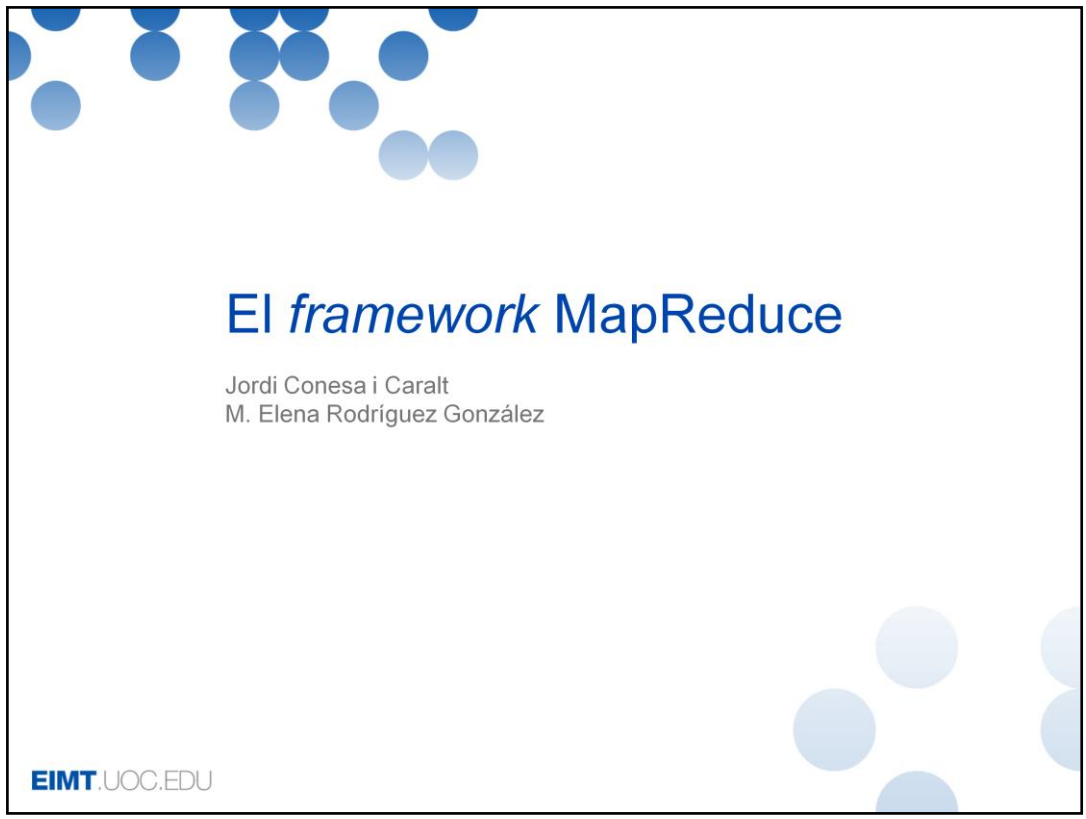
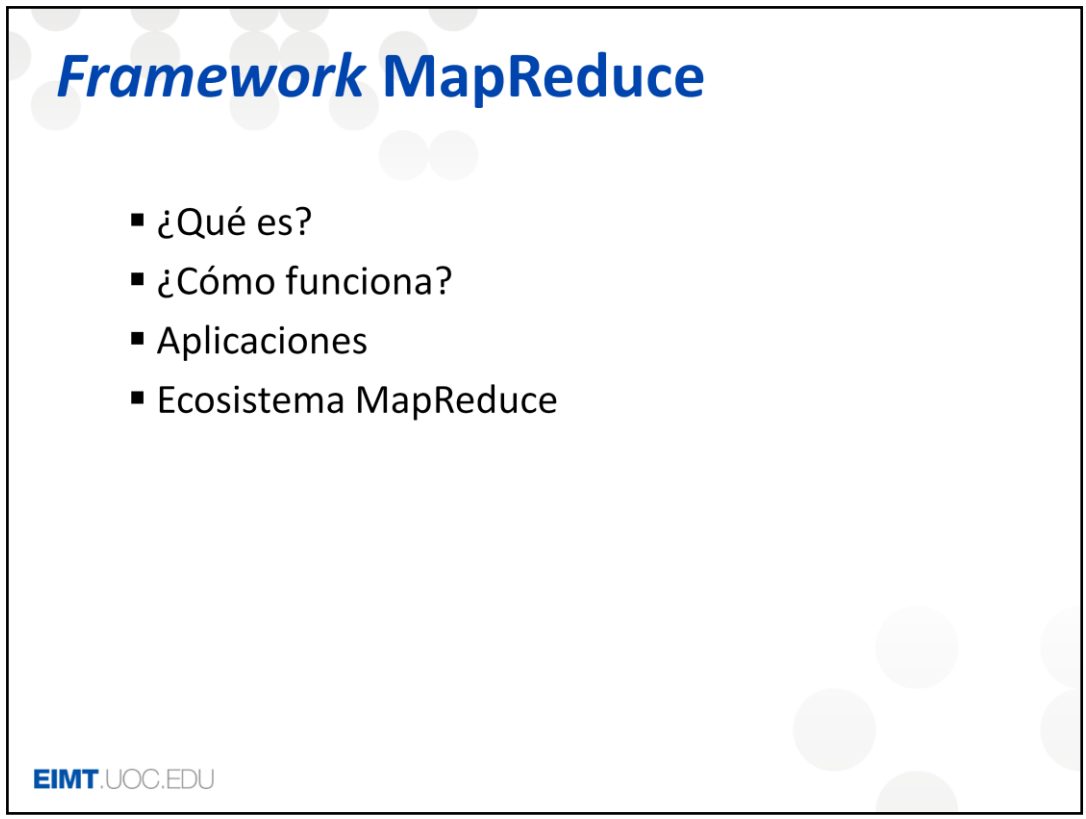


Bienvenidos a una nueva presentación donde analizaremos de forma conceptual el *framework* MapReduce desde una perspectiva de gestión de datos. En concreto, explicaremos qué es, cómo funciona, sus ventajas, así como algunos ejemplos de utilización del mismo y su integración con las bases de datos NoSQL.



Empezaremos hablando de qué es MapReduce, cuáles son sus componentes y en qué situaciones es conveniente su uso.

Posteriormente veremos de forma introductoria (y mediante ejemplos) su funcionamiento. Abordaremos los aspectos más relevantes de funcionamiento, relegando sus detalles a otras asignaturas del máster.

