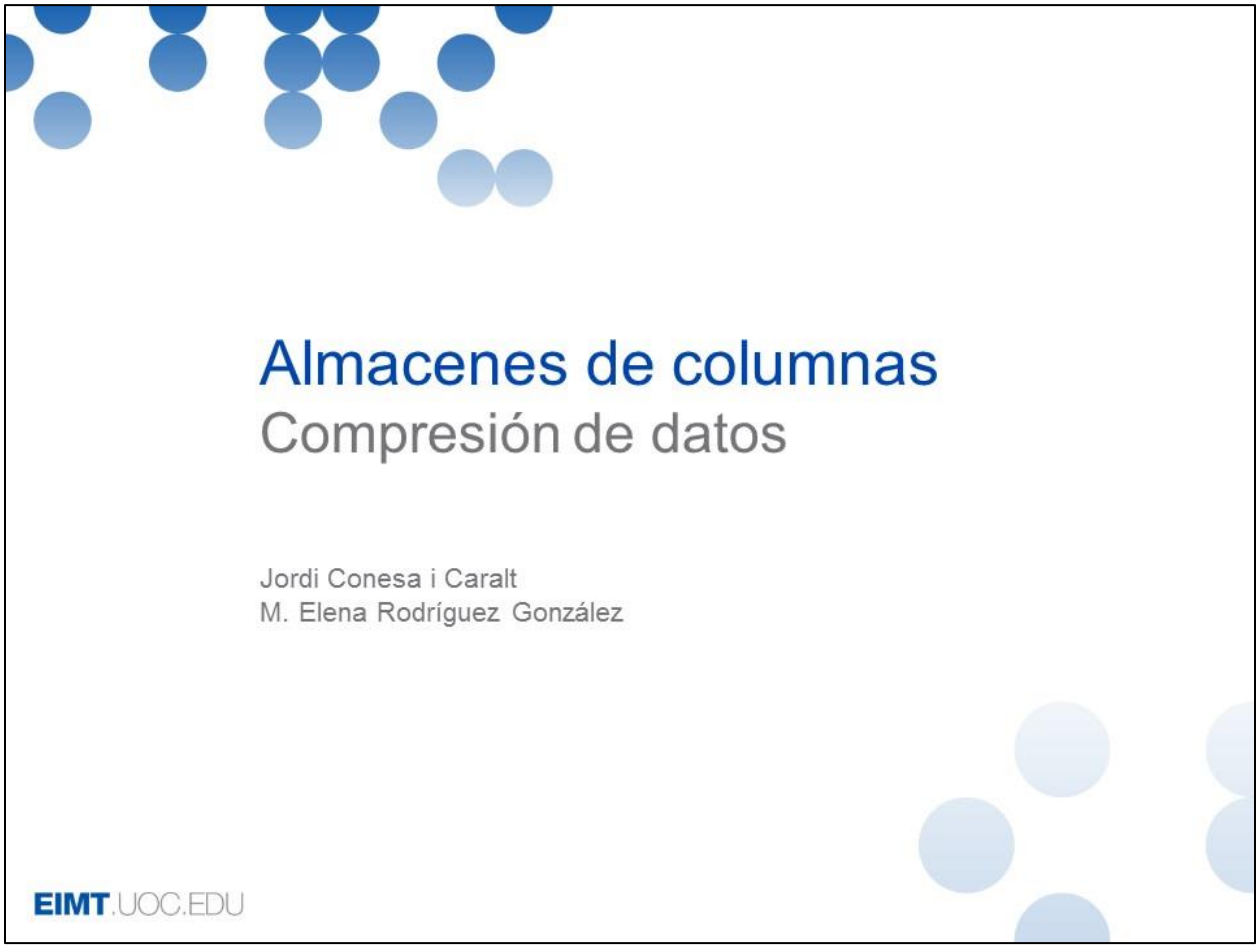
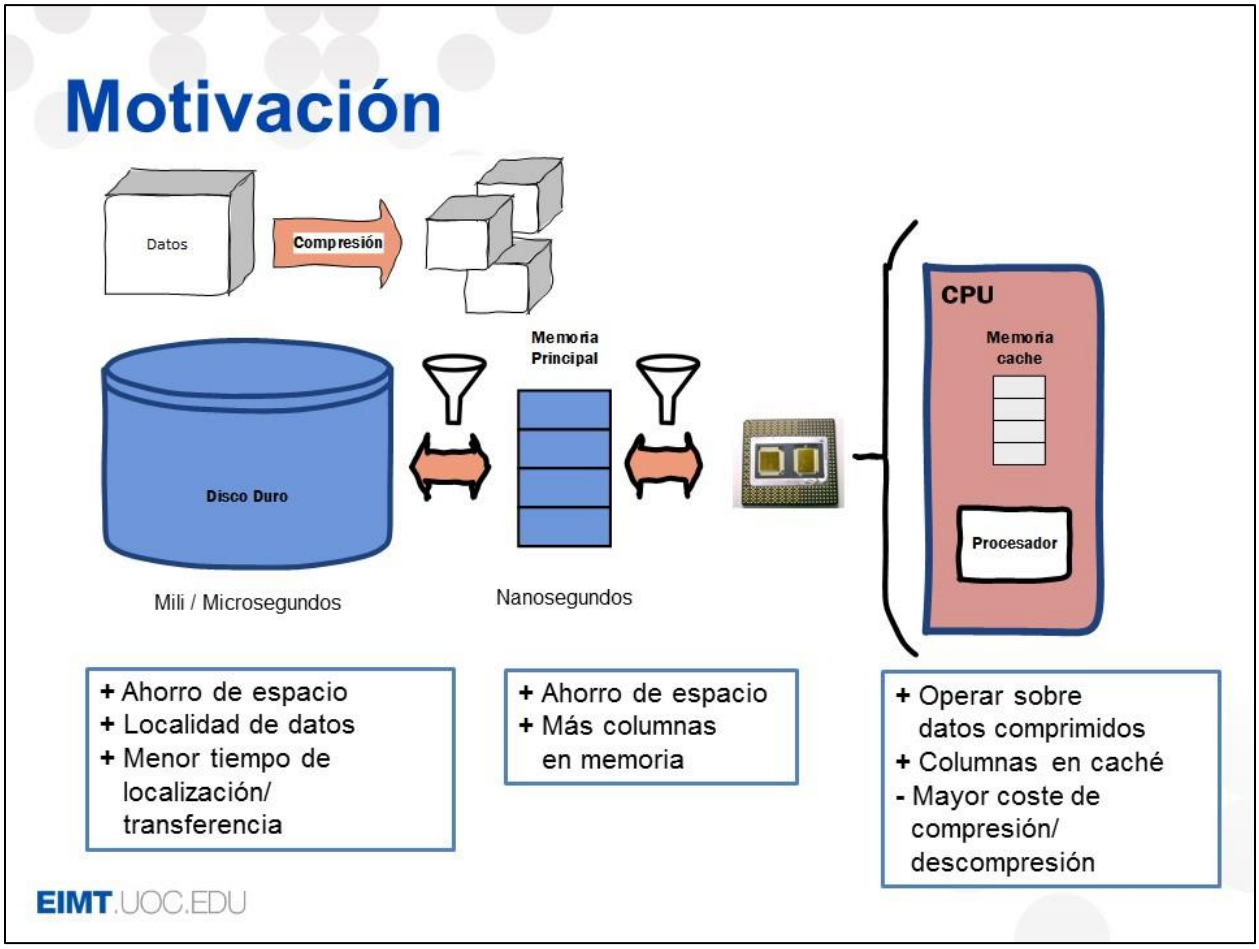


Bienvenidos a la tercera presentación de la asignatura de Bases de datos para *Data Warehouse* que trata temas sobre los almacenes de columnas.

La presentación está dividida en 5 partes:

- Motivación: empezaremos viendo el porqué de la compresión de datos y apuntando las principales razones por las que la compresión es más relevante en los almacenes de columnas que en los almacenes de filas.
- Tipos de compresión de datos: realizaremos una primera clasificación de los mecanismos/algoritmos de compresión de datos, en función de su coste e indicaremos qué tipo de algoritmos son, por norma general, los más adecuados en los almacenes de columnas.
- Algoritmos de compresión: introduciremos, mediante ejemplos, algunos de los algoritmos de compresión más utilizados (o relevantes) en los almacenes de columnas.
- Consideraciones de diseño: esbozaremos brevemente los conceptos clave a tener en cuenta para elegir el algoritmo de compresión más adecuado para cada caso.
- Ejemplos prácticos: hablaremos de la disponibilidad de los algoritmos presentados en los distintos Sistemas Gestores de Bases de Datos (SGBD) que almacenan los datos utilizando almacenes de columnas.



La compresión de datos es una de las principales características de los almacenes de columnas. La compresión también se usa en los almacenes de filas, pero de forma menos exhaustiva, y con resultados peores.

El motivo es que, comprimir los datos contenidos en las columnas de una tabla, suele dar mejores resultados (en otras palabras, se obtiene un mejor ratio de compresión) en los almacenes de columnas, ya que la entropía de las columnas es baja, dado que cada columna representa un mismo concepto del mundo real, y cada columna, además, se almacena en una estructura de almacenamiento (o fichero) separada. Es decir, los distintos valores de una misma columna son muy parecidos entre sí, y comparten características (por ejemplo, son del mismo tipo) y no se mezclan con datos de otras columnas que pueden tener una naturaleza diferente. Eso hace que, potencialmente, serán más y más fácilmente comprimibles. En cambio, en el caso de las filas eso no pasa, ya que una fila contiene conjuntos de valores distintos, con tipos diferentes y de longitud potencialmente distinta, es decir, cada uno de los valores contenidos en las filas representa conceptos diferentes del mundo real que tienen sus propias características. En los almacenes de columnas, en muchos casos, cuando los valores de una columna estén ordenados, se podrá obtener una mayor compresión ya que todos los valores iguales estarán juntos, es decir, estarán almacenados secuencialmente uno tras otro en el dispositivo de almacenamiento.

Alguien puede preguntarse, ¿por qué es necesario comprimir los datos si hoy en día el precio del almacenaje es muy bajo? El objetivo no es tanto ahorrar en disco duro, sino minimizar el número de operaciones de E/S, es decir, minimizar el número de operaciones de lectura y escritura al mismo. El verdadero cuello de botella en los SGBD actuales está en los accesos a disco (e incluso en los accesos a memoria principal). Hay que tener en cuenta que en el mejor de los casos (con los discos duros más rápidos) la lectura de datos en disco es del orden de 1.000 veces más lenta (del orden de mili o microsegundos) que la lectura de datos en memoria principal (que es del orden de nanosegundos). Si comprimimos una columna, reduciremos el número de páginas que se deben leer (o escribir) para obtener (o almacenar) todos sus valores y eso ahorrará operaciones de E/S a disco, mejorando así la eficiencia. Además, en algunos casos, la compresión nos permitirá almacenar completamente una (o más) columnas en la memoria caché del procesador, mejorando aún más el rendimiento del SGBD.

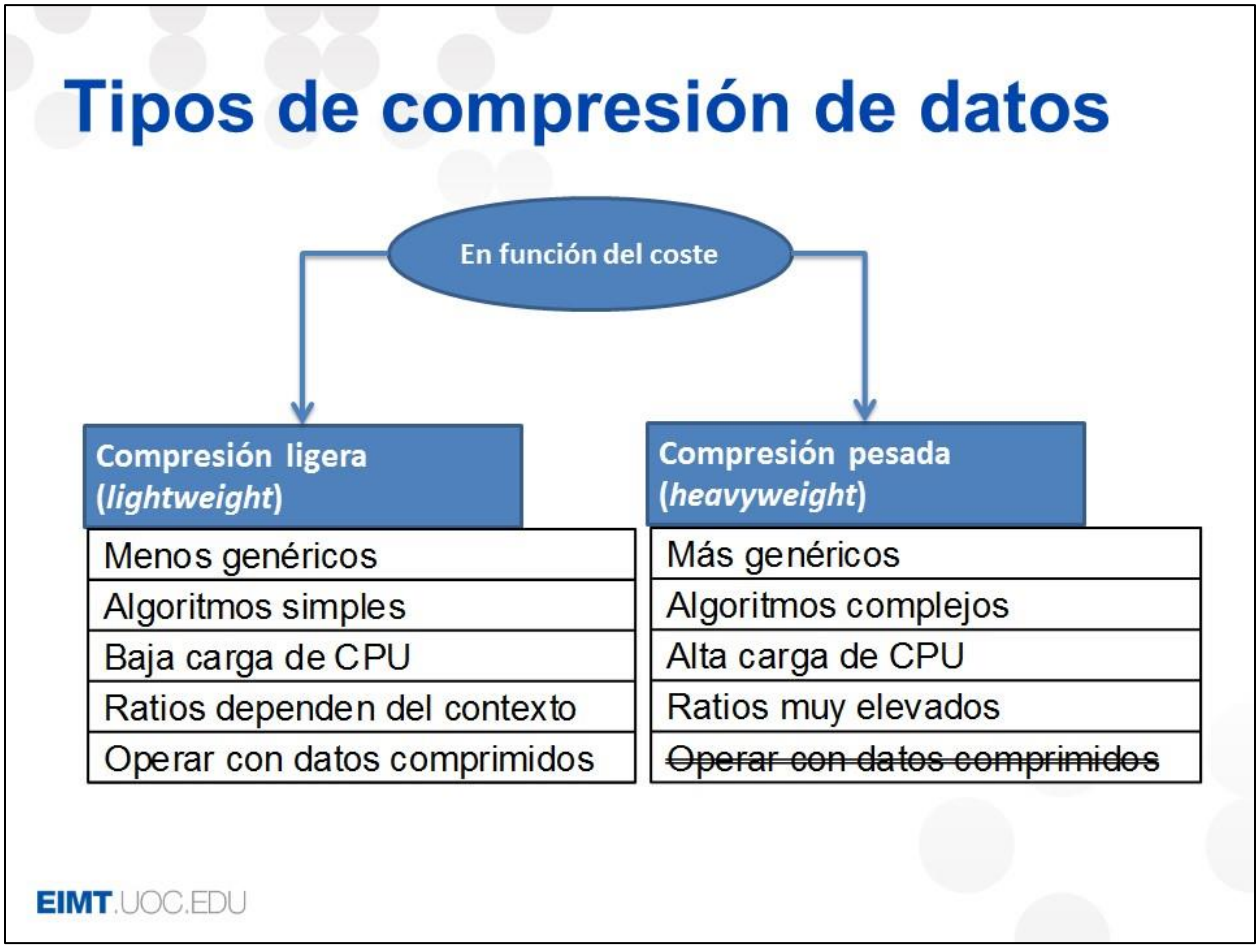
Otra ventaja es que, en algunos casos, es posible realizar las operaciones sobre la BD (por ejemplo, combinaciones, operaciones de selección/filtros, etc.) directamente sobre los valores comprimidos. Eso permitirá incrementar aún más la eficiencia del sistema, ya que se podrán ejecutar las mismas operaciones iterando sobre menos valores (ya que, al estar comprimida la columna, habrá potencialmente menos valores a procesar) en memoria principal o en la caché (ya que la columna ocupa menor tamaño), y se evitará descomprimir los datos antes de trabajar con ellos.

El ahorro de espacio provocado por la compresión de columnas suele ser aprovechado por los SGBD para materializar nuevas estructuras de datos o copias de datos auxiliares, por ejemplo, para almacenar una misma columna varias veces con ordenaciones distintas, para almacenar diferentes proyecciones de una tabla, etc. Este almacenamiento extra permite preparar mejor los datos con el objetivo de responder a distintas consultas de forma más eficiente.

