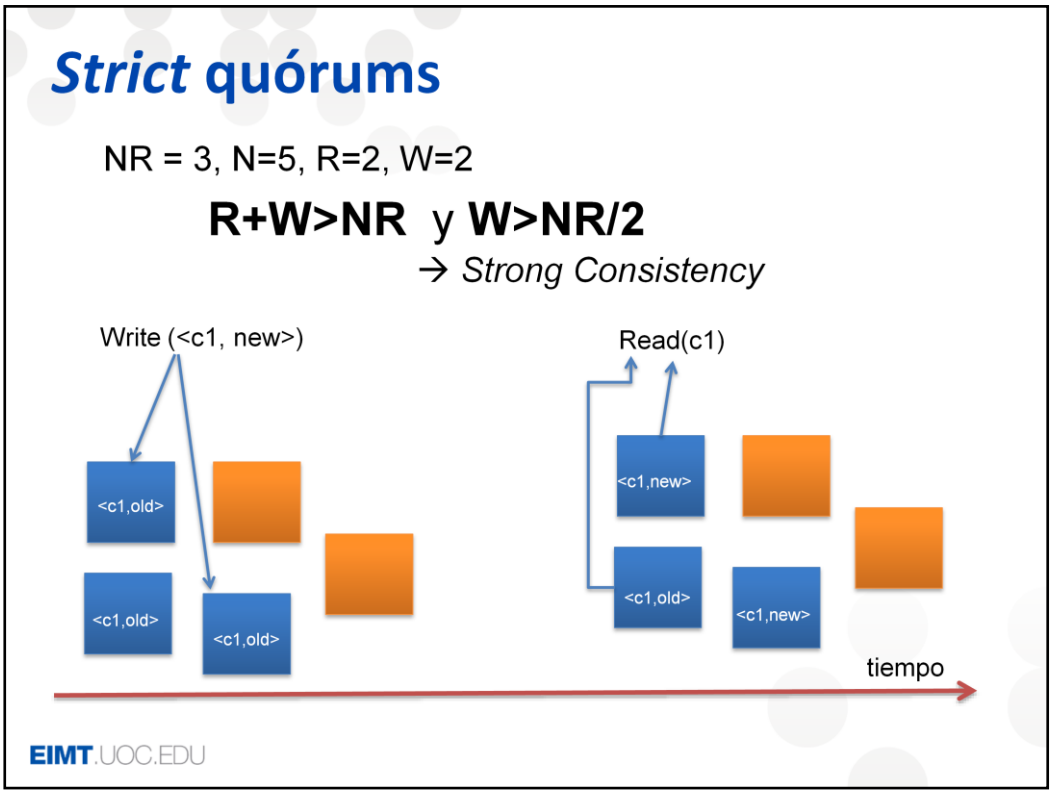
2) Enviar la operación a otros nodos de la base de datos, aunque éstos no almacenen los datos de la réplica afectada. Estos nodos reciben la responsabilidad de almacenar el resultado de la operación "temporalmente" y pasar dicho resultado a los nodos que contienen las réplicas tan pronto como éstos estén disponibles. Esta opción aumenta la disponibilidad de la base de datos, pero reduce la consistencia.

La primera opción es lo que se denomina *strict* quórum, mientras que la segunda opción es lo que se denomina *sloppy* quórum.

Cuando se usa un *strict* quórum las operaciones sólo se distribuyen a los nodos que contienen una réplica de los datos afectados. Eso garantiza que cuando se cumplen las inecuaciones (R+W)>NR y W>NR/2 siempre habrá una intersección entre los nodos afectados por cada operación, garantizando *strong consistency*.

Contrariamente, cuando estamos en un caso de *sloppy* quórum, todos los nodos de la base de datos pueden atender cualquier operación. Esto causa que no podamos garantizar que haya una intersección entre los nodos afectados por las distintas operaciones, lo que implica que el sistema no pueda garantizar *strong consistency*.

Vamos a ejemplificar el funcionamiento de estos tipos de quórums mediante un ejemplo.