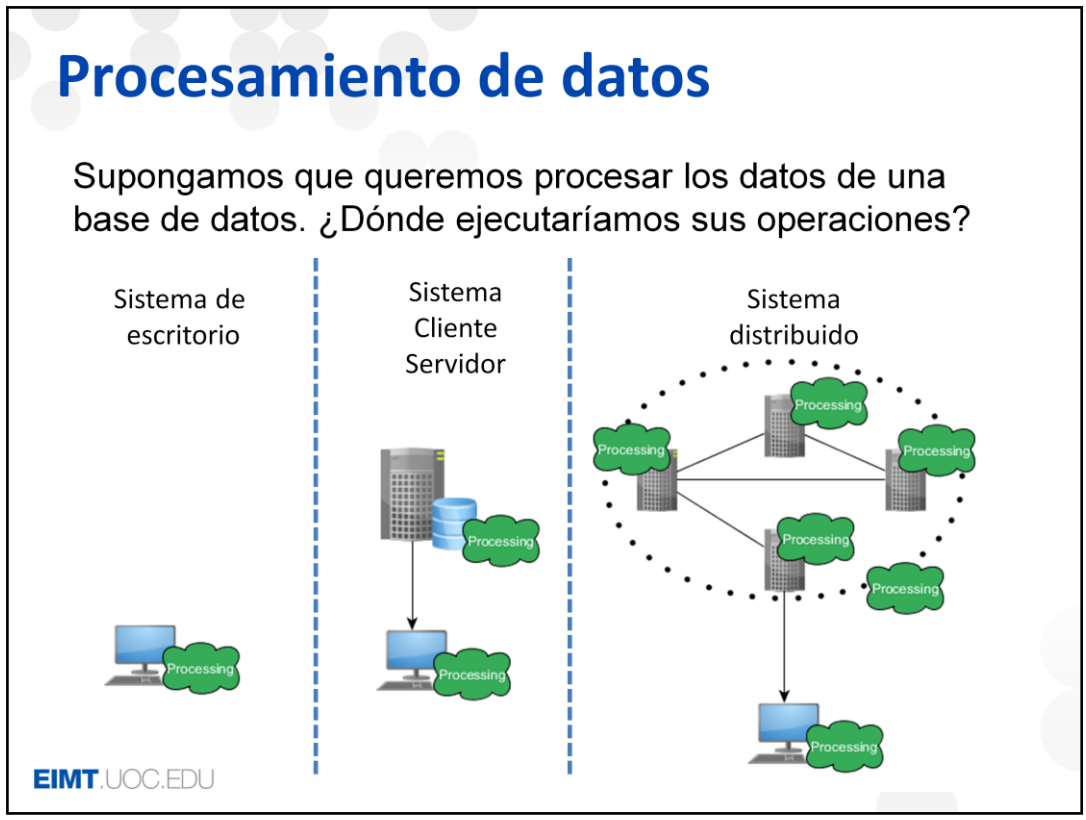
Terminaremos la presentación mostrando un conjunto de ejemplos reales donde se usa MapReduce y enumerando algunas de las herramientas que lo implementan.

Framework MapReduce

- ¿Qué es?
- ¿Cómo funciona?
- Aplicaciones
- Ecosistema MapReduce

EIMT.UOC.EDU

Comenzamos, pues, viendo las ideas fundamentales asociadas a MapReduce.



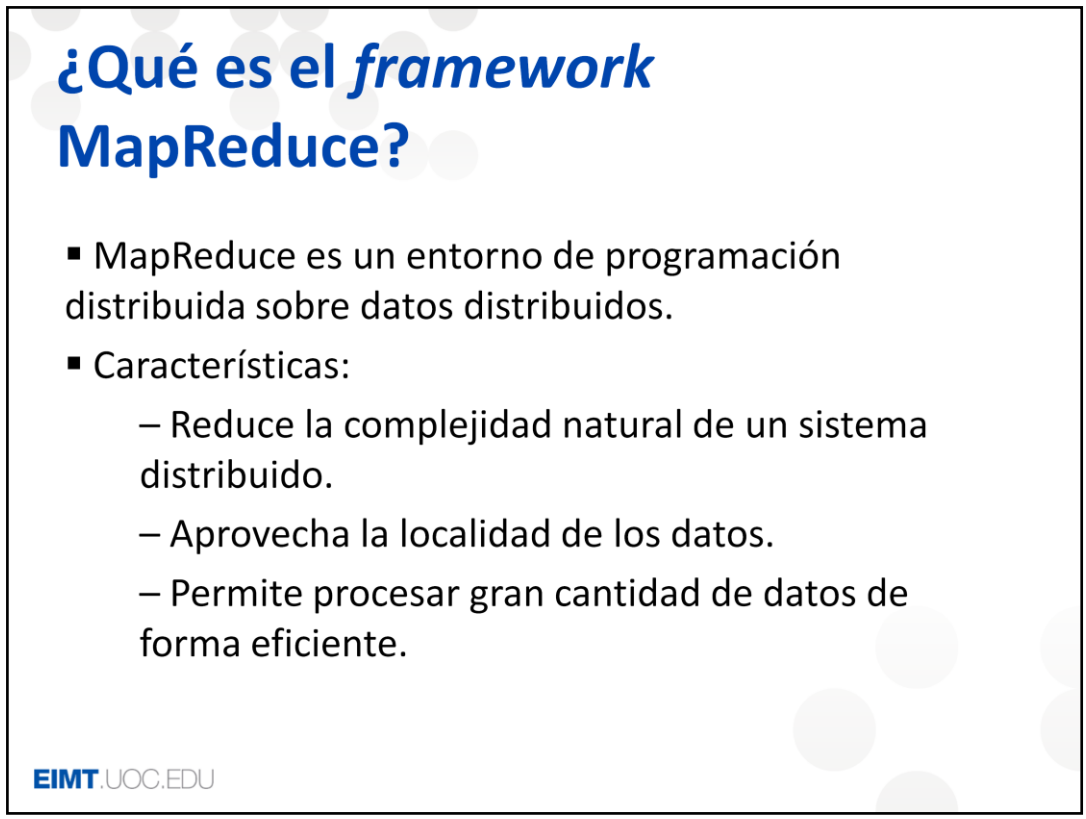
Antes de presentar qué es MapReduce, vamos a justificar mínimamente el porqué de su necesidad. Para ello nos plantearemos el siguiente supuesto: "Supongamos que tenemos una base de datos que contiene información sobre las compras de productos y queremos ejecutar una operación sobre la misma para calcular el total de productos vendidos por cada tipo de producto. ¿Dónde debería calcularse dicha agregación? "

En el caso de que la base de datos esté en un ordenador de escritorio la solución será fácil. La opción más viable será ejecutar la agregación dentro del mismo ordenador.

En el caso que tengamos una base de datos que sigue una arquitectura de cliente servidor el tema ya se complica más, ya que la agregación de los datos se podrá realizar en el servidor o en el cliente. En el primer supuesto, penalizaremos el rendimiento de la máquina servidora, ya que necesitará ejecutar la agregación además de gestionar la base de datos. En el segundo, aligeramos al servidor de la carga de trabajo de realizar la agregación, pero todos los datos en bruto se enviarán al cliente para que éste haga la agregación, provocando una sobrecarga en la red en términos de coste de transmisión de datos.

Si la base de datos es distribuida, como la de la parte derecha de la figura, el procesamiento se puede hacer de forma centralizada o distribuida. Si se realiza de forma centralizada, habrá un nodo que recibirá todos los datos y realizará la operación requerida (la agregación). Cuando se realiza de forma distribuida, el cálculo se distribuye en distintos nodos de la red. En un sistema de datos masivos, el procesamiento centralizado de datos suele ser inviable, tanto por los requerimientos de máquina necesarios para realizar el cálculo en un solo ordenador, como por el elevado coste de enviar todos los datos por la red hacia el nodo que los procese.

Los sistemas MapReduce, como veremos más adelante, permiten el procesamiento distribuido sacando provecho a la localidad de datos.