El inconveniente principal de la compresión de datos es que comprimir y descomprimir los valores de una columna requiere tiempo adicional, ya que el procesador (o sea, la CPU) deberá invertir ciclos extra para comprimir/descomprimir los datos. Estos ciclos extra tanto son necesarios en la resolución de las operaciones de lectura (`SELECT`) como de escritura (`INSERT`, `DELETE`, `UPDATE`) sobre la BD, pero serán potencialmente superiores en el caso de las de escritura. De todas maneras, y como acabamos de comentar, en ocasiones el SGBD puede trabajar directamente sobre datos comprimidos, con lo que (en parte) se mitiga este inconveniente.

En cualquier caso, y a modo de resumen, la compresión será conveniente cuando los ciclos extra del procesador requeridos para la compresión/descompresión se vean compensados por el ahorro de operaciones de E/S a disco. En algunas situaciones esa condición puede no cumplirse, como sería el caso de los algoritmos de compresión más potentes, la compresión LZH, ZIP o RAR, que requieren una carga de procesador que puede superar al ahorro en operaciones de E/S a disco.

A continuación clasificaremos distintos algoritmos de compresión de datos en función de su coste computacional. Para cada tipo de algoritmo describiremos, brevemente, sus principales características y sus ventajas y desventajas al aplicarlos sobre almacenes de columnas.



Los sistemas de compresión se pueden clasificar en ligeros (*lightweight* en inglés) o pesados (en inglés, *heavyweight*), en función del coste que implica comprimir y descomprimir los datos.

Los sistemas de compresión ligera son aquellos que utilizan algoritmos simples y, por lo tanto, son sistemas que no requieren de una elevada carga de CPU (o sea de procesador) para comprimir y descomprimir los datos. En consecuencia, el ratio de compresión de estos algoritmos no es tan alto como en el caso de los algoritmos de compresión pesada. Además, el ratio de compresión suele depender mucho de cada algoritmo y del contexto en el que se aplique. Así, algunos algoritmos serán muy útiles cuando haya valores que se repiten muy frecuentemente, otros serán útiles para comprimir valores largos o textuales, y otros serán más adecuados cuando el espacio (o rango) de valores de la columna posibles sea reducido. Otra ventaja de utilizar compresión ligera es que, en algunos casos, el SGBD puede operar directamente sobre los datos comprimidos, ahorrándose así el tiempo de descomprimir los valores, reduciendo los datos sobre los que se realizan las operaciones, y minimizando el traspaso de datos a lo largo de la jerarquía de memoria (disco/memoria/caché).

Por su parte los sistemas de compresión pesados son aquellos que incrementan el ratio de compresión a costa de penalizar el tiempo de compresión y descompresión. Algunos ejemplos son Lempel-Ziv (o LZH), ZIP o RAR. Estos algoritmos suelen ofrecer elevados ratios de compresión, independientemente del contexto donde se apliquen. Por lo tanto, son la mejor solución para minimizar el tamaño de los datos.

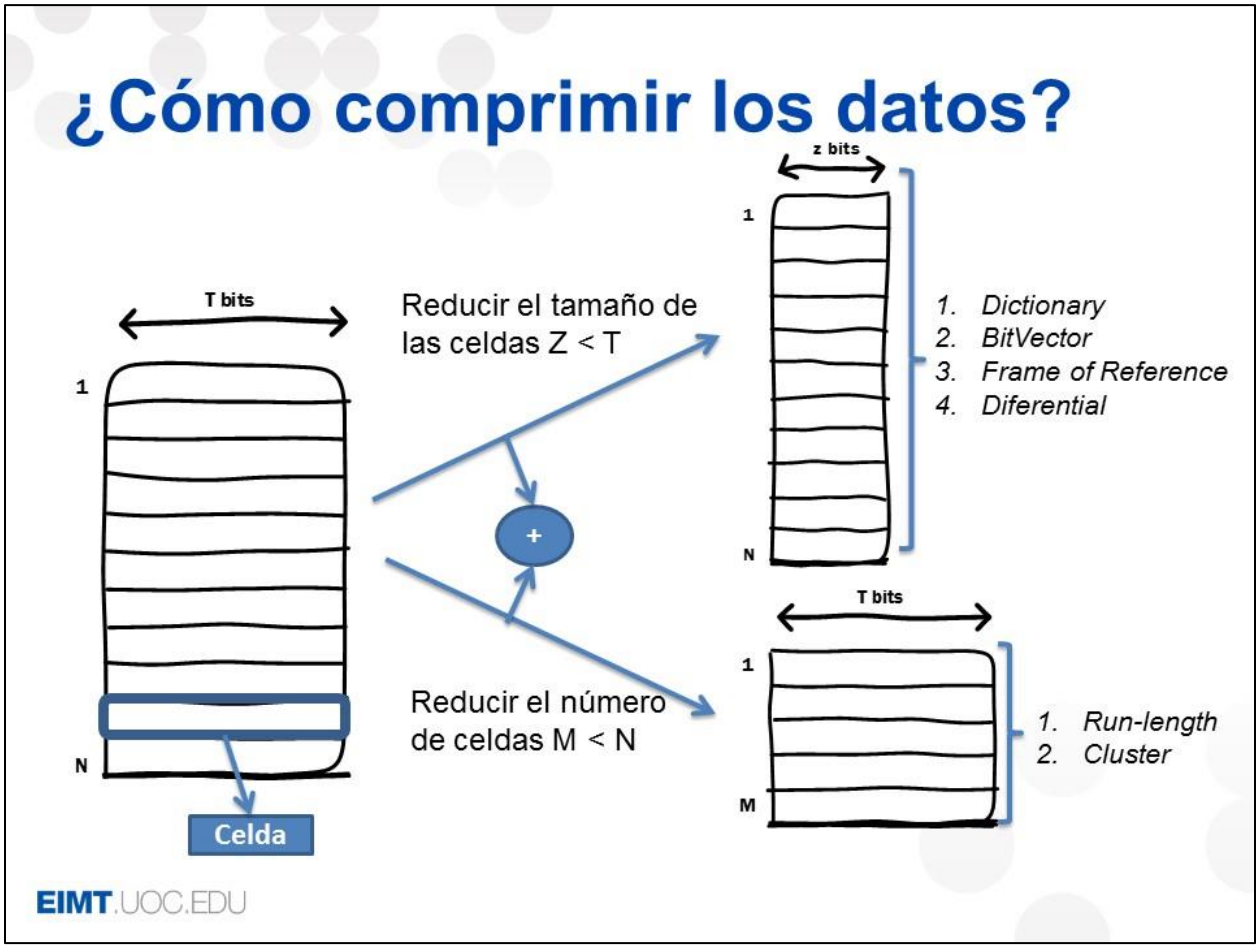
No obstante, su alto coste computacional, el hecho de que no se pueda descomprimir parcialmente un conjunto de valores (por ejemplo, una página de la BD que contiene una parte de una columna) y el hecho de que no se pueda operar directamente sobre los datos comprimidos los hacen una opción poco utilizada en los almacenes de columnas. Aunque estos inconvenientes parecen insalvables, quizá los nuevos cambios en *hardware* permitan utilizar estos algoritmos a corto/medio plazo. Hay estudios que demuestran que, para algunas consultas, se pueden utilizar algoritmos de compresión pesados, como sería el caso del LZH, con resultados satisfactorios (para más información podéis consultar el capítulo 4 de la tesis doctoral de Daniel Abadi, se trata de la primera referencia que tenéis al final de esta presentación). En dicho estudio se argumenta que, con una buena implementación del algoritmo LZH, se tarda menos en leer los datos comprimidos y descomprimirlos, de lo que se tardaría en leer todos los datos no comprimidos.

En esta material didáctico, y debido al uso extensivo que tienen en los almacenes de columnas, sólo trataremos con los algoritmos de compresión ligera.

En este apartado presentaremos, mediante ejemplos, algunos algoritmos de compresión. Hemos escogido los más relevantes por su uso o por su funcionamiento. Para facilitar la separación de los contenidos en distintos vídeos, se han clasificado los algoritmos en dos tipos, en función de cómo comprimen los datos, en concreto:

1) Si lo hacen reduciendo el número de valores (o celdas) de una columna.
2) Si lo hacen reduciendo el espacio que ocupa cada valor (o celda) de la columna.

Cabe señalar que cada uno de los algoritmos presentados aquí puede tener numerosas variantes y adaptaciones. Por lo tanto, antes de empezar a trabajar con un almacén de columnas, siempre será necesario (o al menos muy aconsejable) analizar qué algoritmos proporciona el SGBD y cómo los implementa.

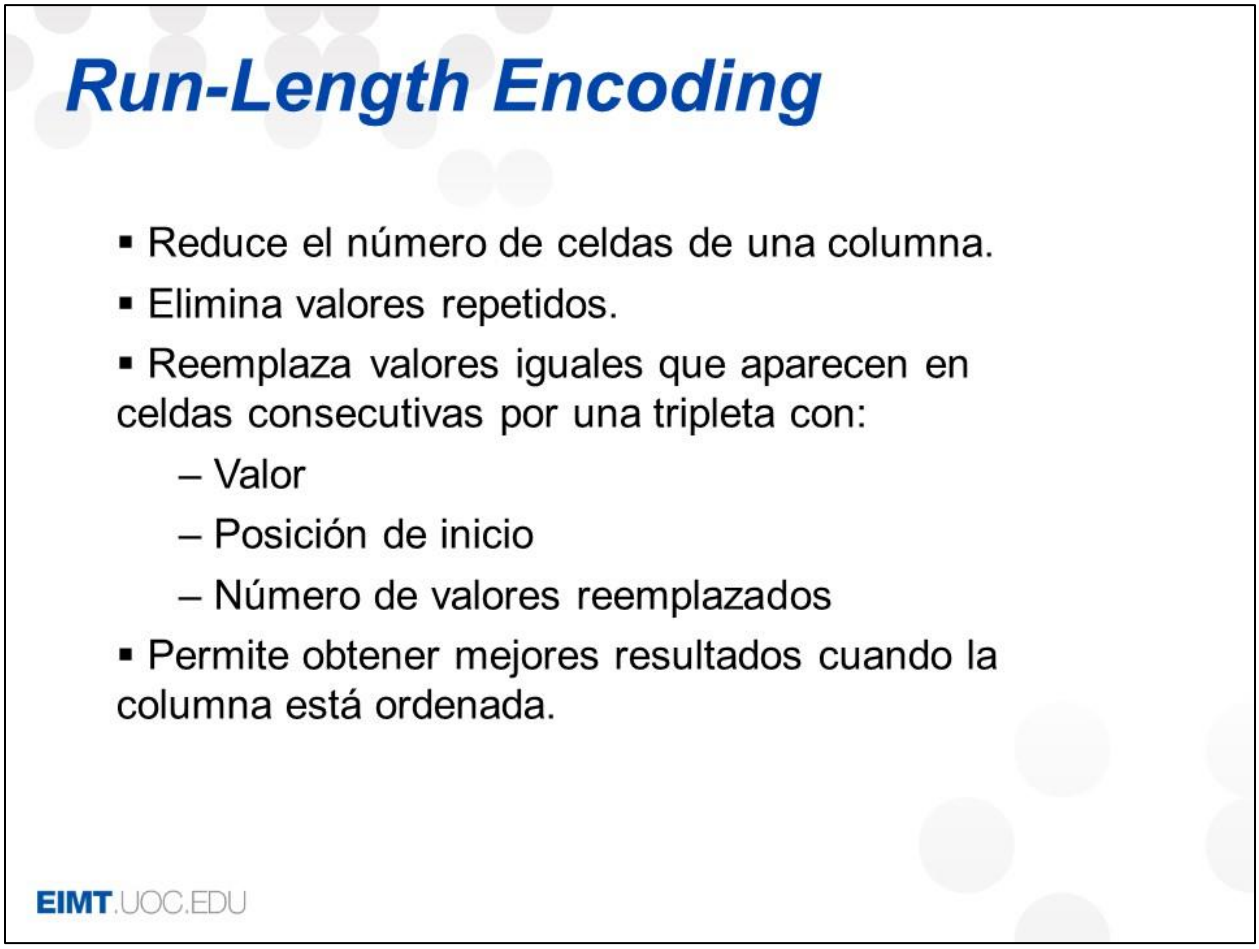


El espacio que ocupa una columna está principalmente condicionado por el número de valores (o celdas) que contiene (indicado mediante la letra *N* en la diapositiva) y el tamaño de cada celda (indicado con la letra *T* en la diapositiva). En consecuencia, el tamaño total de una columna se obtendrá a partir de la siguiente fórmula (para simplificar la explicación, y sin pérdida de generalidad, hemos obviado los metadatos, es decir, no consideramos los atributos que pudieran guardar información de control):

Tamaño_columna = número_de_valores x tamaño_de_cada_valor

Para reducir el tamaño de una columna se puede reducir cualquiera de los dos operandos (*número_de_valores* y/o *tamaño_de_cada_valor*) que intervienen en la fórmula previa. Es decir, se puede reducir el tamaño haciendo que la columna tenga menos celdas (o sea que almacene menos valores) o haciendo que cada celda ocupe menos espacio. También se podrán comprimir los datos realizando ambas tareas a la vez, o sea combinando distintos algoritmos de compresión, por ejemplo, utilizando conjuntamente los algoritmos *Dictionary Encoding* y *Run-Length Encoding*, como veremos más adelante en esta misma presentación.

En este material vamos a estudiar algoritmos que comprimen una columna reduciendo su número de celdas, como sería el caso del *Run-Length Encoding* y el *Cluster Encoding*, o reduciendo el tamaño de sus celdas, como sería el caso de los algoritmos *Dictionary Encoding, BitVector Encoding*, el *Frame of Reference Encoding* y el *Differential Encoding*.



Este algoritmo comprime los datos de una columna eliminando gran parte de los valores repetidos de la misma. Por lo tanto, su compresión se basa en la reducción del número de celdas de una columna.