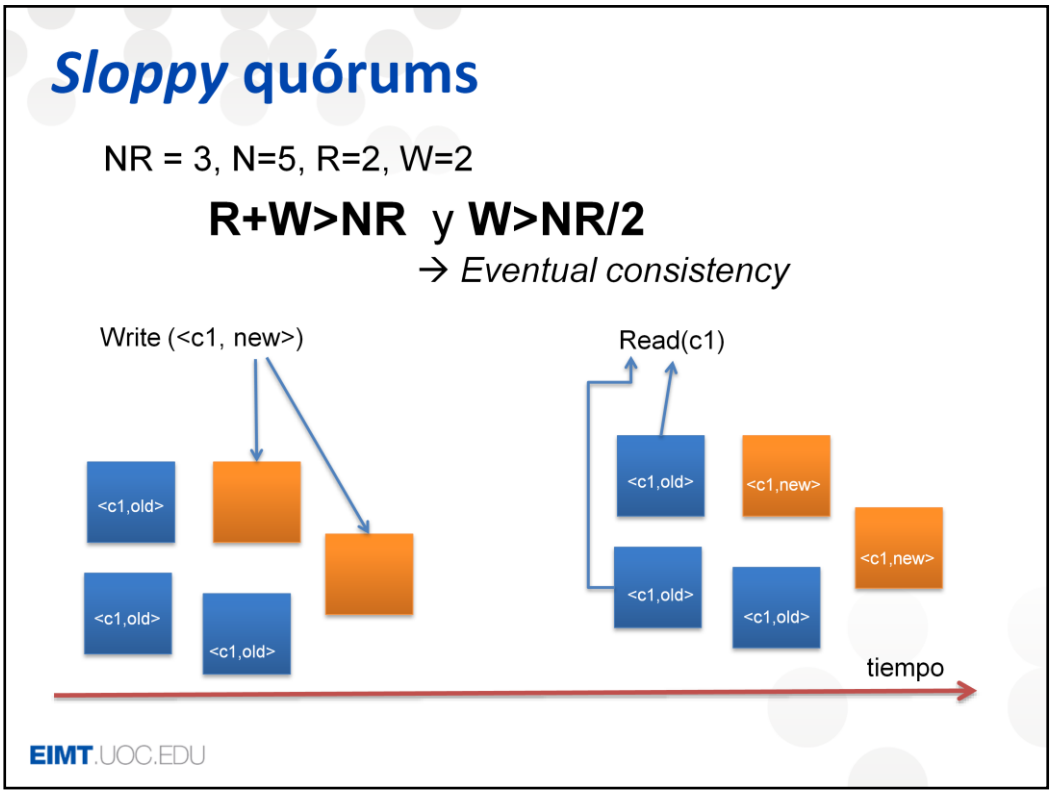


Supongamos una base de datos de tipo clave-valor distribuida y replicada con 5 servidores que guarde 3 réplicas de los datos. Supongamos también que distintas réplicas se almacenan en distintos servidores, y que la base de datos utiliza un sistema de *strict* quórum para gestionar las réplicas.

Imaginemos que queremos realizar una modificación del objeto con clave *c1*. Como el sistema utiliza un sistema estricto de quórums sólo enviará la operación de escritura sobre los servidores que almacenen las réplicas de los datos (en la figura son los servidores pintados en azul). Con la configuración actual la operación resultaría satisfactoria en el caso de que sólo dos servidores respondieran afirmativamente a la operación, tal y como podemos ver en la figura. En este caso, tendríamos dos réplicas con el valor nuevo y una réplica con el valor antiguo.

Más adelante, si se realizara una operación de lectura sobre la misma clave (*c1*), la operación se distribuiría a los tres servidores que contienen las réplicas. En este caso, si respondieran dos de ellos, se aceptaría la operación y podríamos garantizar que el resultado contiene alguna réplica con el valor más actualizado. El motivo es que seguro que una de las dos respuestas provendrá de uno de los servidores que participó en la operación anterior. Por lo tanto, seguro que uno de los valores devueltos será el correcto y, en caso de conflicto, el sistema será capaz de identificar que el valor *new* es el más reciente, y por lo tanto, devolverá este valor como válido.

Pero… ¿Cuál sería el comportamiento del sistema en caso de que trabajáramos con *sloppy* quórums?