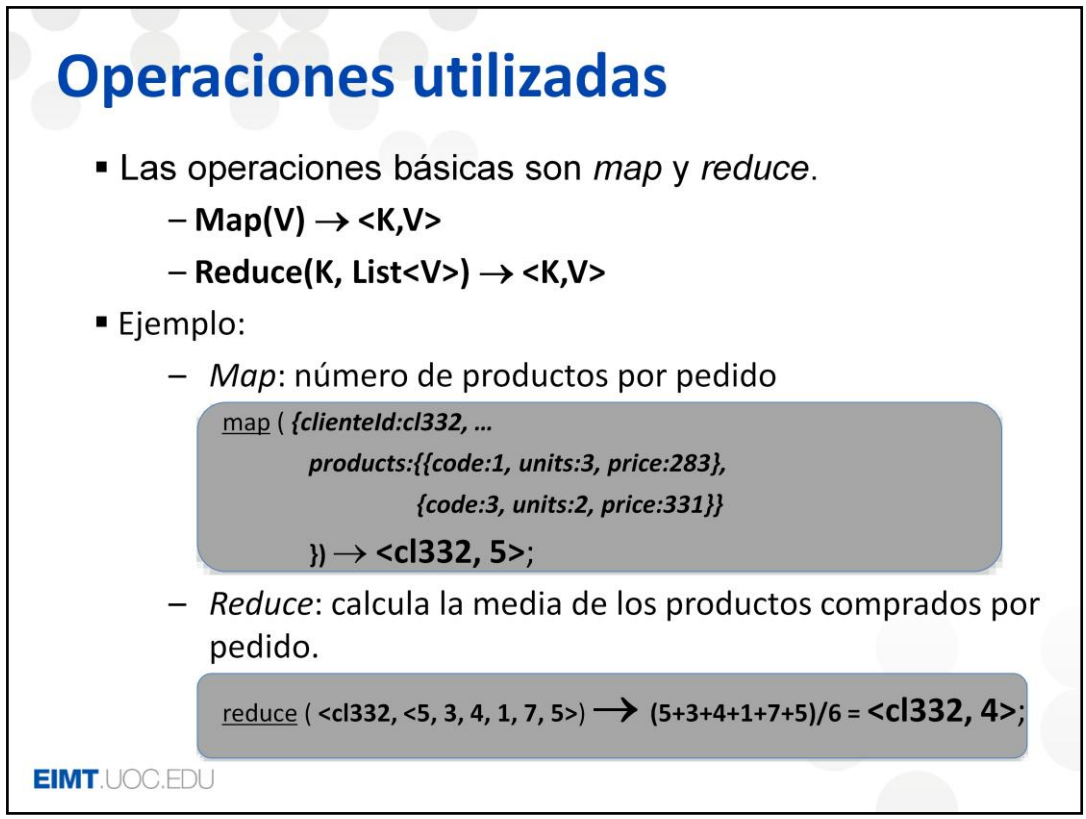


MapReduce es un sistema de computación distribuida que aprovecha la localidad de datos para realizar cálculos de forma más eficiente. Fue originariamente propuesto por Google Inc. en 2003 que lo utilizaba para optimizar sus operaciones de análisis e indexación de datos.

El sistema MapReduce simplifica en gran medida la complejidad de la programación, configuración y mantenimiento de un sistema distribuido. Por un lado, un programador sólo debe implementar dos operaciones para ejecutar una operación de MapReduce: las operaciones *map* y *reduce* que veremos más adelante. Conceptualmente, la idea es que la operación *map* se ejecute en los nodos que contienen los datos y permite realizar operaciones sobre dichos datos (convertir los datos en un formato más adecuado, realizar un precálculo necesario, etc.). Posteriormente, una operación *reduce* se encargará de recoger las salidas de las operaciones de *map* realizadas en los distintos nodos e integrarlas para devolver el resultado final. MapReduce proporciona un entorno que se encarga de gestionar todo el proceso automáticamente, reduciendo la carga de trabajo del programador para garantizar la distribución de datos, la tolerancia a fallos y la escalabilidad de la operación.

Por lo tanto, MapReduce permite preprocesar los datos de forma localmente en los nodos donde se encuentran, y enviar a la función de *reduce* sólo la información necesaria para generar el resultado final. Esta estrategia permite aprovechar la localidad de los datos y evita sobrecargar la red enviando datos irrelevantes.

Por estos motivos, MapReduce permite la ejecución de procesos sobre datos masivos en sistemas altamente distribuidos de forma eficiente. Una de las dificultades del sistema MapReduce es encontrar unas buenas funciones de *map* y *reduce* para la operación requerida.



Las operaciones básicas de MapReduce, como ya hemos dicho en la transparencia anterior, son *map* y *reduce*. Aparte de estas funciones se pueden definir otras funciones, como pueden ser, por ejemplo, las funciones *reader*, *writter*, *partition* y *shuffle*. Las dos primeras permiten leer y escribir los datos que participan en el proceso MapReduce y las dos últimas sirven para elegir cómo agregar y distribuir los resultados de la función *map* en los nodos *reduce*.

La operación de *map* se encarga de aplicar una función a un valor. El objetivo de la función de *map* es reducir un problema a N subproblemas más simples. Una función *map* puede devolver 0 o más parejas *<clave, valor>.*

La operación de *reduce* se encarga de integrar los resultados parciales (obtenidos por las operaciones de *map*) para resolver los subproblemas planteados. Esta función recibirá una clave y un conjunto de valores, y retornará 0 o más parejas *<clave, valor>* que representan la solución al problema.

Las operaciones de MapReduce pueden concatenarse, es decir, las salidas de una función *reduce* podrían utilizarse como entrada de nuevas operaciones *map*.

Por ejemplo, supongamos que queremos averiguar, para cada cliente, el número medio de productos adquiridos en cada pedido. Para calcular este valor, lo que podemos hacer es identificar, para cada pedido, el código de cliente y el número de productos adquiridos, para luego agrupar estos datos por cliente y realizar la media de los productos comprados. En este caso la función *map* sería la encargada de leer un pedido y generar una pareja *<clave, valor>* donde la clave es el código del cliente y el valor es el número de productos vendidos en el pedido. En el ejemplo de la transparencia, podemos ver como dado el pedido del cliente *cl332*, la salida es *<cl332, 5>*, ya que este cliente compró 5 productos en dicho pedido.

Una vez se haya ejecutado la operación de *map* para todos los pedidos del sistema, tendremos N entradas por cliente. Por ejemplo, para el cliente *cl332*, tendríamos N parejas de tipo *<cl332, valor_i>*. Donde *valor_i* es el número de productos comprados por el cliente en el pedido i.

La función *reduce* recibe, para cada cliente, sus N valores. Cada valor indica el total de productos comprados en un pedido por el cliente. Esta función debe hacer la media de dichos valores y retornar el valor de esta media tal y como se muestra en la figura.