En particular, el algoritmo sustituye las celdas consecutivas que tienen valores iguales por una única celda que contiene una tripleta con los siguientes campos (o metadatos):

(valor, inicio, número_de_ocurrencias)

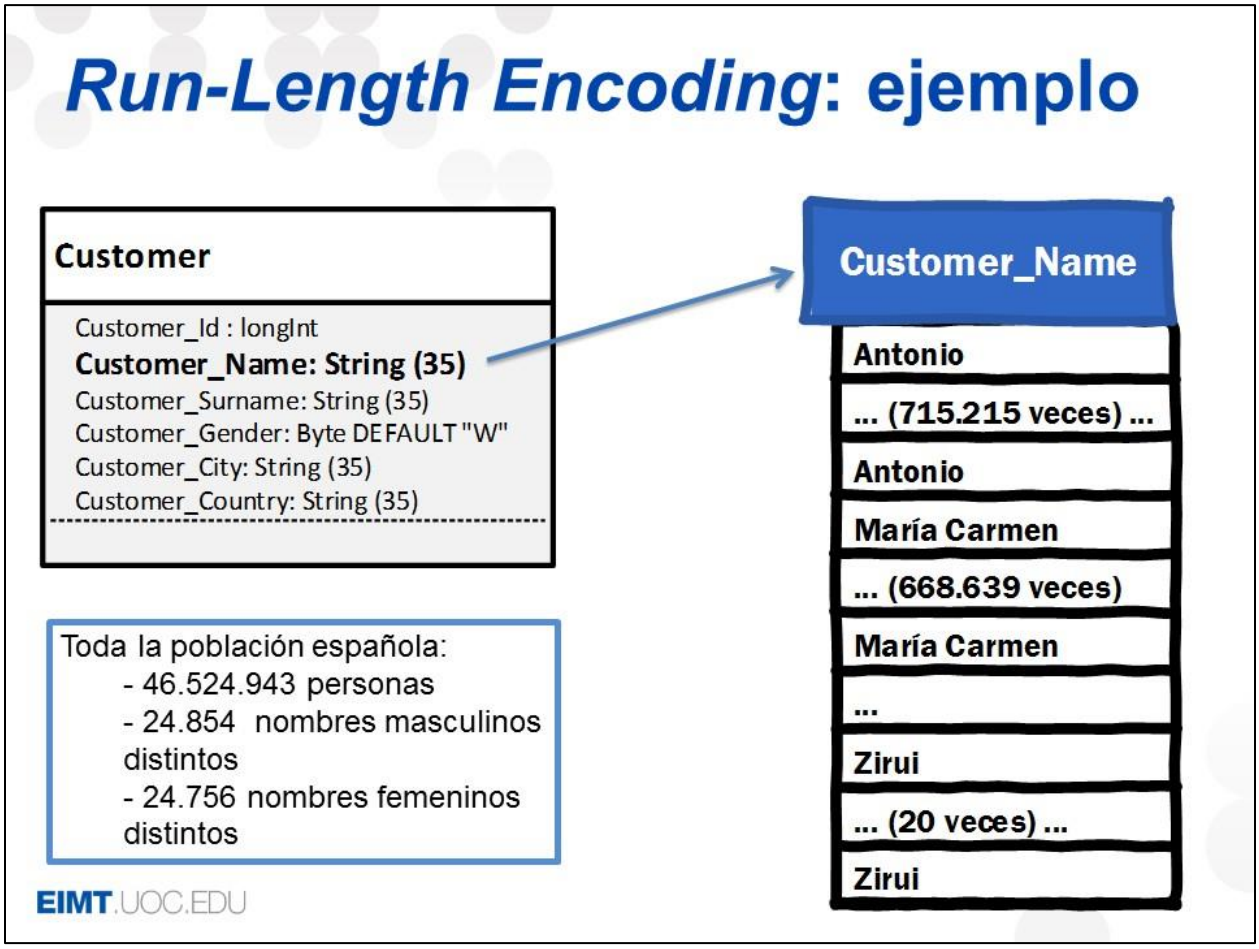
Donde *valor* indica el valor substituido, *inicio* la posición de la primera celda donde se ha encontrado la repetición y *número_de_ocurrencias* indica el número de veces que el valor estaba consecutivamente repetido. Existen variaciones del algoritmo que almacenan otros tipos de metadatos.

Este algoritmo reduce el número de celdas de la columna, pero incrementa el tamaño de las mismas, ya que tiene que añadir los metadatos *inicio* y *número_de_ocurrencias*.

Este algoritmo funciona de forma eficiente cuando hay pocos valores posibles con un número elevado de ocurrencias. Como puede intuirse, el algoritmo generará mejores resultados cuando los valores de la

columna, además, estén ordenados. El motivo es que, en ese caso, todos los valores iguales estarán almacenados en celdas consecutivas, y el resultado de utilizar la compresión será obtener una nueva columna con tantas celdas como valores distintos haya.

El algoritmo se puede aplicar tanto de forma aislada, como conjuntamente con otros. Éste sería el caso, por ejemplo, del algoritmo *Dictionary Encoding*.



Vamos a ver cómo funciona el algoritmo de *Run-Length Encoding* mediante un ejemplo.

Recordemos el ejemplo que hemos estado utilizando en estos materiales dedicados a los almacenes de columnas: un pequeño *datamart* que contiene información de ventas de una empresa en función de la fecha de venta, los productos vendidos y el cliente. El modelo del ejemplo tenía una tabla de hechos, la tabla *Sale* y tres dimensiones de análisis: la fecha (*Date*), a quién se vendió (*Customer*) y qué se vendió (*Product*). Es importante destacar que el esquema de la BD puede haber sufrido alteraciones con respecto a presentaciones previas, con el objetivo de buscar ejemplos más adecuados a lo que se pretende explicar. En este primer ejemplo, vamos a comprimir la columna *Customer_Name* de la tabla *Customer*. Esta columna guarda el nombre de pila del cliente, mientras que la columna *Customer_Surname* guarda el apellido (o apellidos) del cliente.

Para tener un volumen significativo de datos y mostrar la potencialidad de la compresión con datos reales, supondremos que nuestra empresa ha realizado ventas a toda la población española. En consecuencia, la tabla contendrá 46.524.943 filas, una por cada persona residente en España (según datos del Instituto Nacional de Estadística –INE– a principios del 2016).

La columna *Customer_Name* contendrá los nombres de dichas personas. Según el INE (ver http://www.ine.es/daco/daco42/nombyapel/nombyapel.htm) en España existe un total de 24.854 nombres

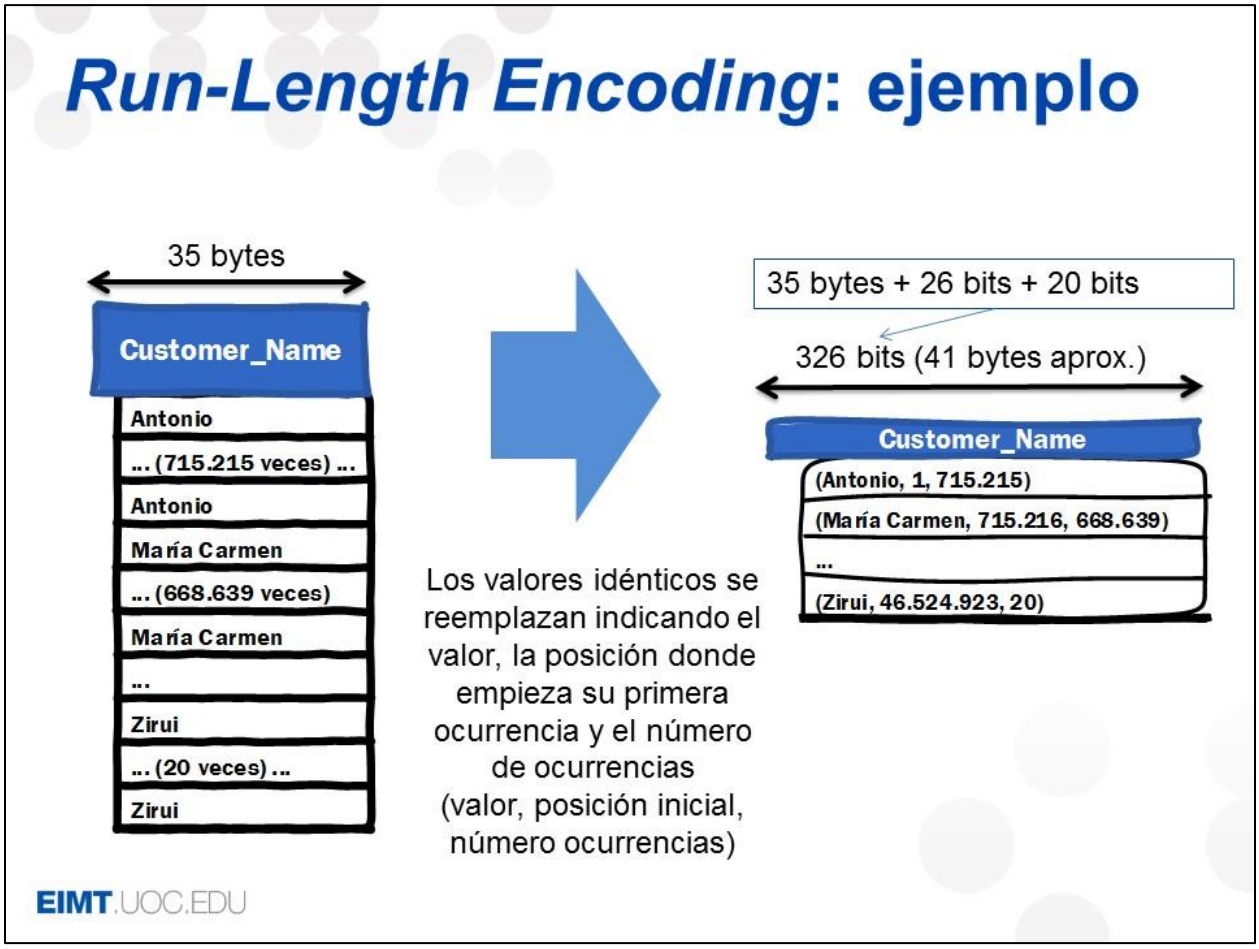
masculinos y 24.756 nombres femeninos distintos[1]. Por lo tanto, habrá muchas repeticiones de nombres, 715.215 repeticiones de *Antonio*, 668.639 repeticiones del nombre *María Carmen*, 20 repeticiones del nombre *Zirui*, etc. Supondremos también que la columna está ordenada en función del número de ocurrencias de los nombres de los clientes[2].

En la transparencia podemos ver una representación gráfica de la columna de *Customer_Name*. Por convención, utilizaremos trazos manuales (como escritos a mano) para indicar los valores de las columnas.

Una vez presentado el contexto, vamos a ver cómo el algoritmo *Run-Length Encoding* comprime esta columna.

[1] Hay que tener en cuenta que sólo se tienen en cuenta nombres con una frecuencia de 20 o más apariciones.

[2] Es decir, primero estará *Antonio*, que es el nombre que comparten más clientes, después *María Carmen*, que es el segundo más común, y así consecutivamente. No es un orden muy común ni útil, pero es para ejemplificar mejor el algoritmo.



En el contexto descrito, nos encontramos con el mejor escenario. Al estar ordenados los valores de la columna, todas las celdas con valores iguales estarán en posiciones consecutivas.

El algoritmo de compresión creará una nueva columna donde se almacenarán los valores comprimidos. El espacio de las celdas de la nueva columna deberá ser mayor, ya que cada celda deberá almacenar:

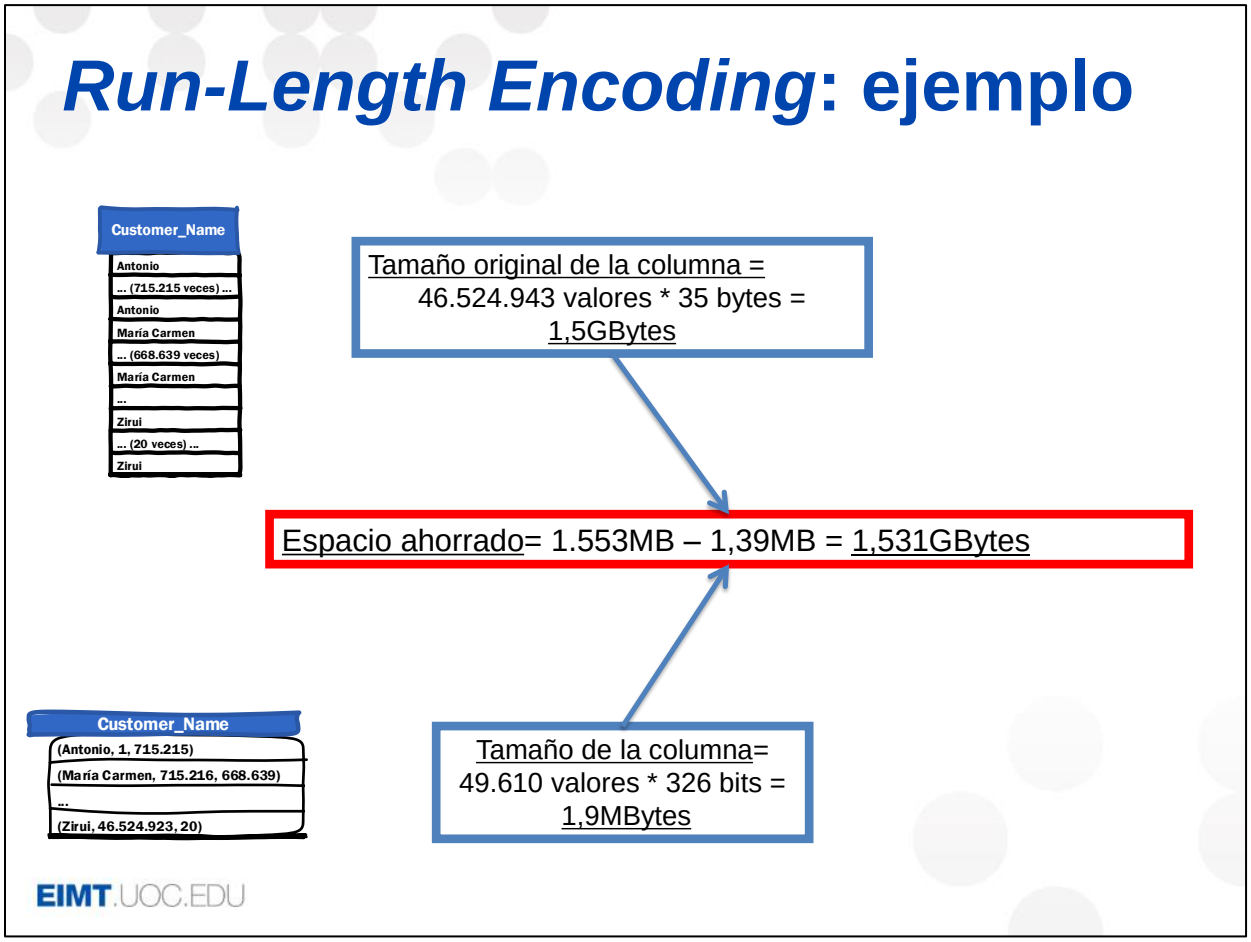
1. Un valor que ocupa 35 bytes.
2. La posición del valor en la columna original, que puede tomar valores entre 1 y 46.524.923. Se necesitan 26 bits para representar este número.
3. El número de ocurrencias que, según los datos estadísticos del INE, son como máximo 715.215. Se necesitan 20 bits para representar estos valores.

Por lo tanto, las celdas de la columna comprimida ocuparán 326 bits (las de la columna original ocupaban 280).

Para comprimir los datos se consultaría la columna original de forma secuencial. Para cada celda de la columna (es decir, para cada posición *i*) se realizaría lo siguiente:

1. Se lee el valor de la celda *i*:
   - Si la celda siguiente (*i+1*) tiene un valor distinto, entonces:
     - En la columna original se cumplirá $valor_{i+1} \neq valor_i$.
     - Se asigna a la columna comprimida la tripleta $(valor_i, i, 1)$.
     - La siguiente celda a leer será la que esté en la posición *i+1*.
   - Si la celda siguiente (*i+1*) tiene el mismo valor, se consultan y se cuentan las celdas consecutivas a *i* que tengan el mismo valor (supongamos que hay *k* celdas consecutivas con el mismo valor). Entonces:
     - En la columna original se cumplirá $valor_{i+k} = valor_i$.
     - En la columna original se cumplirá $valor_{i+k+1} \neq valor_i$.
     - Se asigna a la columna comprimida la tripleta $(valor_i, i, k)$.
     - La siguiente celda a leer será la que esté en la posición *i+k+1*.
2. Si no hemos llegado al final de la estructura que almacena los datos de la columna, volvemos al paso 1.

La aplicación del algoritmo en el ejemplo de la transparencia es bastante simple porque todos los valores tienen repetición y están ordenados por el número de ocurrencias. En este caso, se empezaría a comprimir la columna original desde el principio. Al estar ordenada la columna, el valor de la primera celda (*Antonio*) estaría repetido *715.216* veces (recordemos que es el número de personas con dicho nombre según el INE). En consecuencia, crearíamos una nueva tripleta en la columna comprimida que tenga el valor *Antonio,* que indique que la primera ocurrencia de dicho valor está en la celda número *1* de la columna y que dicho valor se encuentra repetido *715.216* veces. Vemos que el algoritmo ha permitido comprimir 715.216 ocurrencias de un valor por una (1) sola ocurrencia del mismo a cambio de añadir unos pocos metadatos. Posteriormente, se haría lo mismo con el siguiente valor de la columna (*María Carmen*). En este caso tenemos 668.639 ocurrencias de este nombre. Todas ellas se sustituirán en la columna comprimida por una nueva tripleta que contiene el valor *María Carmen*, su primera ocurrencia (la *715.216*, ya que viene después de los 715.216 Antonios) y el número de ocurrencias de la misma (*668.639*). El algoritmo iría ejecutando estas sustituciones, nombre a nombre, hasta llegar al último nombre de la columna: *Zirui*, que se sustituiría por la tripleta (*Ziriu*, *46.524.923* –posición donde aparece por primera vez–, *20* –número de ocurrencias consecutivas del nombre–).