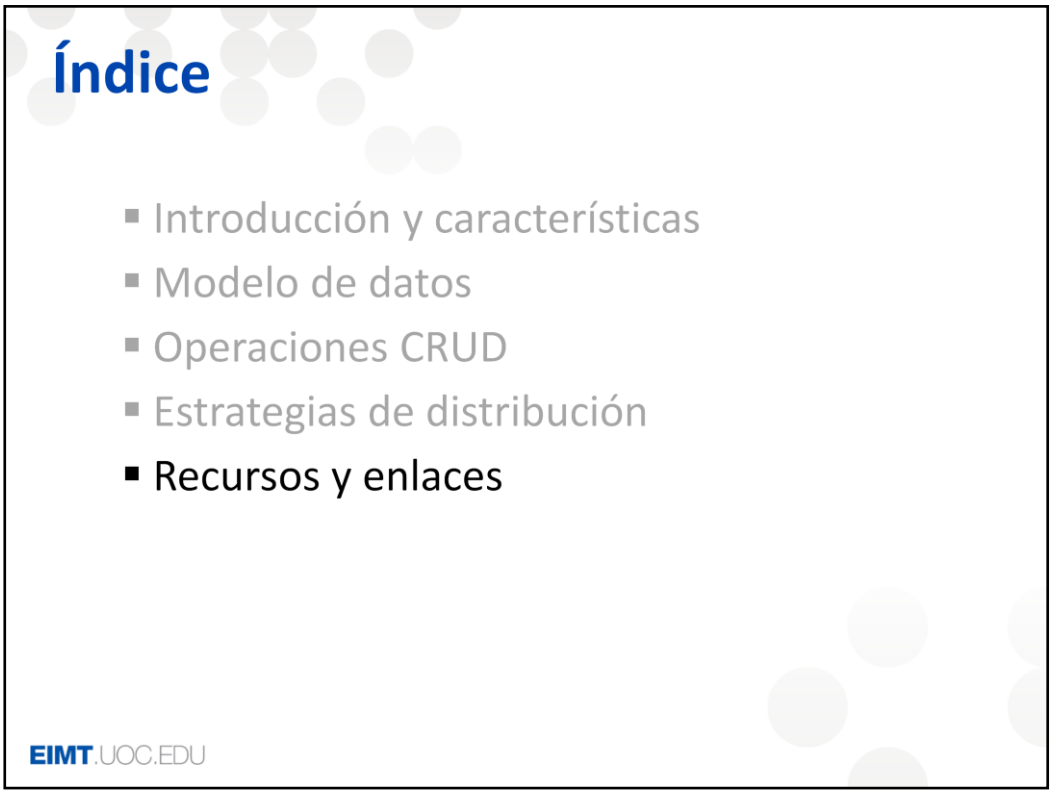
Vamos a suponer que la base de datos trabaja con *sloppy* quórums.

Supongamos que, por ejemplo, por problemas en la red al ejecutar la operación de escritura, los nodos principales (los que almacenan las réplicas) no contestan. Entonces el gestor de base de datos podría redirigir la operación a los otros dos nodos restantes. Estos nodos (pintados en naranja en la figura) no contenían anteriormente datos del objeto con clave *c1*.

En sistemas como éste, los valores escritos sobre los nodos "alternativos" se propagarán a los nodos que contienen las réplicas de los objetos tan pronto como éstos estén accesibles. No obstante, supongamos que los problemas de la red se solucionan y los nodos que contienen las réplicas reciben una operación de lectura del objeto con clave *c1* antes de que los nuevos valores de *c1* se hayan podido propagar. En este caso dos de los nodos principales devolverían el valor antiguo del objeto (este valor es *old*). El valor no es el último valor actualizado en la base de datos. Queda claro que el sistema no garantiza consistencia fuerte.

Con el tiempo los nuevos valores de *c1* se propagaran a los nodos principales, y finalmente todos los nodos contendrán valores consistentes. Por lo tanto podemos decir que en este caso el tipo de consistencia que garantiza el sistema es consistencia final en el tiempo.

Riak utiliza *sloppy* quórums para gestionar las réplicas hasta la versión 2.0. No obstante, a partir de la versión 2.0, se pueden configurar los *buckets* para que garanticen *strong consistency*.



Y hasta aquí esta introducción a Riak. En esta presentación hemos examinado las principales características de esta base de datos, su modelo de datos, las operaciones que permiten añadir, borrar, modificar y consultar datos en la base de datos y cómo se realiza la distribución y la replicación de datos. La presentación tiene por objetivo ser eminentemente introductoria. Para cualquier desarrollo real en Riak aconsejamos consultar su manual de referencia 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 Riak: http://basho.com/riak/

Documentación oficial: http://docs.basho.com/riak/2.0.0pre11/

Little Riak Book http://littleriakbook.com/

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.

Y hasta aquí esta presentación dedicada a Riak. Esperamos que os haya sido de ayuda.

Que tengáis un buen día.