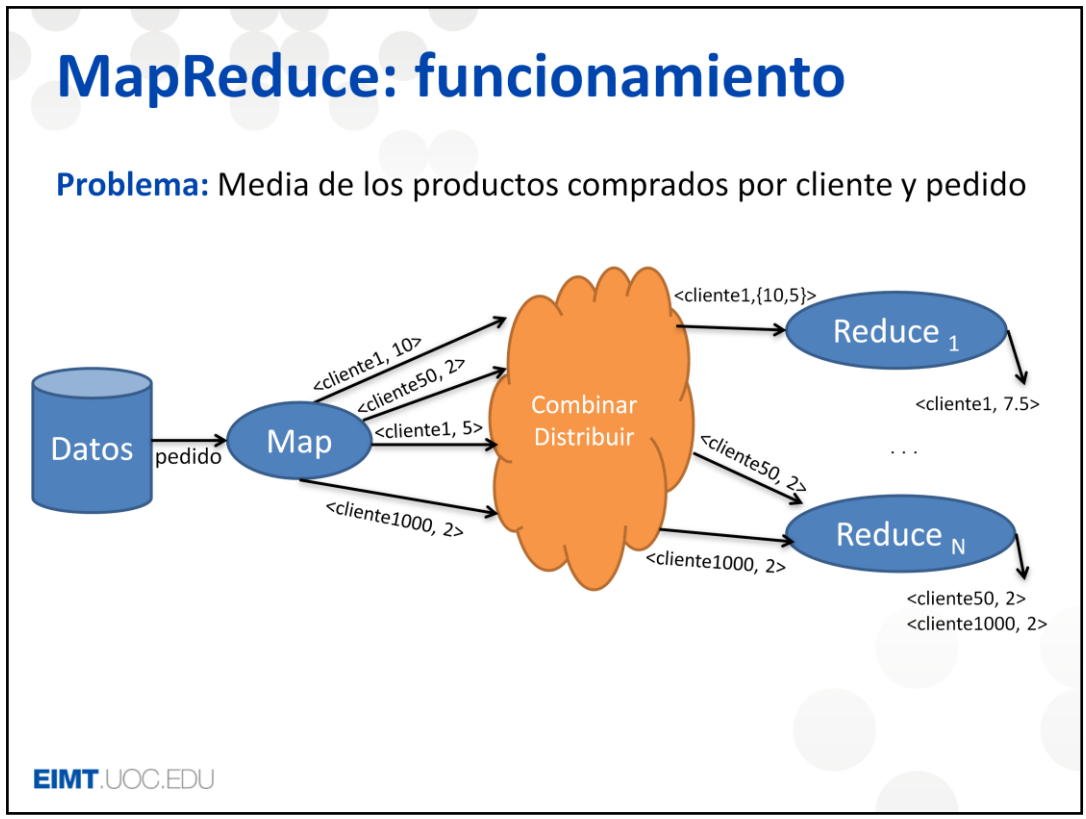
Notad que ésta es sólo una de las posibles implementaciones MapReduce que permiten solucionar el problema planteado. Dado un problema, puede haber distintas maneras de solucionarlo utilizando funciones MapReduce. La elección de una u otra puede condicionar enormemente la eficiencia del proceso, tanto en coste computacional como en la cantidad de datos a enviar por la red.

Framework MapReduce

- ¿Qué es?
- ¿Cómo funciona?
- Aplicaciones
- El ecosistema MapReduce

EIMT.UOC.EDU

Una vez vistas las ideas básicas de MapReduce, analizaremos en más detalle su arquitectura y funcionamiento mediante algunos ejemplos.



Vamos a ver cómo funciona el *framework* MapReduce de forma incremental. Empezaremos viendo cómo se integran las operaciones *map* y *reduce* para después complementar las explicaciones con las operaciones de combinación (*partition*) y distribución (*shuffle*) de datos. En esta explicación nos centraremos en el funcionamiento de MapReduce en el contexto de bases de datos.

Tal y como hemos dicho, tratar un problema en MapReduce implica dos fases principales. Una fase de *map* donde el problema se subdivide en problemas más simples y otra parte *reduce* que recoge los resultados de estos problemas más simples y se combinan para solucionar el problema general.

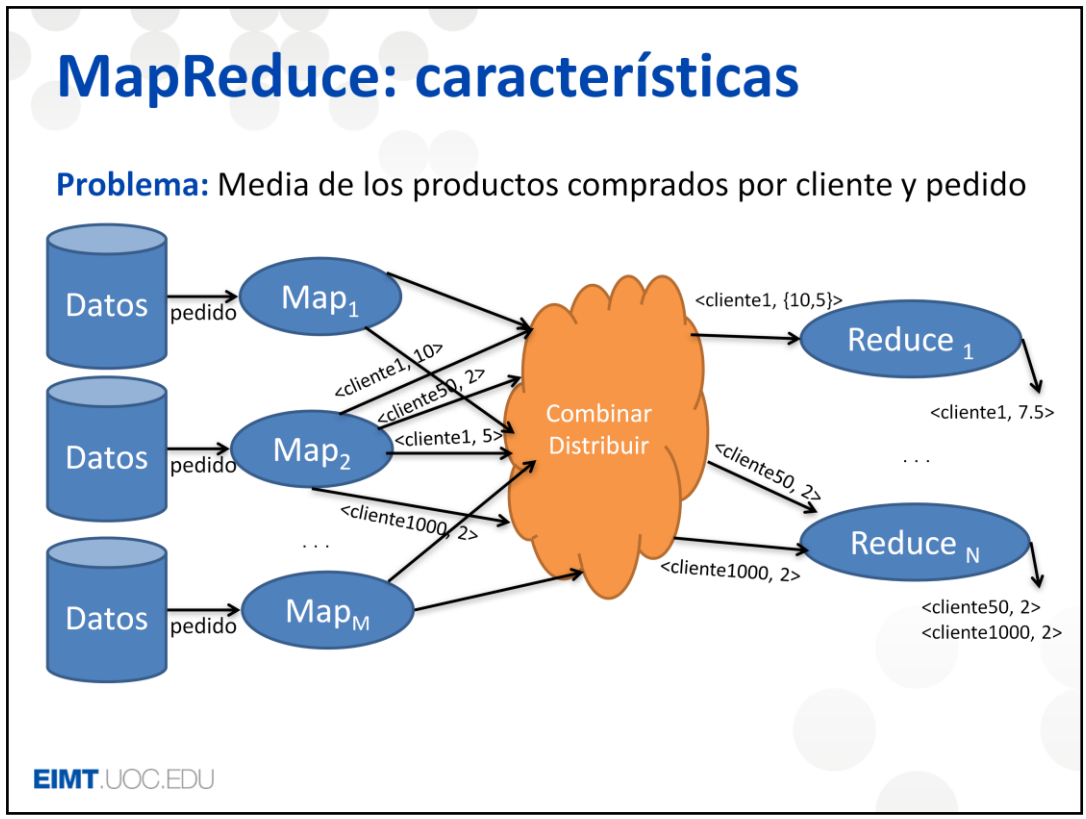
La fase de *map* se encarga de leer los datos a procesar y generar salidas de tipo *<clave, valor>*. Normalmente, en un entorno de bases de datos la entrada de la función *map* suele ser un registro de la base de datos. Por ejemplo, en el problema que nos ocupa (calcular la media de los productos comprados en cada pedido por cada cliente) la entrada a la función *map* podría ser un conjunto de registros, donde cada registro es un agregado que define un pedido. La operación *map* recibiría un agregado cada vez e identificaría el cliente que ha efectuado la compra y el número de productos que ha adquirido. La salida podría ser un par *<clave, valor>* donde la clave es el código de cliente y el valor es el número de productos demandados. En la transparencia podemos ver algunos ejemplos.

Una vez ejecutada la operación *map*, se distribuyen sus resultados a diferentes nodos *reduce*. Antes de que los datos lleguen a los nodos *reduce* se realizan dos operaciones, una de combinación y otra de distribución. Más adelante entraremos en el detalle de dichas operaciones. De momento supondremos el caso más sencillo, que es cuando el sistema se encarga de agrupar los datos por clave y de realizar la distribución de forma automática. En este caso, el sistema agrupará todas las entradas con la misma clave en una nueva pareja *<clave, valor>* donde el valor será la secuencia de valores recibida con la misma clave (por ejemplo, en el caso del *cliente1* el resultado será una pareja con clave *"cliente1"* y valor igual a la secuencia de 10 y 5). Por otro lado, una vez combinadas, las entradas se distribuirán a los diferentes nodos *reduce*.