Como hemos comentado, las celdas de la nueva columna comprimida ocupan más espacio, por lo tanto, el ahorro de espacio de almacenaje viene por la reducción del número de celdas. A continuación vamos a ver el resultado de la compresión en el ejemplo realizado. Veremos qué ocupaba la columna inicial, qué ocupa la columna comprimida, y qué ganancia (en términos de ahorro en espacio de almacenamiento) hemos obtenido.

El tamaño de la columna original es de 1,5 GBytes. Se puede calcular multiplicando el número de celdas de la columna (46.524.943) por el tamaño de cada celda (35 bytes):

Tamaño Original(Customer_Name) = 46.524.943 valores x 35 bytes =
1.628.373.005 bytes ≈ 1.553 MBytes

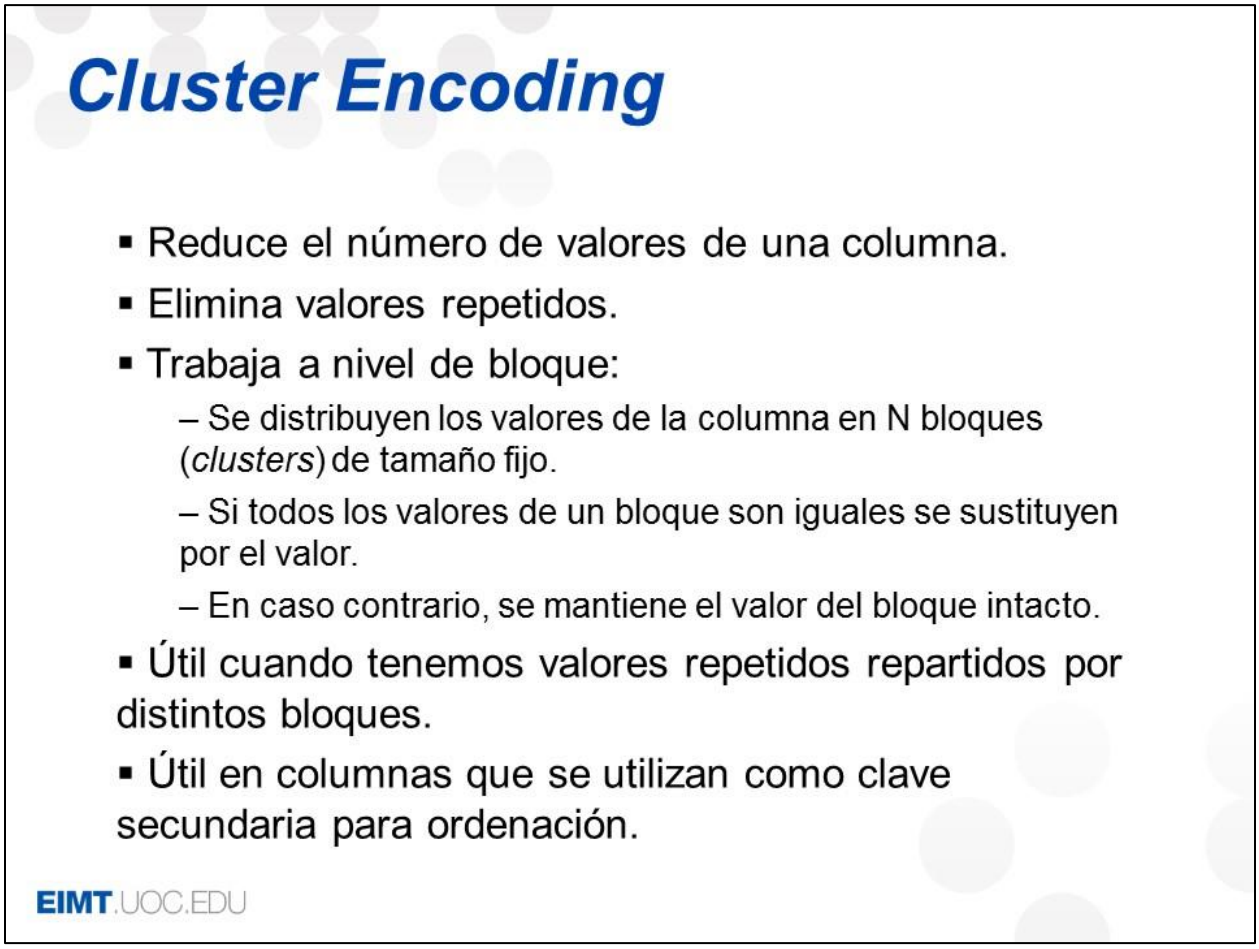
El tamaño de la nueva columna es de cerca de 2 Mbytes. Se calcula de la misma forma que antes, sólo que ahora tendremos 49.610 celdas y cada celda ocupará 326 bits.

Tamaño comprimido(Customer_Name) = 49.610 valores x 326 bits =
11.707.960 bits ≈ 1,39 MBytes

En consecuencia, el ahorro final vendrá dado por la diferencia entre el tamaño inicial y el comprimido:

Ahorro(Customer_Name) = 1.553 Mbytes – 1,39 Mbytes = 1.531,61 MBytes

Podemos observar que el ahorro obtenido en este caso es muy bueno, porque los datos estaban ordenados y todos los posibles valores tenían repeticiones. En el supuesto de que la columna hubiera contenido el nombre y el apellido de los clientes concatenado (tal y como sucedía en la tabla *Customer* usada en las presentaciones previas), hubiera habido menos repeticiones y los resultados habrían sido peores. Por lo tanto, escoger cómo distribuir los datos en distintas columnas de una tabla puede tener efectos muy significativos en el resultado (o ratio final) de compresión de datos.



El algoritmo *Cluster Encoding* comprime los datos de una columna eliminando gran parte de los valores repetidos de la misma. En consecuencia, su compresión también se basa en la reducción del número de celdas de una columna.

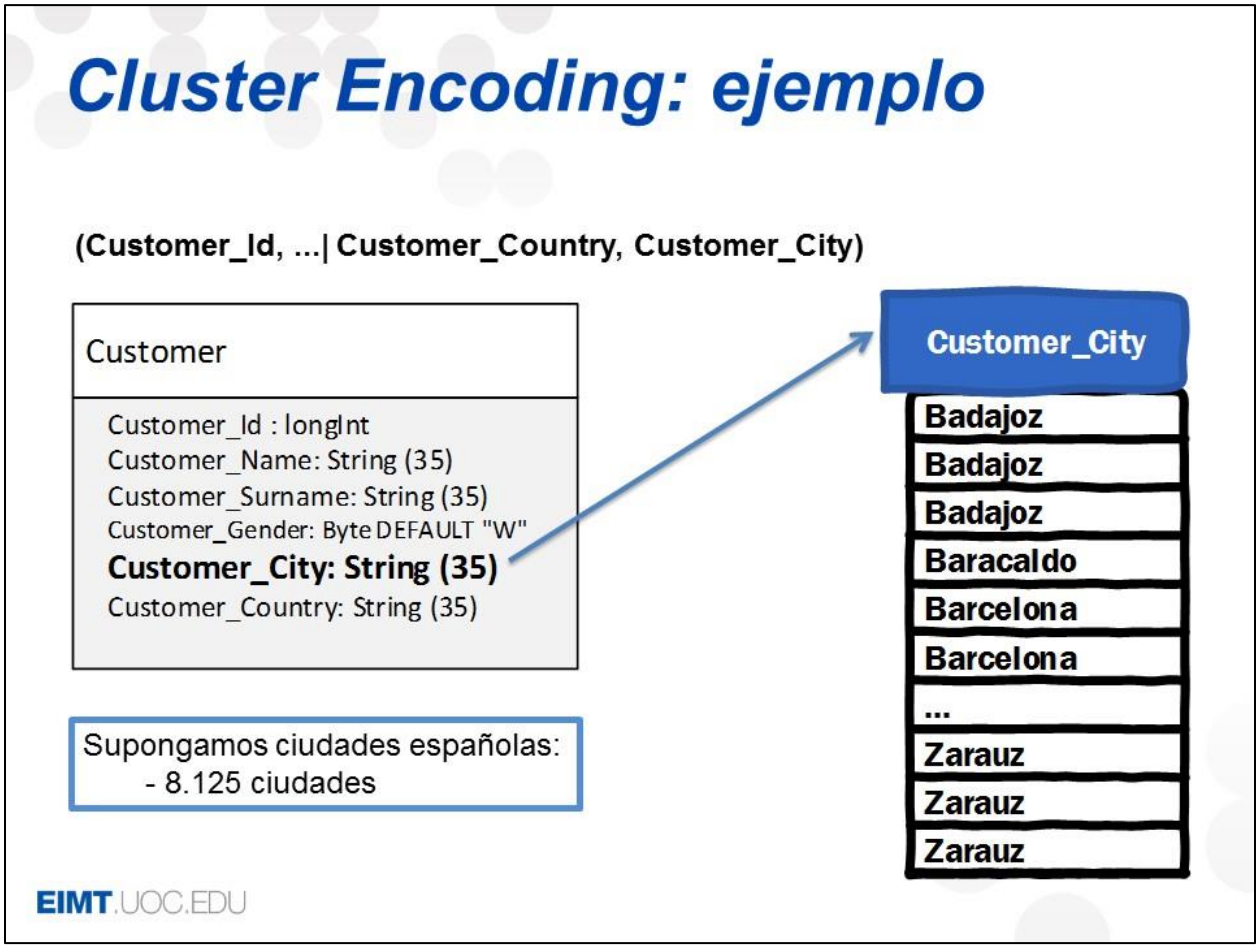
Funciona de forma similar al algoritmo *Run-Length Encoding*, pero trabajando a nivel de *cluster* o bloques de celdas, en vez de trabajar a nivel de celdas individuales. El algoritmo divide una columna en *N* bloques del mismo tamaño. Normalmente, el tamaño del bloque es de 1.024 celdas. Para cada bloque de la columna original se comprueba si todos los valores de sus celdas son iguales. En caso afirmativo, se sustituye el bloque entero por una entrada en la columna comprimida con el valor repetido. En el caso de que no todas las celdas del bloque tengan el mismo valor, se copia el bloque original en la columna comprimida. Para tener constancia de qué bloques se han comprimido y qué bloques no se han comprimido, se guardará un vector de bits con tantas posiciones como bloques existan. En cada posición *i* del vector se indicará si el bloque *i-ésimo* ha sido comprimido (esto se indicará asignando un valor *1* a la posición) o no (en este caso se asignará un *0* a la posición que representa el bloque).

En este algoritmo es importante escoger bien el tamaño del bloque, en función del número esperado de repeticiones consecutivas que nos podamos encontrar en la columna a comprimir. En caso contrario, la ganancia de compresión puede ser poca, ya que sólo con que haya un valor distinto en un bloque, éste no se comprimirá.

Este algoritmo será especialmente útil cuando los datos repetidos estén agrupados en distintas partes de la columna. Éste sería el caso, por ejemplo, de que la columna fuese usada como clave secundaria en una ordenación. Recordemos que las tablas de un almacén de columnas pueden definir claves de ordenación formadas por una o varias columnas, y que en ocasiones (en el caso de que se permitan definir proyecciones) diversas columnas de la tabla se pueden agrupar y reordenar sus valores de acuerdo a otras claves de ordenación. En el caso de que la clave de ordenación incluya diversas columnas, la primera columna es la clave primaria de la ordenación, mientras que el resto de columnas son las claves secundarias de ordenación.

Supongamos que tenemos una columna que determina el país de los clientes, y otra que determina su ciudad (serían las columnas *Customer_City* y *Customer_Country* de nuestra tabla *Customer*). Si ordenamos los datos en primera instancia por país (esta columna sería la clave primaria en la ordenación) y luego por ciudad (esta columna sería la clave secundaria en la ordenación), tendremos que todas las ciudades del mismo país aparecerán consecutivas en la columna *Customer_City*. Por ejemplo, el valor *Madrid* aparecerá consecutivo en todos los clientes españoles, pero aparecerá también consecutivo en otra parte de la columna para los clientes de Colombia, ya que en ese país también existe una ciudad denominada Madrid.

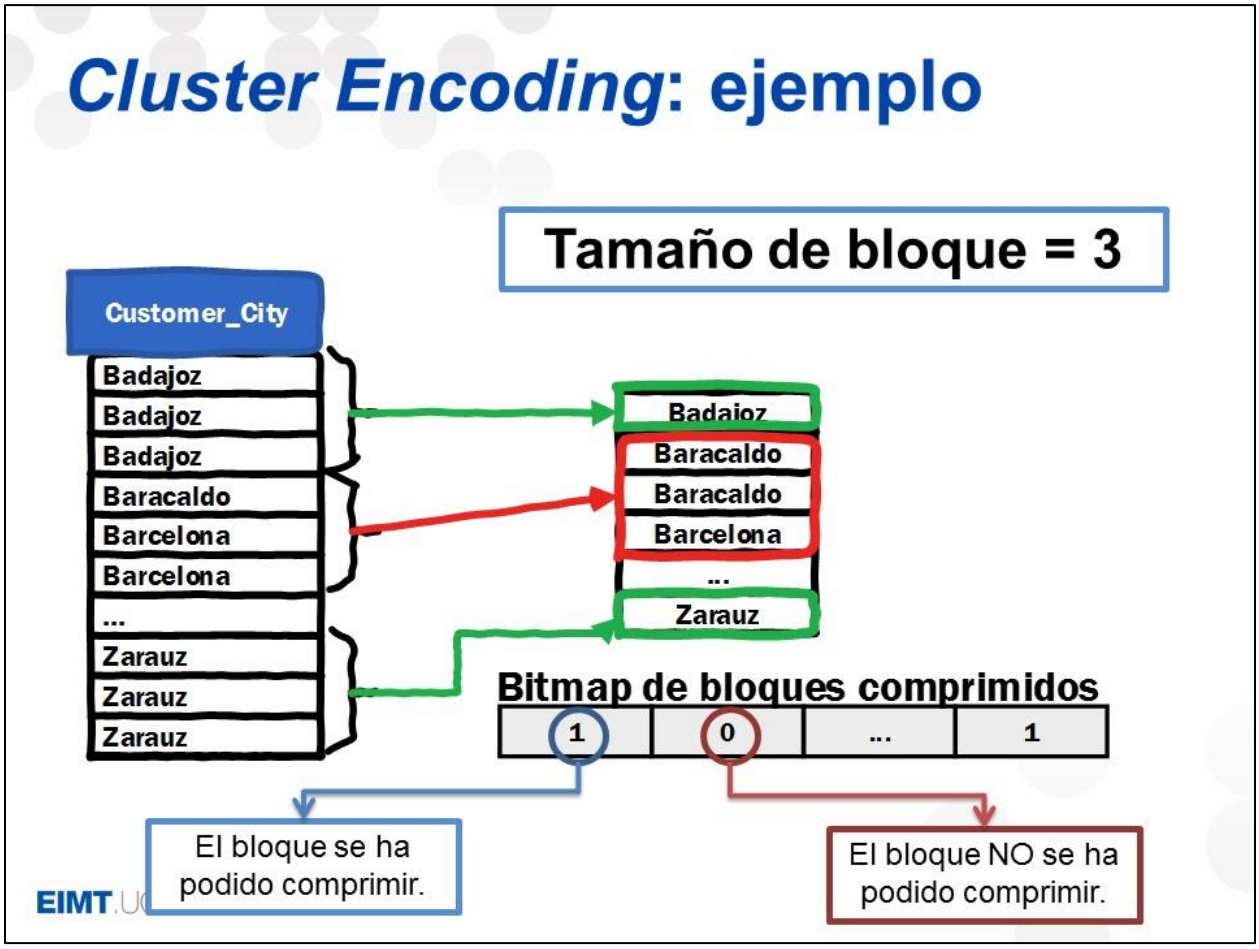
El algoritmo se puede aplicar tanto de forma aislada, como conjuntamente con otros como sería el caso, por ejemplo, del algoritmo *Dictionary Encoding*.