La distribución de los datos intermedios a los nodos *reduce* se realizará en función de su clave. En el caso ideal tendríamos tantos nodos como valores distintos de la clave. Normalmente esto no sucede y se realizará la distribución de datos enviando distintas claves a un mismo nodo.

Cada nodo *reduce* recibirá entradas del tipo *<clave, lista_de_valores>* y deberá ejecutar la operación *reduce* para cada una de estas entradas y generar 0 o más salidas. Las salidas podrán ser almacenadas en una base de datos o bien podrán ser consumidas por cualquier otro proceso.

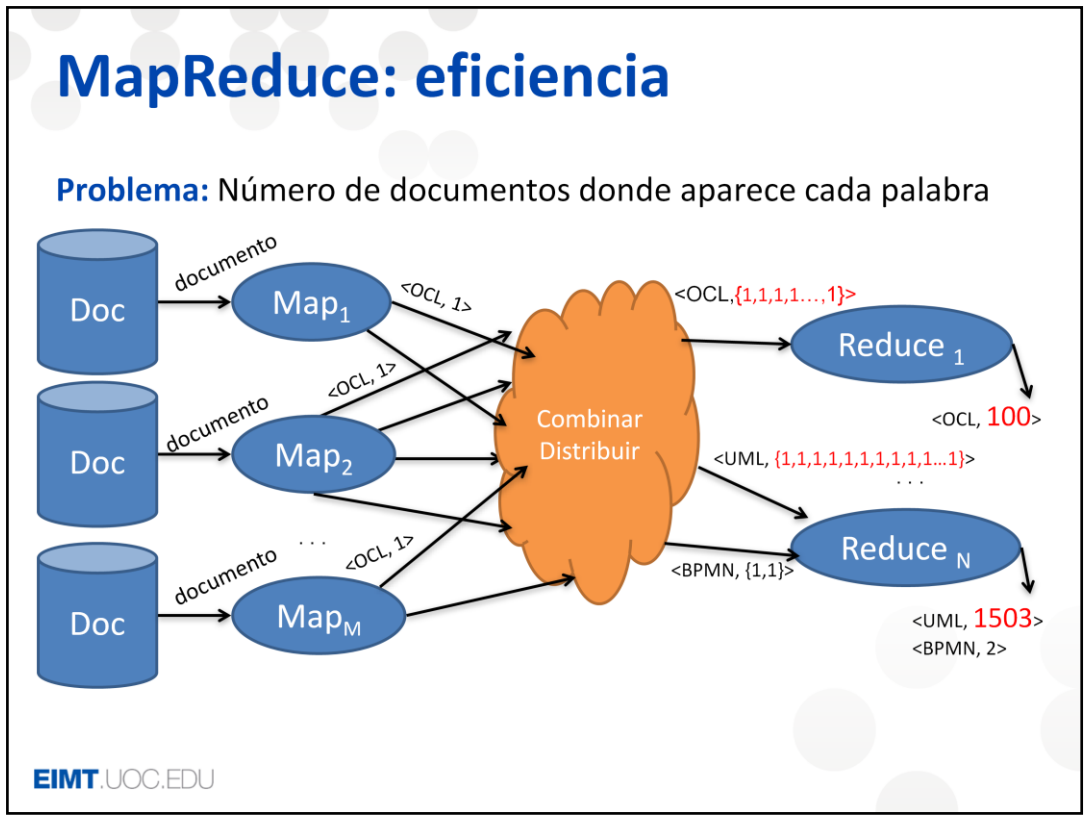
La función *reduce* será la encargada de combinar los resultados parciales para generar la solución del problema. En el caso que nos ocupa los nodos *reduce* deberán, para cada cliente, sumar los valores de la lista y dividir por el número de valores. En el ejemplo presentado, podemos ver que el sistema indica que el cliente1 adquiere una media de 7.5 productos por pedido y los clientes 50 y 100 una media de 2.



El caso presentado hasta ahora considera que sólo existe un nodo *map* (lo que podría verse como una base de datos centralizada). En el caso de que tengamos una base de datos distribuida podríamos tener tantos nodos *map* como nodos de la base de datos, tal y como podemos ver en la figura.

Como se ha comentado anteriormente y se puede ver en la figura, un esquema de procesamiento MapReduce facilita procesar los datos en origen, reduce el tráfico de la red, facilita el paralelismo y tiene una elevada tolerancia a fallos. La localidad se garantiza porque el procesado de datos se realiza en origen (en los nodos donde se almacenan) por las funciones *map*, haciendo que los datos parciales se filtren y agreguen de forma local. Como la función *map* retorna sólo los datos relevantes para el problema a tratar, se minimiza el tráfico de datos innecesarios en la red.

Por otro lado, el hecho de que los nodos *reduce* trabajen sólo con los datos de una clave determinada, implica que los datos son independientes entre sí y favorece el paralelismo. Por último, la distribución del procesamiento entre distintos nodos mejora la tolerancia a fallos del sistema, ya que el sistema puede levantar un nuevo nodo en caso de que uno de los nodos del sistema falle.



Para ilustrar más a fondo cómo se realiza la combinación y distribución de datos antes de enviarlos a los nodos *reduce*, vamos a ver un nuevo ejemplo.

Supongamos que queremos identificar las palabras clave más relevantes de una empresa. En caso de que tuviéramos todos los documentos de la empresa en una base de datos clave-valor podríamos hacerlo identificando las palabras que contienen y el número de documentos en que aparece cada palabra.

Bajo ese supuesto, vamos a utilizar la plataforma MapReduce para resolver el problema.

Supongamos que hemos creado una función *map* que retorna, para cada documento N salidas de tipo *<keyword, 1>*. Donde N es el número de palabras distintas del documento y *keyword* es cada una de las palabras. Por lo tanto, una salida del tipo *<OCL, 1>* indicará que la palabra OCL aparece en un documento. El sistema MapReduce, antes de enviar los datos a los nodos *reduce*, identificará las entradas *<keyword, 1>* con las misma clave y las agrupará de manera que tengamos *<keyword, sequencia_de_1>*, donde *secuencia_de_1* será una secuencia de tantos 1 como documentos donde aparezca la palabra. Después el sistema distribuirá dichos datos entre los distintos nodos *reduce* intentando equiparar la carga de trabajo de los mismos.

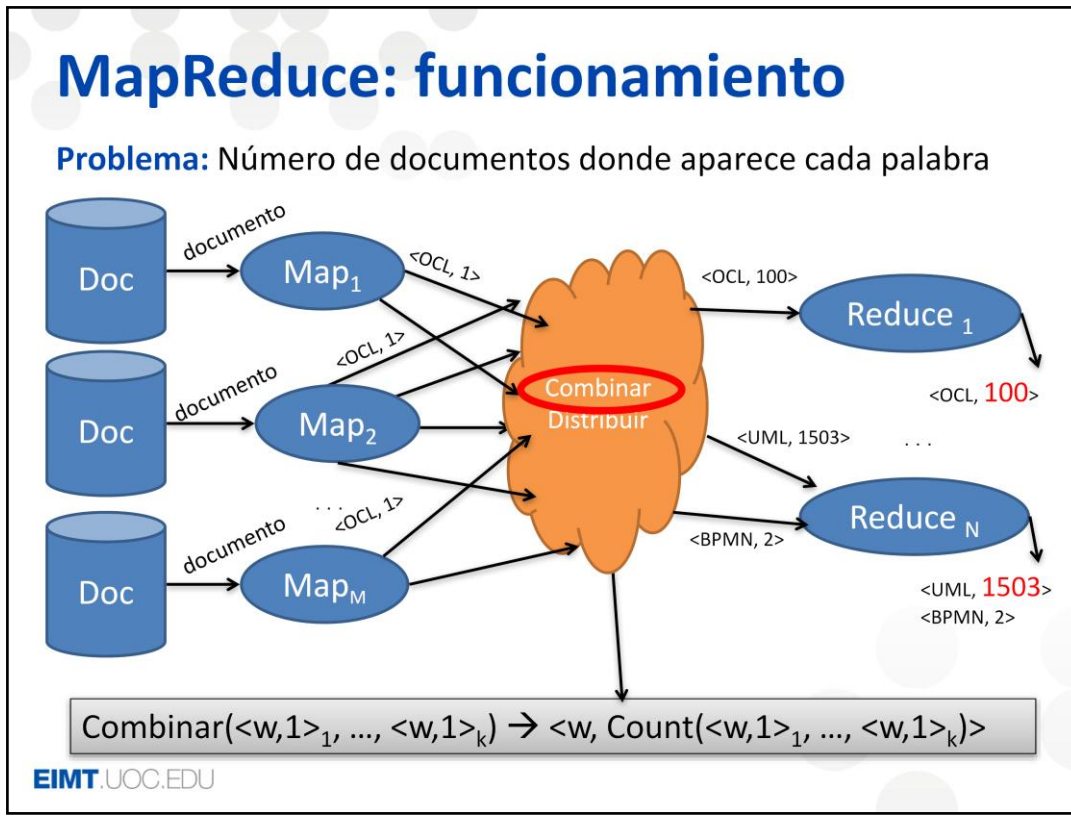
Por lo tanto la operación *reduce* recibirá una entrada del tipo *<palabra, lista de valores>*, donde lista de valores será una secuencia de unos de longitud variable.

En principio el sistema que hemos creado funcionaría correctamente y devolvería entradas como las que se muestran en la transparencia: la palabra "OCL" aparece 100 veces, la palabra "UML" aparece 1503 veces, etc.

No obstante, antes de continuar, vamos a hacernos una pregunta: ¿Creéis que el sistema creado es eficiente?

La respuesta es rotundamente NO. Tal y como se ha diseñado el sistema actual hay una gran cantidad de datos potencialmente irrelevantes que se transfieren por la red. Por ejemplo, para la palabra UML tenemos una secuencia de algo más de mil quinientos unos y para la palabra OCL cien.

Si tenemos un gran número de documentos y con una variabilidad de palabras alta podemos encontrarnos en una situación similar para el resto de palabras. Eso quiere decir cantidades ingentes de valores repetidos que podrían ser reducidos (o agregados). La pregunta es ¿Podemos optimizar el sistema para este proceso?



La respuesta es SÍ. Recordad que había dos operaciones que el sistema hacía por defecto pero que nosotros en algunos casos podremos personalizar, que son las operaciones combinar y distribuir. La operación de distribuir nos permite indicar cómo se distribuyen las claves en los distintos nodos *reduce*. Por otro lado, la función combinar es una función que permite agrupar (o combinar) los resultados de la función *map* con una misma clave antes de ser enviada a los procesos *reduce*.

En este caso, podemos definir la función combinar para que combine k entradas con la misma clave en una entrada con el mismo valor de clave y k como valor. De esta forma cambiamos la semántica del par *<p, 1>* (la palabra p aparece en un documento) por *<p, k>* (la palabra p aparece en k documentos).

Además también se deberá cambiar la operación *reduce* para que tenga en cuenta que hay que sumar la lista de valores en vez de contarlos.

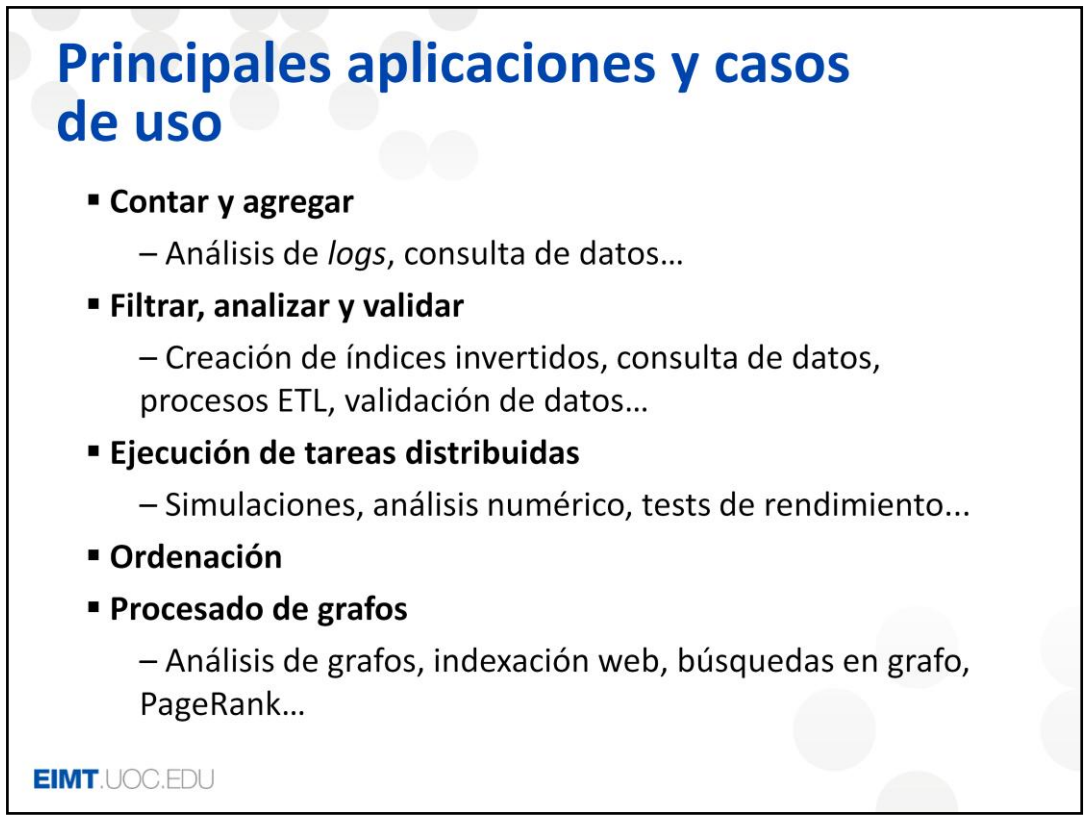
Es fácil de ver que utilizando la operación combinar podemos reducir el numero de datos que viajan en la red, y por lo tanto, conseguimos mejorar el rendimiento del proceso MapReduce.

Framework MapReduce

- ¿Qué es?
- ¿Cómo funciona?
- Aplicaciones
- El ecosistema MapReduce

EIMT.UOC.EDU

En esta sección veremos algunos de los problemas que se solucionan mediante MapReduce.