Seguidamente vamos a ver cómo funciona el algoritmo de *Cluster Encoding* a través de un ejemplo.

Continuamos con la tabla *Customer* utilizada en el algoritmo anterior, que recordemos tiene 46.524.943 filas. La columna *Customer_City* contendrá los nombres de las ciudades donde viven los clientes. En España existen 8.125 ciudades, en consecuencia la columna podrá tener 8.125 valores posibles. Supongamos también, para este ejemplo, que la tabla está ordenada por las columnas *Customer_Country* y *Customer_City* –es decir, el par (*Customer_Country*, *Customer_City*) es la clave de ordenación de la tabla *Customer*–.

En la transparencia podemos ver una representación gráfica y simplificada de la columna de *Customer_City*. Una vez presentado el contexto, vamos a ver cómo el algoritmo *Cluster Encoding* comprime esta columna.



Para facilitar la explicación de este ejemplo, asumiremos que se ha escogido un tamaño de bloque de 3 posiciones. Esto, como hemos comentado, es poco realista, ya que los valores de bloque acostumbran a ser del orden 1024 elementos o superior.

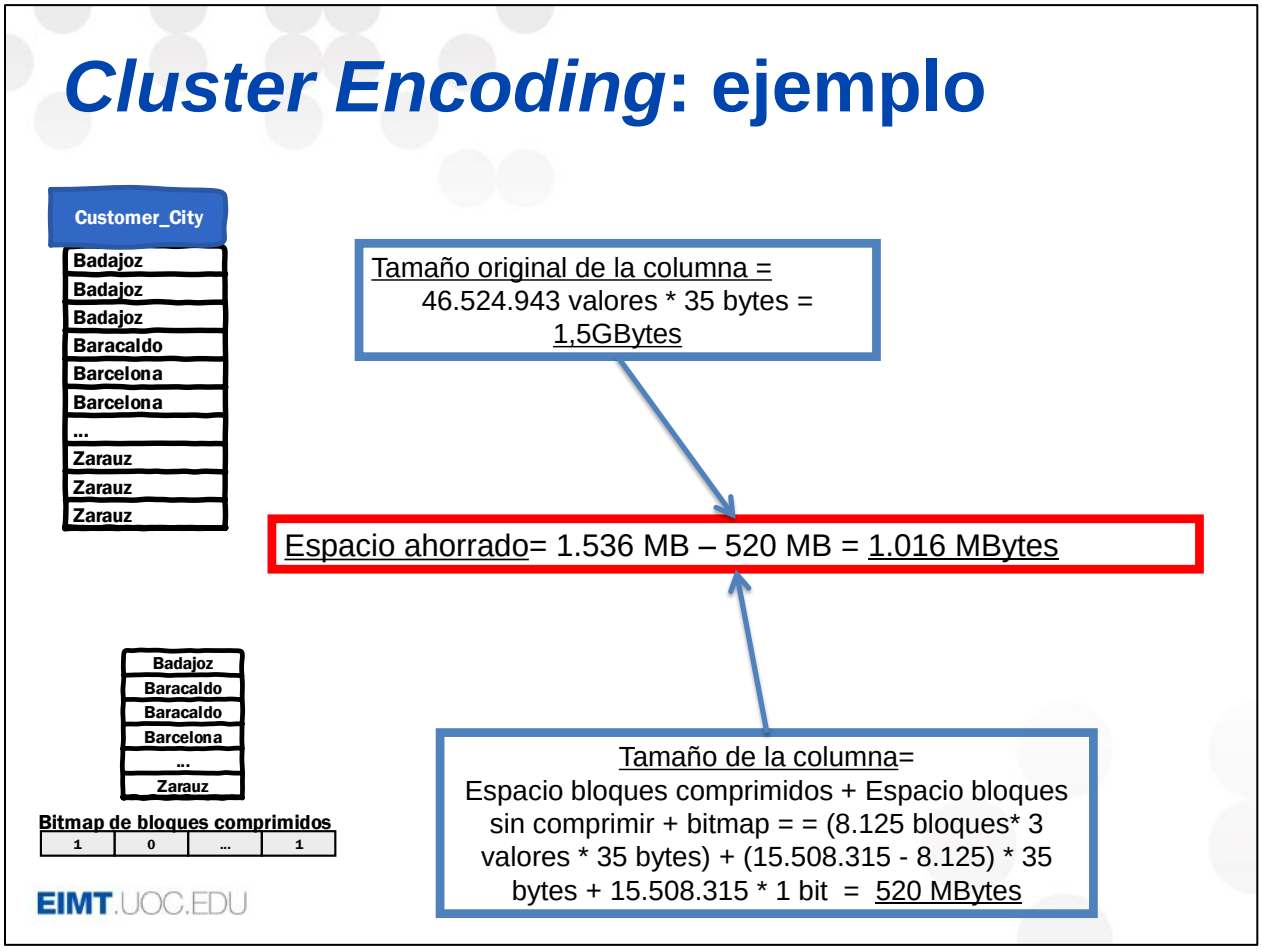
El algoritmo de compresión creará una nueva columna donde se almacenarán los valores comprimidos y un vector de bits que indicará qué bloques se han podido comprimir. En este caso, el espacio de las celdas de la nueva columna será igual al de la columna original ya que no deben almacenarse metadatos extra (es decir, información de control) para cada valor.

El algoritmo clasificaría la columna original en *N* bloques de tamaño 3. Después, para cada bloque de la tabla (posición *i*), se realizaría la siguiente pregunta: ¿todos los valores de las celdas del bloque *i* son iguales?

- En caso afirmativo:
  - Se asigna a la columna comprimida un solo valor: el valor repetido.
  - Se asigna un *1* en la posición *i-ésima* del vector de bits para indicar que el bloque *i* se ha comprimido.

- En caso negativo:
    - Se asigna a la columna comprimida el bloque entero, las tres celdas de la columna original.
    - Se asigna un *0* en la posición *i-ésima* del vector de bits para indicar que el bloque *i* no se ha comprimido.

En el ejemplo de la transparencia, se clasificaría la columna en bloques de 3 celdas. El primer bloque estaría formado por 3 celdas con el mismo valor: *Badajoz*. Al tener el mismo valor, se escribiría en la columna comprimida un solo valor de *Badajoz* y se asignaría un *1* a la posición *1* del vector de bits para indicar que el primer bloque ha sido comprimido. El segundo bloque de la columna no puede ser comprimido, porque contiene valores distintos (*Barcelona* y *Baracaldo*). En este caso se copiaría el bloque entero en la columna comprimida, y se indicaría en el vector de bits que no se ha podido comprimir (un 0 en la posición 2 del vector). Continuaríamos aplicando el algoritmo hasta el final. Fijaos que el último bloque se podría comprimir porque todas sus celdas comparten el mismo valor (*Zarauz*), sustituyendo el bloque entero por el valor *Zarauz* en la columna comprimida e indicándolo insertando un *1* en la última posición del vector de bits.



Como hemos comentado, se ha escogido un tamaño de bloque de 3. Teniendo en cuenta que la tabla *Customer* contiene 46.524.943 filas, tendremos 15 millones de bloques:

Número de bloques(Customer_City) = 46.524.943 valores / 3 celdas = 15.508.315 bloques

Las celdas de la nueva columna comprimida ocupan el mismo espacio que en la columna original. Por lo tanto, el ahorro deberá venir por la reducción en el número de celdas. A continuación vamos a ver el resultado de la compresión para nuestro ejemplo. Veremos qué ocupaba la columna inicial, qué ocupa la columna comprimida y qué ganancia (en términos de ahorro de espacio de almacenamiento) hemos obtenido.

El tamaño de la columna original es de 1,5 GBytes. Se puede calcular multiplicando el número de celdas de la columna (46.524.943) por el tamaño de cada celda (35 bytes):

Tamaño original(Customer_City) = 46.524.943 valores x 35 bytes = 1.628.373.005 bytes ≈ 1.553 MBytes

Para calcular el tamaño de la nueva columna se deberá estimar el número de bloques comprimidos y el número de bloques no comprimidos. Teniendo en cuenta que la tabla *Customer* almacena sólo información de clientes de España, y que está ordenada por país y ciudad (columnas *Customer_Country* y *Customer_City*), sabemos que todos los valores repetidos estarán en celdas contiguas, y que en total hay *8.125* valores distintos (es decir, hay 8.125 ciudades diferentes). En consecuencia, podemos concluir que, en el peor de los casos, habrá 8.125 bloques sin comprimir. Una vez hecha esta estimación, podemos calcular cuál será el tamaño de la columna comprimida (de unos 520 MBytes) mediante la fórmula que se *muestra a continuación.*

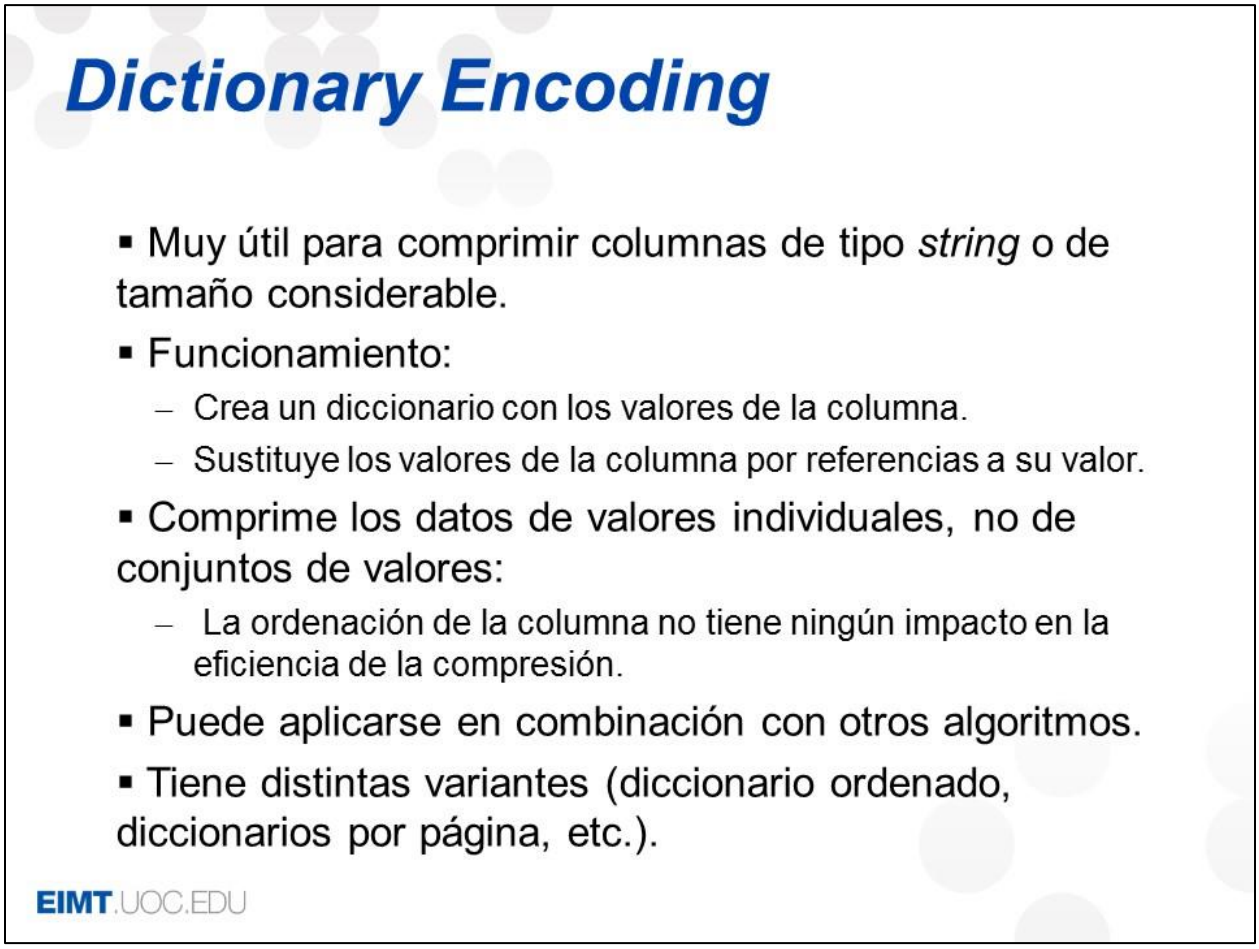
Tamaño comprimido(Customer_City) = (Número de bloques NO comprimidos x Número de celdas por bloque x Tamaño de celda) + (Número de bloques comprimidos x Tamaño de celda) + Tamaño vector de bits = (8.125 bloques 3 valores * 35 bytes) + (15.508.315 bloques comprimidos - 8.125 bloques sin comprimir) * 35 bytes + 15.508.315 posiciones * 1 bit*
≈ 520 MBytes

Finalmente, el ahorro en el espacio de almacenamiento vendrá dado por la diferencia entre el tamaño inicial y el comprimido:

Ahorro(Customer_City) = 1.536 MB – 520 MB = 1.016 MBytes

Podemos observar que el ahorro obtenido en nuestro ejemplo es muy bueno, porque los datos estaban ordenados y todos los posibles valores tenían repeticiones. No obstante, debido a que los valores repetidos no estaban distribuidos a lo largo de la columna, sino que se encontraban juntos, aplicar un algoritmo como *Run-Length Encoding* hubiera sido más eficiente. Se deja como tarea al lector realizar los cálculos para ver qué mejora se obtendría con el algoritmo *Run-Length Encoding*, y que lo compare con el ahorro actual.

Una vez vistos los algoritmos que reducen el número de valores (o celdas) de una columna, vamos a centrarnos en los que comprimen los datos reduciendo el tamaño de cada valor (o celda) de la columna, es decir, en los algoritmos que reducen el tamaño necesario para almacenar cada valor de una columna.



El algoritmo *Dictionary Encoding* comprime los datos de una columna reduciendo el tamaño necesario para representar sus valores. Su compresión no se basa en la reducción del número de celdas de una columna, sino en la reducción del tamaño de cada celda. En consecuencia, si hay 100 valores iguales consecutivos en una columna, el algoritmo generará 100 entradas, una para cada valor individual. Por ello, tener la columna ordenada no implicará ninguna mejora en este algoritmo de compresión.

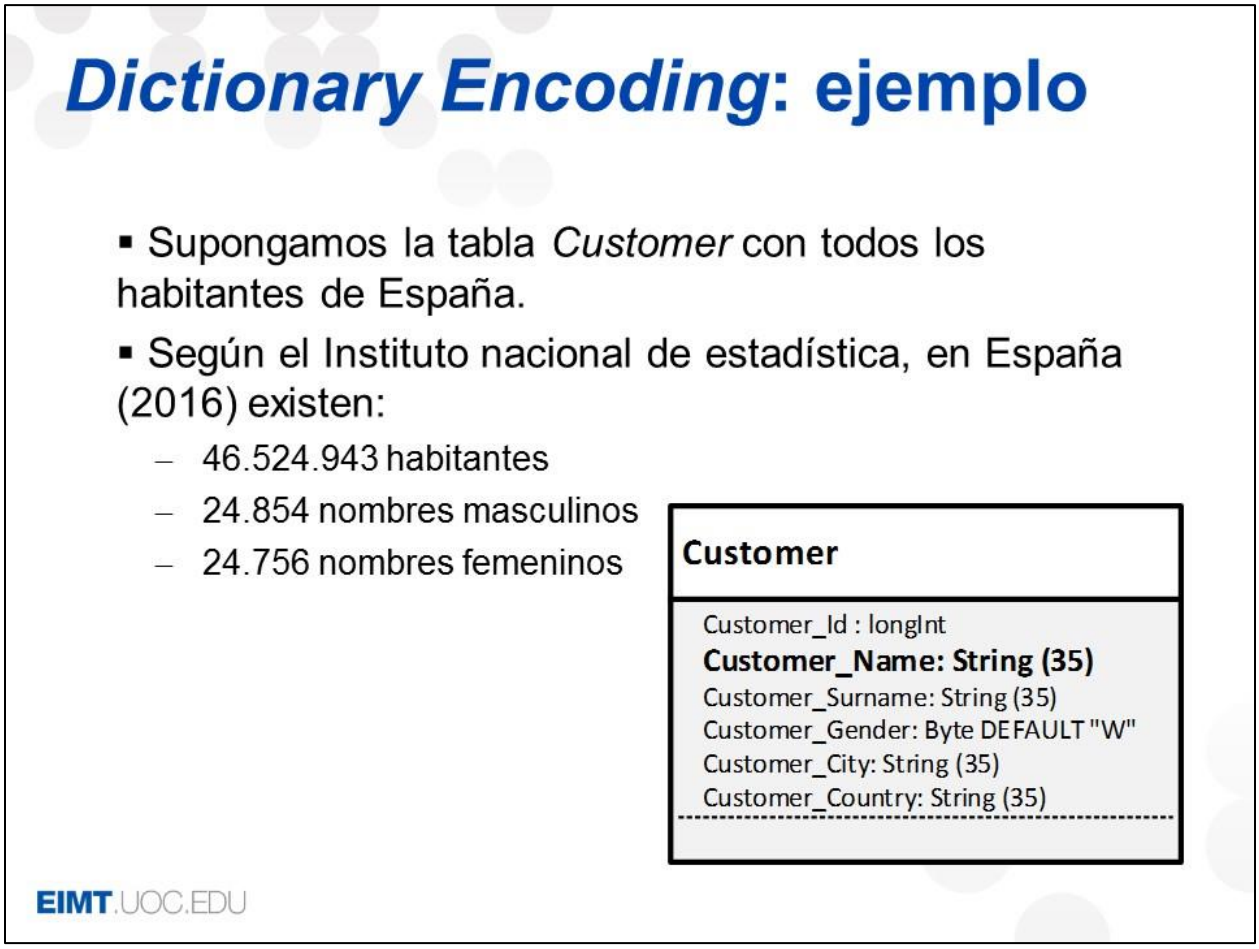
Es un algoritmo que resulta muy útil para comprimir *strings* y columnas que tengan asociado un tipo de datos de un tamaño significativo. En el caso de columnas de gran tamaño como sería el caso, por ejemplo, de columnas con un tipo de datos asociado BLOB, este algoritmo puede no ser necesario, dado que, como ya sabemos, el SGBD suele guardar dichos objetos grandes en estructuras externas separadas, de forma parecida a cómo se haría mediante este algoritmo.

Para comprimir los datos, el algoritmo crea un diccionario que contiene los valores distintos que aparecen en la columna. Una vez hecho esto, se comprime la columna sustituyendo el valor de cada una de sus celdas por el índice de la posición del diccionario que contiene dicho valor. La ganancia está en que se ha convertido el valor original, que potencialmente ocupa varios bytes, en un número natural que puede representarse utilizando muy pocos bits. Normalmente el diccionario se almacena en una página distinta del disco. El ahorro aparece en los valores repetidos, ya que antes repetíamos un valor que tenía

un coste de bytes elevado múltiples veces, y ahora sólo repetimos un índice que tiene un coste de bits mucho menor. Por otro lado, y como aspecto potencialmente negativo, se está añadiendo una nueva redirección (o indirección) para consultar los datos, cosa que puede hacer inviable el algoritmo si la ganancia no es muy buena, y dependiendo también del tipo de consultas que se deban realizar.

Este algoritmo es uno de los más utilizados hoy en día. Se puede aplicar conjuntamente con otros algoritmos que reducen el número de celdas, como son los dos algoritmos de compresión que hemos visto anteriormente, los algoritmos *Run-Length Encoding* y *Cluster Encoding*.

El algoritmo tiene distintas variantes. Una de dichas variantes consiste, por ejemplo, en almacenar el diccionario de forma ordenada (esto puede ser muy útil para realizar lecturas secuenciales por valor), o bien definir un diccionario para cada página de la BD.

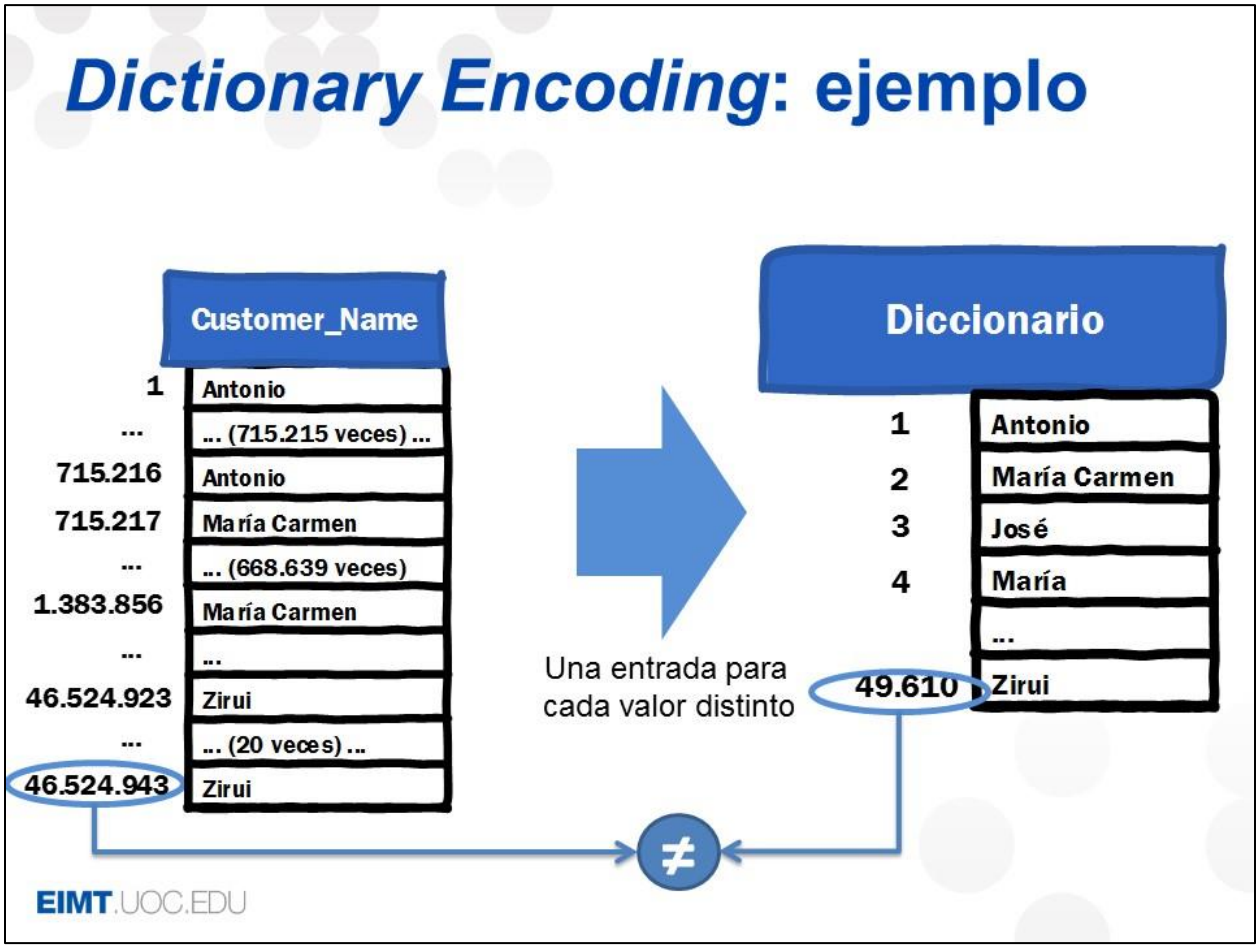


Vamos a ver cómo funciona este algoritmo. Para ello utilizaremos el mismo ejemplo que hemos usado para el algoritmo *Run-Length Encoding*. Recordémoslo:

En este ejemplo vamos a comprimir la columna *Customer_Name* de la tabla *Customer*.

Supondremos que la empresa ha realizado ventas a toda la población española. Por lo tanto, la tabla *Customer* contendrá 46.524.943 filas, una para cada persona residente en España. La columna *Customer_Name* contendrá los nombres de dichas personas. Según el INE[3] en España existen un total de 24.854 nombres masculinos y 24.756 nombres femeninos distintos. Por ello habrá muchas repeticiones de nombres, 715.215 repeticiones de *Antonio*, 668.639 repeticiones del nombre *María Carmen*, 20 repeticiones del nombre *Zirui*, etc. Supondremos también que la columna está ordenada en función del número de ocurrencias de los nombres de los clientes.

[3] Ver http://www.ine.es/daco/daco42/nombyapel/nombyapel.htm



Una vez presentado el contexto, vamos a ver cómo este algoritmo comprime la columna *Customer_Name*.

El algoritmo realiza dos acciones para comprimir los datos, en concreto:

1. La creación de un diccionario con los valores de la columna.
2. La compresión de la columna sustituyendo los valores originales por índices que apuntan a la ubicación de dichos valores en el diccionario.

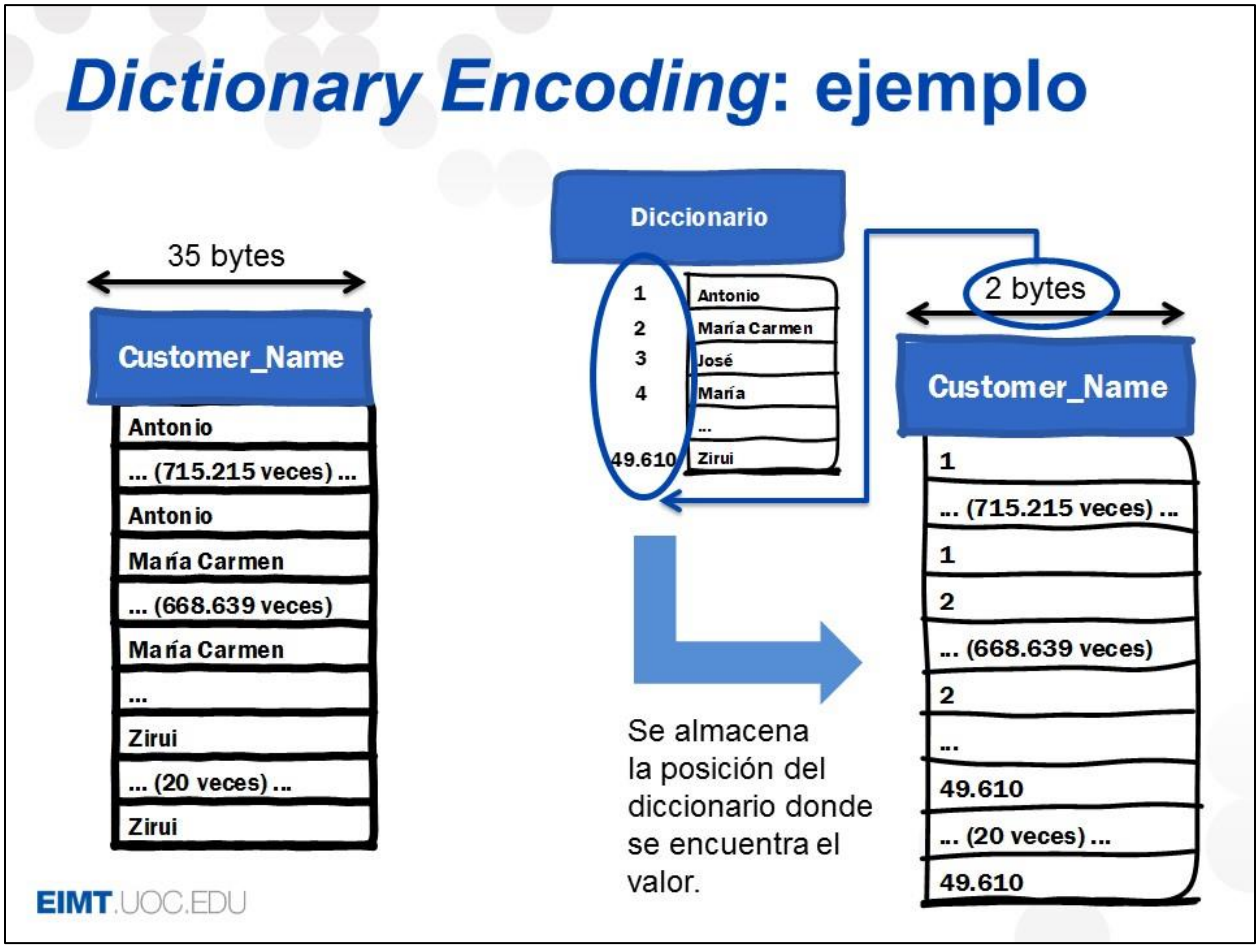
Ambas acciones se pueden realizar a la vez, pero en esta presentación las efectuaremos de forma secuencial por una mayor claridad.

Lo primero que se hará es crear el diccionario. Para ello hay que saber el tamaño que debe tener cada entrada del diccionario, así como el número de entradas que deberá tener. El tamaño de cada entrada del diccionario será de 35 bytes ya que debe almacenar nombres de clientes. Por otro lado, el diccionario debe almacenar todos los valores que aparecen en la columna. Según los datos ofrecidos por el INE, tendremos 49.610 valores posibles, por lo tanto el diccionario tendrá 49.610 entradas. Saber el número de entradas del diccionario es importante, porque condicionará el tamaño de las celdas de la columna

comprimida. En particular, en el caso que nos ocupa, necesitamos 16 bits (2 bytes) para direccionar hasta 65.000 valores. En consecuencia, la nueva columna comprimida contendrá celdas de 2 bytes.

Una vez creado el diccionario, se poblará con el contenido de la columna. Para ello se iterará sobre todas las celdas de la columna. Para cada celda, se comprobará si su valor ya está almacenado en el diccionario. Si no fuese así, se añadirá su valor en el diccionario. La manera en que se almacenan los datos en el diccionario puede ser distinta. Se podrían almacenar los valores de forma ordenada, por ejemplo. Sobre nuestro caso de ejemplo concreto, para simplificar, se añadirán en orden de ocurrencia.

Notad que, gracias al diccionario, hemos podido reducir las repeticiones de valores, ya que hemos pasado de la representación de 46 millones de valores a la representación de cerca de 50 mil. La ganancia de este algoritmo vendrá en el caso de que los valores ocupen mucho más que los índices hacia el diccionario, como sucede en el caso de nuestro ejemplo.