Los sistemas MapReduce se han democratizado y actualmente se utilizan en multitudes de problemas complejos que tratan con datos más o menos masivos. Podemos clasificar estos problemas en distintos patrones tal y como se muestra en esta transparencia.

Dado un gran numero de documentos, donde cada documento contiene un conjunto de elementos, puede ser relevante contar las ocurrencias de determinados elementos, la agregación de elementos, etc. Ejemplos de dichos procesos pueden ser el análisis de ficheros de *log* o realizar consultas de datos, tal y como hemos visto en los anteriores ejemplos de esta presentación. Un ejemplo de análisis de este tipo se puede encontrar en http://www.informit.com/articles/article.aspx?p=2017061, donde se calcula el número de visitas por hora en una página web determinada. En este caso la función *map* genera un par <*clave, valor*> donde la clave define la hora de visualización (concatenación del día, el mes, el año y la hora de visita) y el valor es 1. La función *reduce* cuenta las ocurrencias de 1, y por lo tanto, devuelve las visitas de la página por cada hora.

En otros casos, tenemos un conjunto de registros y debemos seleccionar cuáles de ellos cumplen una determinada condición y/o realizar alguna modificación sobre los mismos. Ejemplos de este patrón son la creación de índices invertidos (que sirven para la indexación de texto), la consulta de datos de una base de datos, la ejecución de procesos ETL o los procesos de validación de datos en un sistema de BI.

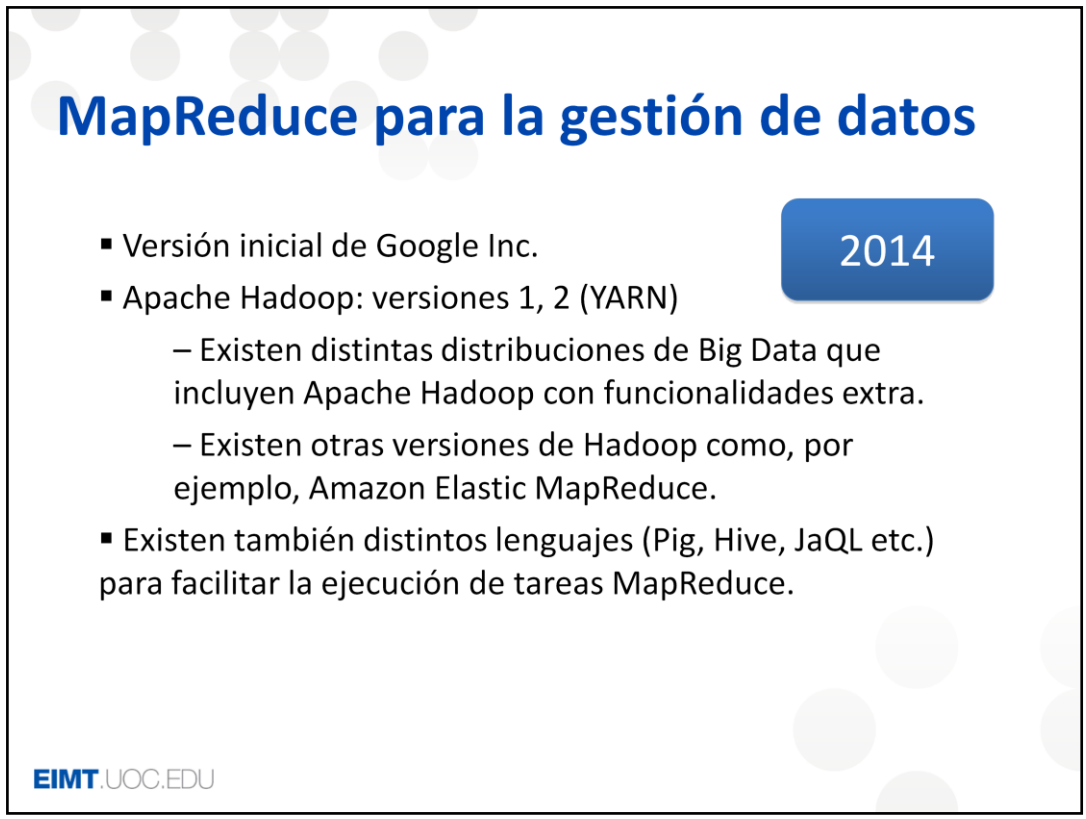
Otros ámbitos de aplicación de MapReduce son la ejecución de tareas distribuidas, la ordenación de grandes cantidades de datos y el procesado de grafos. Este último ámbito es especialmente relevante en el análisis de redes sociales, ya que permite realizar búsquedas en los grafos que forman las redes sociales, analizar estos grafos con distintos tipos de algoritmos, indexar/clasificar sitios web según distintos criterios (pageRank por ejemplo), etc.

Framework MapReduce

- ¿Qué es?
- ¿Cómo funciona?
- Aplicaciones
- El ecosistema MapReduce

EIMT.UOC.EDU

Ya por ultimo repasaremos brevemente el ecosistema de MapReduce en el contexto de las bases de datos NoSQL.

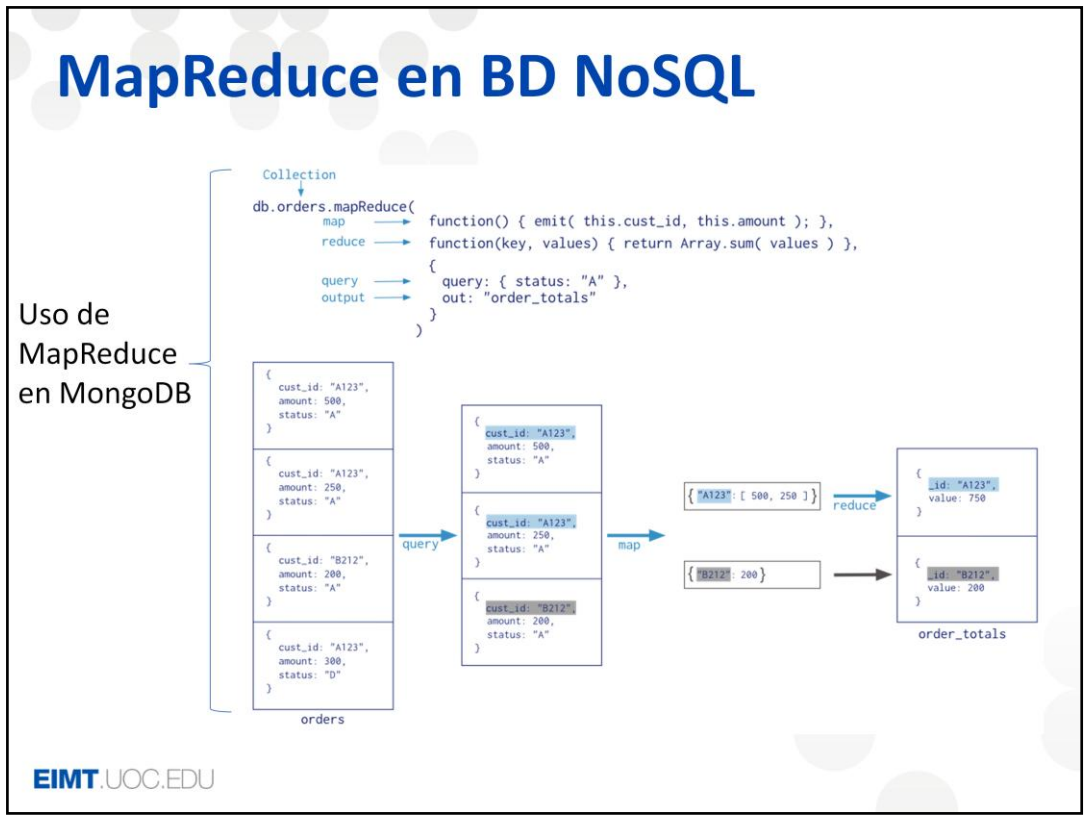


Aunque MapReduce es una plataforma inicialmente propuesta por Google, actualmente su distribución más popular es Apache Hadoop. Apache Hadoop es un proyecto de Apache de código abierto que proporciona un entorno MapReduce.