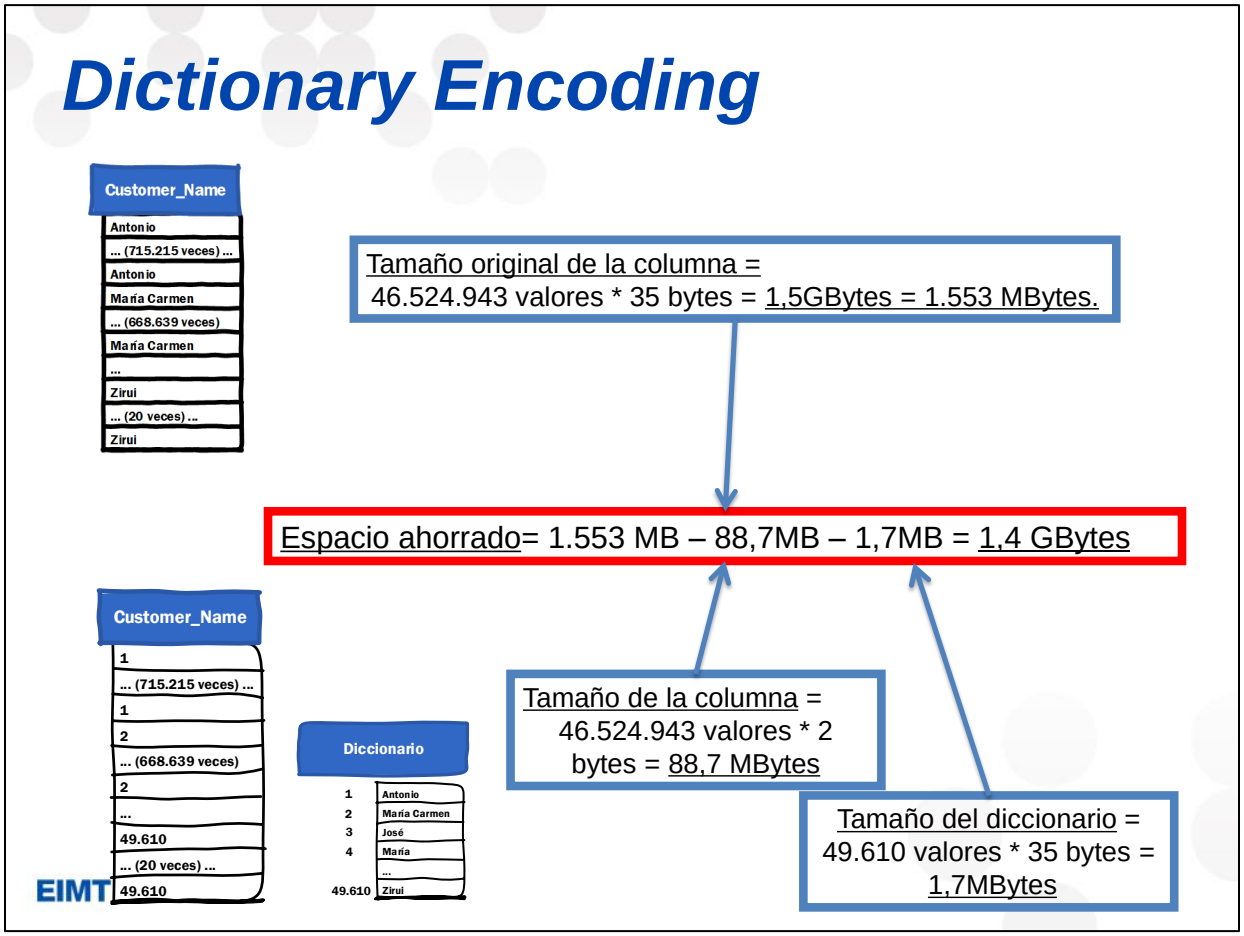


Una vez creado el diccionario se podrá proceder a la compresión de los valores de la columna. Para hacerlo, se iterará sobre la columna original y, para cada celda (posición *i-ésima*) de la misma, se hará lo siguiente:

- Se buscará su valor en el diccionario.
- Se obtendrá el índice (el número de entrada) del diccionario donde se encuentra el valor.
- Se añadirá dicho índice a la columna comprimida.

En el caso que nos ocupa, para la primera celda, se consultaría en el diccionario su valor (*Antonio*). Como *Antonio* está almacenado en la primera posición del diccionario, su valor se sustituiría por el número *1* en la columna comprimida. Esto se haría para las siguientes 715.215 celdas, donde el valor de *Antonio* se repite. En la celda 715.216, se consultaría el valor de *María Carmen* en el diccionario, y se obtendría su posición (la número 2). Por lo tanto, se sustituiría el valor *María Carmen* por el número *2* en la columna comprimida. Esto se haría para todas sus repeticiones, y así sucesivamente para el resto de valores de la columna y sus ocurrencias.

Al final, la columna comprimida tendrá el mismo número de celdas que la columna original y el mismo número de repeticiones. La diferencia estriba en que se ha cambiado el tipo de la misma por un tipo numérico que ocupa 2 bytes, mucho menos que los 35 bytes que ocupaban los nombres.



A continuación vamos a ver el resultado de la compresión en el ejemplo realizado. Veremos qué ocupaba la columna inicial, qué ocupa la columna comprimida y qué ganancia (en términos de ahorro en espacio de almacenamiento) hemos obtenido.

El tamaño de la columna original es de 1,5 GBytes. Se puede calcular multiplicando el número de celdas de la columna (46.524.943) por el tamaño de cada celda (35 bytes):

$$Tamaño\ Original(Customer\_Name) = 46.524.943\ valores \times 35\ bytes = 1.628.373.005\ bytes \approx 1.553\ MBytes$$

El tamaño de la nueva columna vendrá dado por el tamaño de la columna más el tamaño del diccionario. Dichos tamaños pueden calcularse tal y como se indica a continuación.

Tamaño diccionario(Customer_Name) = 49.610 valores x 35 bytes = 1.736.350 bytes ≈ 1,7 MBytes

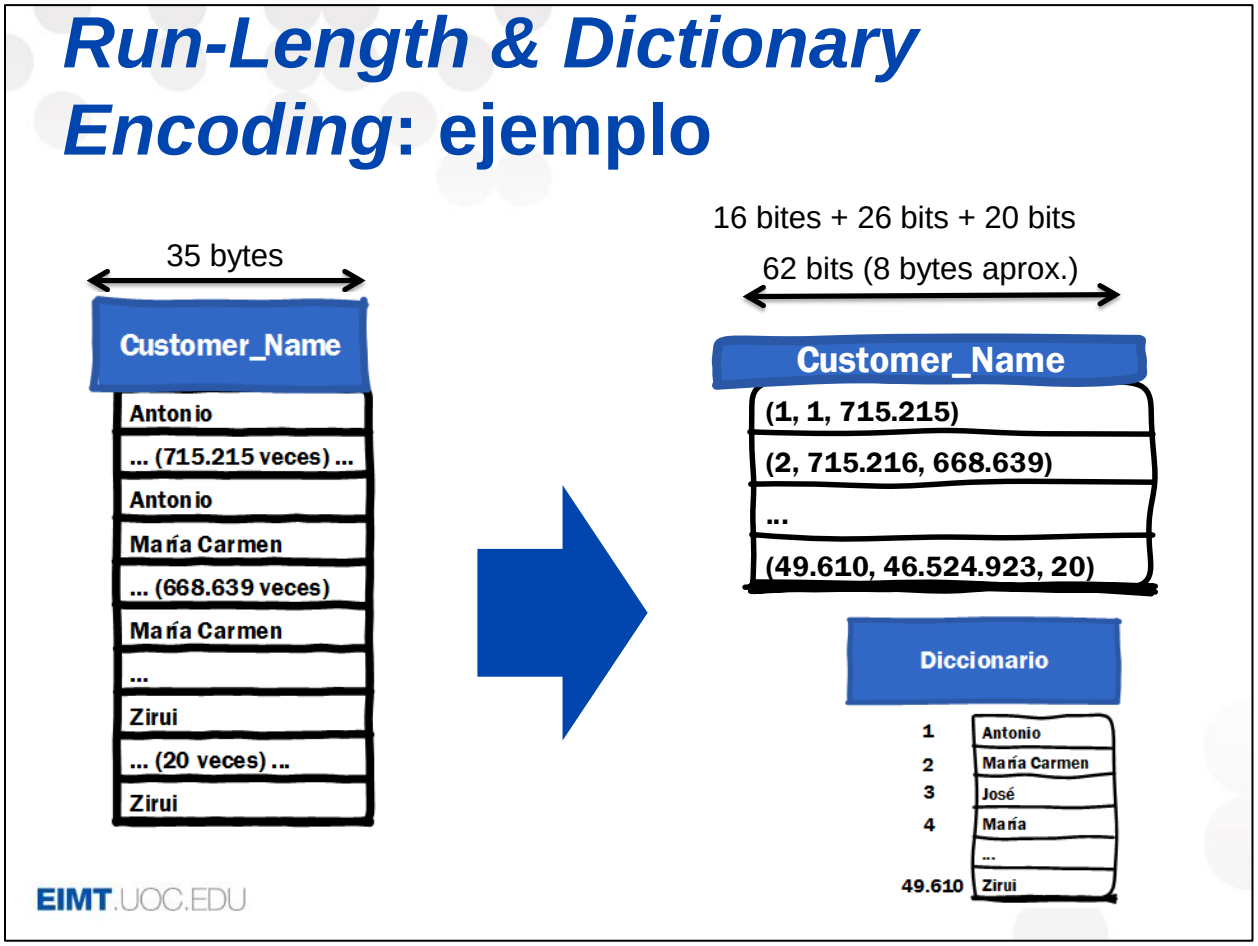
Tamaño comprimido(Customer_Name) = 46.524.943 valores x 2 bytes = 93.049.886 bytes ≈ 88,7 MBytes

Por lo tanto, el valor final de la columna comprimida será de 90,4 MBytes. El ahorro final vendrá dado por la diferencia entre el tamaño inicial y el comprimido:

Ahorro(Customer_Name) = 1.553 MBytes – 1,7 MBytes – 88,7 MBytes = 1.462,6 MBytes

Podemos observar que el ahorro obtenido en este caso es muy bueno, porque había muchas repeticiones y los datos originales eran mucho mayores que los índices necesarios para acceder al diccionario. No obstante, la columna resultante aún tiene muchas repeticiones. En otras palabras, la columna puede ser susceptible de ser comprimida con algoritmos que reduzcan el número de celdas de una columna, para alcanzar un mejor ratio de compresión.

En la siguiente transparencia mostraremos cómo se puede complementar la compresión realizada con el algoritmo *Dictionary Encoding*, con el algoritmo *Run-Length Encoding* para reducir aún más el tamaño de la columna *Customer_Name* de nuestra tabla de ejemplo *Customer*.

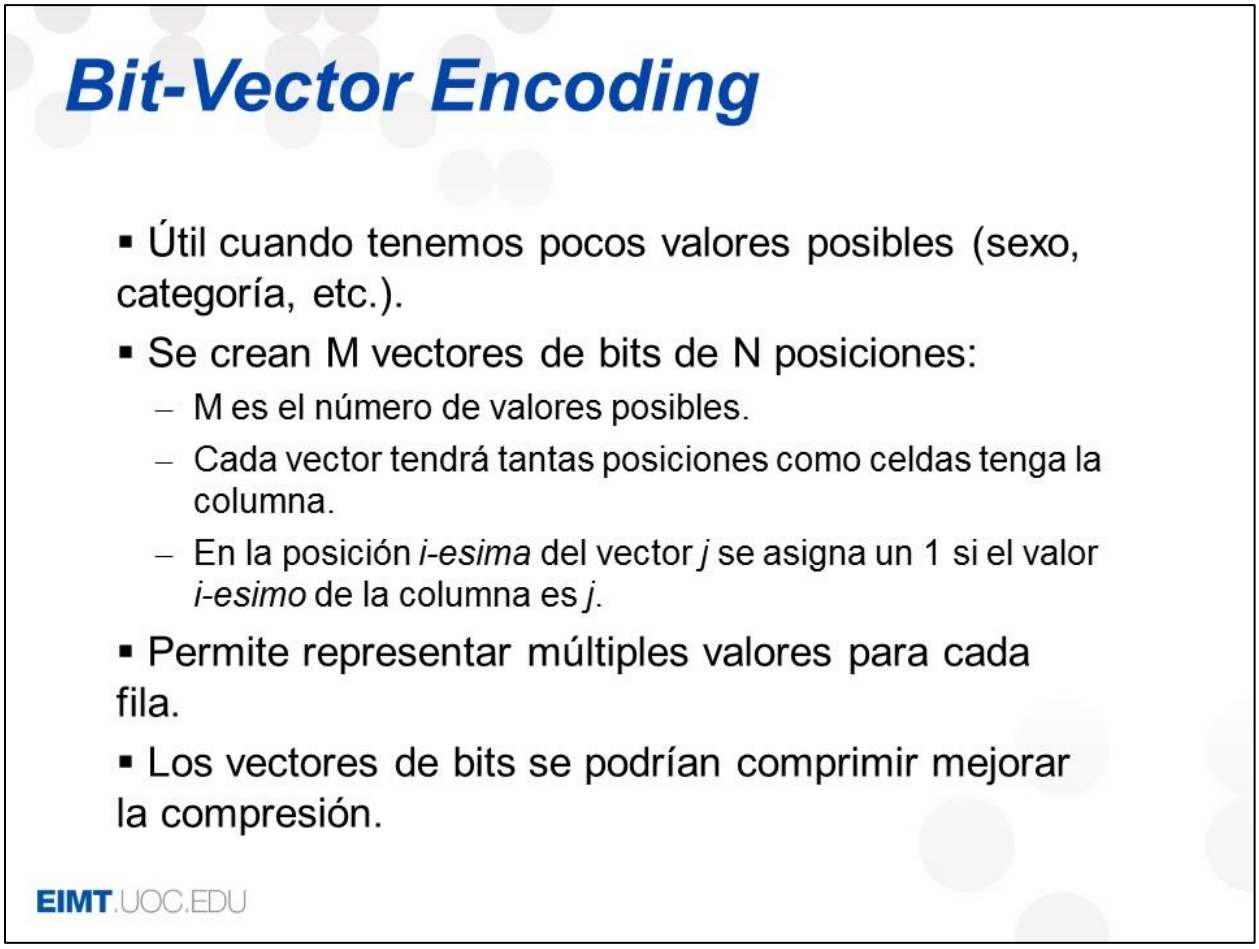


Tal y como acabamos de comentar, la columna *Customer_Name* se podría comprimir combinando los algoritmos *Run-Length Encoding* y *Dictionary Encoding*. En el caso de que lo hiciésemos, nos encontraríamos con un resultado como el que se muestra en la diapositiva.

Para ello se crearía un diccionario que almacenaría los nombres de las personas (que son de 35 bytes), que tendría el mismo tamaño y características que hemos visto en las transparencias anteriores. Posteriormente, se procedería a compactar las celdas con valores repetidos, utilizando el algoritmo *Run-Length Encoding*. El tamaño de las celdas de la columna comprimida en este caso sería de 62 bits (16 bits para representar la entrada del diccionario que contiene el valor, 20 bits para mostrar la posición de la celda donde aparece el valor y 26 bits para mostrar el número de repeticiones consecutivas).

Una vez calculado el tamaño de la nueva columna, la compresión se efectuaría de la misma forma que vimos en el apartado de esta presentación dedicado al algoritmo *Run-Length Encoding*, pero teniendo en cuenta que ahora el valor no es de tipo *string*, sino de tipo numérico. En este caso, la primera entrada de la columna comprimida no tendría el valor *(Antonio, 1, 715.215)* como antes, sino *(1, 1, 715.215)*. Vemos cómo se ha cambiado el valor de *Antonio* por la posición de *Antonio* en el diccionario; dicha posición es la número *1*.

Dejamos como ejercicio para el lector comprobar la ganancia al realizar la compresión mostrada en esta transparencia. Animamos a que, una vez calculada, la compare con las ganancias de aplicar *Dictionary Encoding* y *Run-Length Encoding* de forma aislada y reflexione sobre los resultados obtenidos.