Actualmente existen dos versiones de Hadoop, la versión 1 y la versión 2. La versión 2 se denomina también YARN.

Hadoop se integra en muchas de las *suites* de Business Intelligence y/o de tratamiento de Big Data actuales, como pueden ser BigTop, Amazon Elastic MapReduce, Cloudera, HortonWorks Data Platform, MapR, IBM Infosphere BigInsights, Microsoft HDInsight, etc.

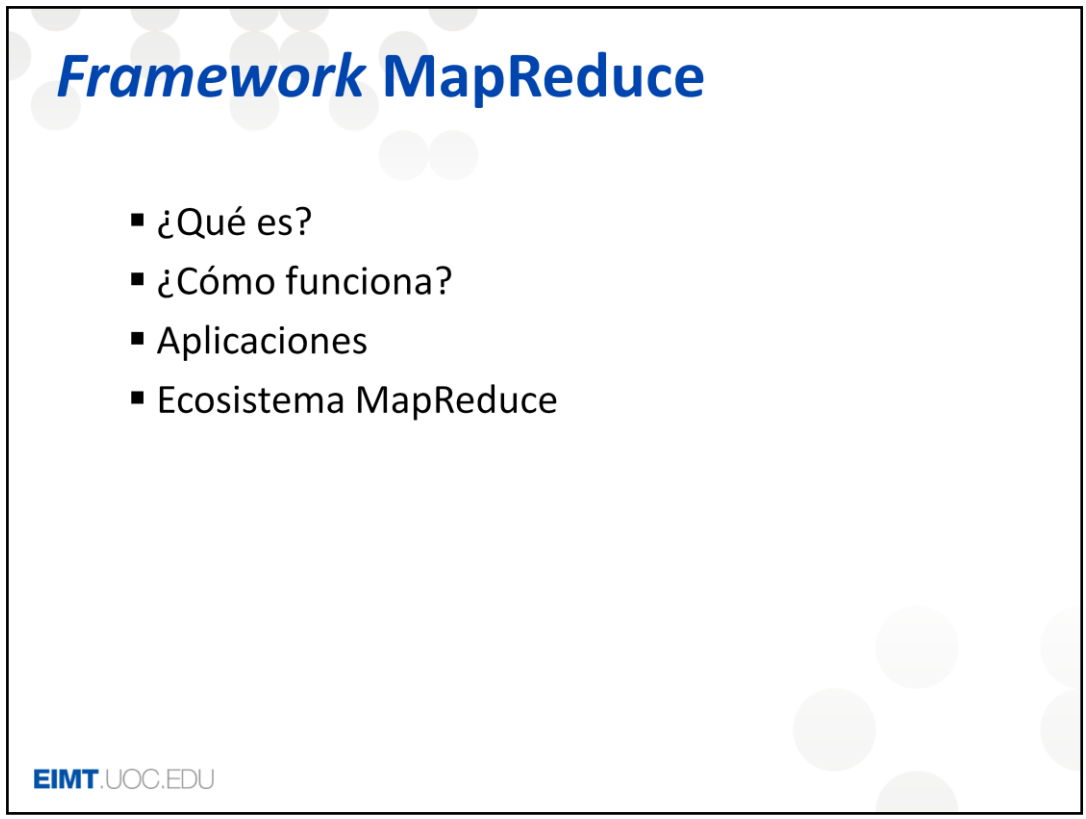
Por otro lado, para facilitar el uso de la infraestructura MapReduce, se han creado multitud de lenguajes que permiten crear procesos MapReduce de forma fácil y con un alto nivel de abstracción. Algunos de ellos son Pig (que permite realizar consultas de alto nivel con un lenguaje similar a SQL), Hive (que define un lenguaje llamado HQL-Hive Query Language que permite consultar datos desde una perspectiva MapReduce) y JaQL (lenguaje de consulta funcional y declarativo que facilita la explotación de información en formato JSON).



Pero MapReduce no sólo se usa como plataforma para el tratamiento de datos sino que también es utilizado como método de consulta y procesado en bases de datos. En el caso particular de NoSQL, MapReduce se utiliza principalmente para realizar consultas de datos agregados o consultas que requieran una transformación de datos que están distribuidos. Por ese motivo, las bases de datos NoSQL que siguen un modelo de agregación (clave-valor, documental y de columnas) acostumbran a facilitar la ejecución de operaciones de consulta/modificación de datos mediante procesos MapReduce.

A continuación, podemos ver un ejemplo en MongoDB que muestra cómo consultar el número de productos totales que ha comprado cada cliente utilizando MapReduce. Suponemos que los pedidos se almacenan en una colección de documentos denominada *orders*. Vemos que MongoDB simplifica enormemente el proceso, permitiendo definir el proceso de MapReduce en el contexto de una colección determinada (los pedidos) y definiendo de forma sencilla las operaciones de *map* y de *reduce*. En este caso, la función *map* devuelve, para cada pedido, el código de cliente y el total de productos comprados. Como se muestra en la figura, la función *reduce* recibiría, para cada cliente, la pareja <*código de cliente, valores*>, donde valores es la lista de los productos adquiridos en cada pedido. La lista de valores de cada cliente se suma para generar el resultado final de la función *reduce*. Finalmente, la sección *query output* indica que el resultado del proceso debe almacenarse en una colección llamada *order_totals*.

Tal y como muestra el ejemplo, las bases de datos NoSQL integran el *framework* MapReduce de forma nativa para facilitar el procesamiento de las consultas sobre datos distribuidos de forma fácil y eficiente.



Hasta aquí esta presentación introductoria al *framework* MapReduce. En ella hemos visto sus componentes y los aspectos más importantes en relación a su funcionamiento.

También hemos presentando un conjunto de ejemplos reales donde se usa MapReduce y hemos comentado algunas de las herramientas que lo implementan.

Referencias

C. Coronel, S. Morris & P. Rob (2013). *Database Systems: Design, Implementation and Management 10e*, Cengage Learning.

J. Dean, S. Ghemawat (2004). MapReduce: Simplified Data Processing on Large Clusters, "6th Symposium on Operating System Design and Implementation (OSDI'04)". (http://static.googleusercontent.com/media/research.google.com/es//archive/mapreduce-osdi04.pdf).

S. Ghemawat, H. Gobioff & S.T. Leung (2003). The Google File System, "19th ACM Symposium on Operating Systems Principles". (http://static.googleusercontent.com/media/research.google.com/es//archive/gfs-sosp2003.pdf).

I. Katsov (2012). "MapReduce Patterns", *Highly Scalable Blog. Articles on Big Data, NoSQL, and Highly Scalable Software Engineering*. (http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/)

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

T. White (2012). *Hadoop: The Definitive Guide*, 3rd Edition, O'Really Media.

EIMT.UOC.EDU

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.