El algoritmo *Bit-Vector Encoding* comprime los datos de una columna reduciendo el tamaño necesario para representar sus valores. Al igual que en el caso del *Dictionary Encoding*, su compresión se basa en la reducción del tamaño necesario para almacenar los valores de cada celda, es decir, no reduce el tamaño en función de las repeticiones de valores, ni tendrá una mejor ganancia cuando los datos estén ordenados.

El algoritmo es muy útil cuando una columna tiene pocos valores posibles, como sería el caso del género de una persona, la categoría de un producto, el tipo de un cliente etc.

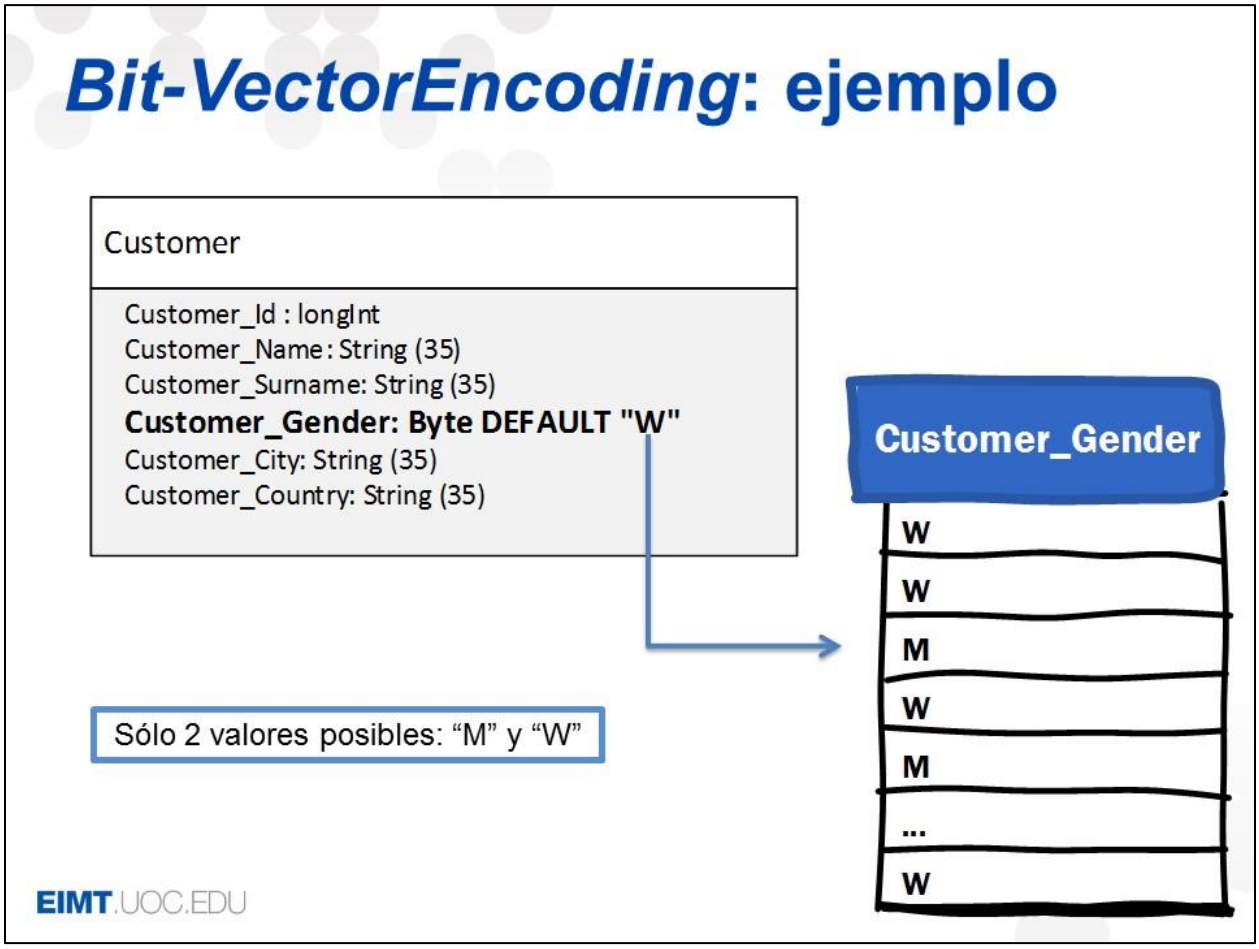
Para comprimir los datos, se crea un vector de bits para cada posible valor que pueda tomar la columna. Este vector de bits tiene tantas posiciones como celdas tiene la columna original. Cada posición *i* del vector contiene un bit que indica si la celda en la posición *i* tiene asignado ese valor. Aparte de las mejoras que se puedan obtener en términos de compresión de datos, trabajar con vectores de bits es muy eficiente y conveniente para realizar algunas operaciones sobre la BD (como por ejemplo realizar operaciones booleanas sobre un conjunto de columnas o comprobar que filas tienen un determinado valor), y por lo tanto, puede representar una ganancia considerable en su rendimiento. Además, el uso de vectores de bits permite representar columnas que permitan valores múltiples, por ejemplo, una columna que representa el conjunto de *keywords* que describen un producto. Aunque (al menos desde un punto de vista teórico) una BD basada en el modelo relacional no permite definir columnas

multivaluadas, debemos tener en cuenta que existen otras BD basadas en modelos de datos diferentes que sí lo permiten (por ejemplo, las BD NoSQL).

Como aspecto negativo, comentar que se requiere crear tantos vectores de bits, como valores pueda tener una celda. Eso puede ser inviable en función del número de valores posibles.

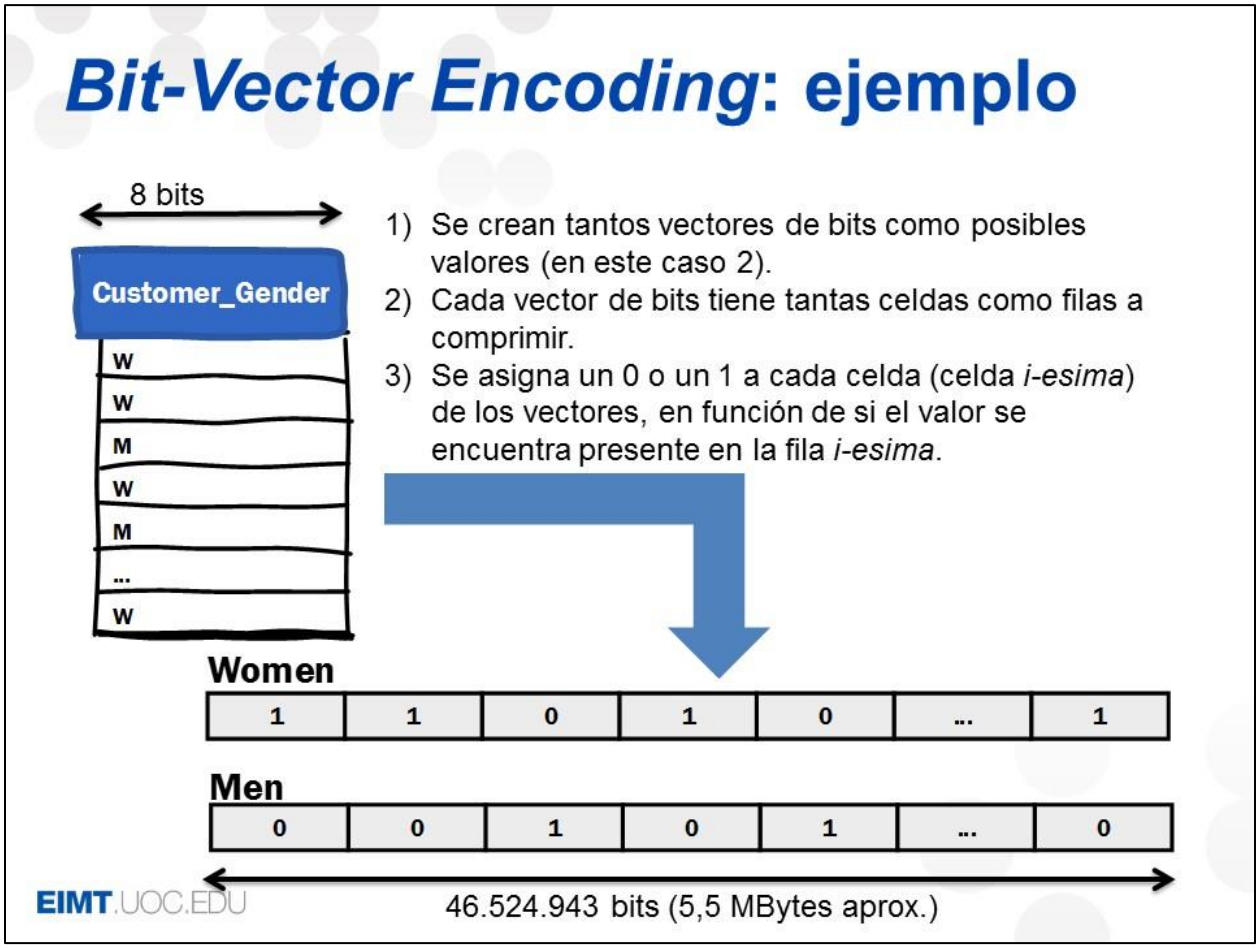
Es importante notar que en este tipo de compresión, la columna comprimida no existe como tal, sino que se compone por el conjunto de vectores de bits creados.

Por sus características, los vectores de bits pueden tener muchas repeticiones. En el caso que se considere necesario, se podrían comprimir usando alguno de los algoritmos vistos anteriormente.



Seguidamente vamos a ver cómo funciona este algoritmo mediante un ejemplo. Para ello utilizaremos la tabla *Customer* que hemos estado utilizando hasta ahora. En este caso, procederemos a comprimir la columna *Customer_Gender*, que sólo acepta dos valores: *M* para indicar que el cliente es un hombre (*Men*) y *W* para indicar que el cliente es una mujer (*Women*).

Una vez presentado el contexto, vamos a ver cómo se comprime esta columna mediante el algoritmo *Bit-Vector Encoding*.



Tal y como hemos comentado, el primer paso del algoritmo es crear un vector de bits para cada posible valor de la columna. Como los posibles valores de la columna son dos (*Men* y *Women*) se han creado dos vectores de bits:

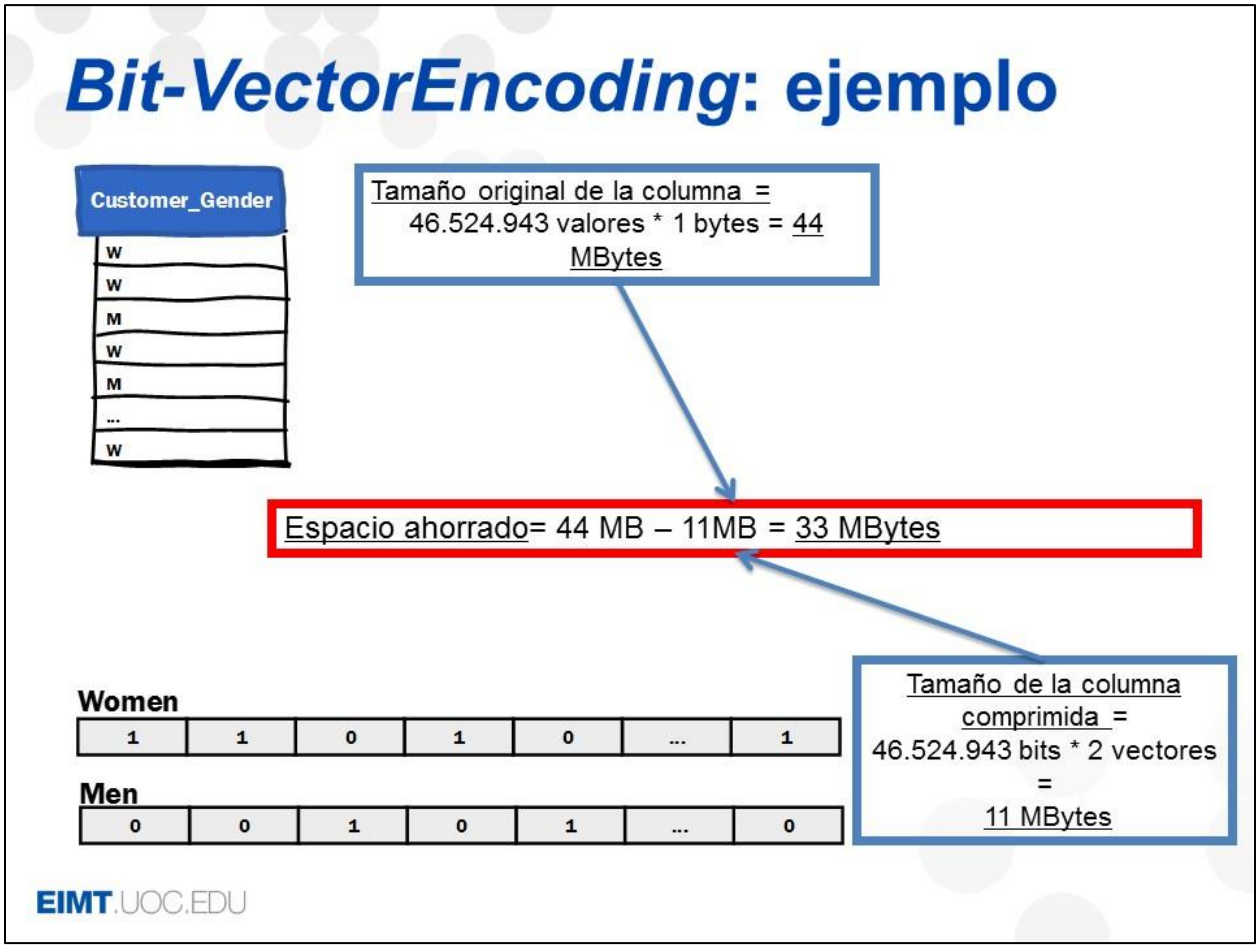
1. *Women*: indica qué clientes son de género femenino.
2. *Men*: indica qué clientes son de género masculino.

Los vectores tendrán tantas posiciones como celdas tenga la columna. En este caso 46.524.943 bits. En consecuencia, cada vector ocupará cerca de 5,5 MBytes.

Para asignar los valores a los vectores se iteraría sobre todas las celdas de la columna original. Para cada celda (siendo *i* su posición) se realizarían las siguientes acciones:

- Si $valor_i$ es *M*, entonces se asignaría un *1* en la posición *i* del vector *Men* y un *0* en la posición *i* del vector *Women*.
- Si $valor_i$ es *W*, entonces se asignaría un *1* en la posición *i* del vector *Women* y un *0* en la posición *i* del vector *Men*.

En el ejemplo de la transparencia, la primera y segunda celdas contienen el valor *W*. Por lo tanto, se asignará un *1* en la primera y en la segunda posición de *Women*, y un *0* en las dos primeras posiciones de *Men*. La tercera celda contiene el valor *M*, en consecuencia se asignará un *1* en la tercera posición de *Men* y un *0* en la tercera posición de *Women*. Y así sucesivamente. En la transparencia podemos ver el resultado de aplicar el procedimiento descrito.



A continuación vamos a ver el resultado de la compresión en el ejemplo realizado. Veremos qué ocupaba la columna inicial, qué ocupa la columna comprimida y qué ganancia (en términos de ahorro en espacio de almacenaje) hemos obtenido. Supondremos que la columna original almacena los valores *M* y *W* mediante un carácter, y por lo tanto, cada celda ocupa 1 byte. Bajo este supuesto, el tamaño de la columna original (*Customer_Gender*) es de unos 44 MBytes.

Tamaño original(Customer_Gender) = 46.524.943 valores x 1 bytes = 46.524.943 bytes ≈ 44 MBytes

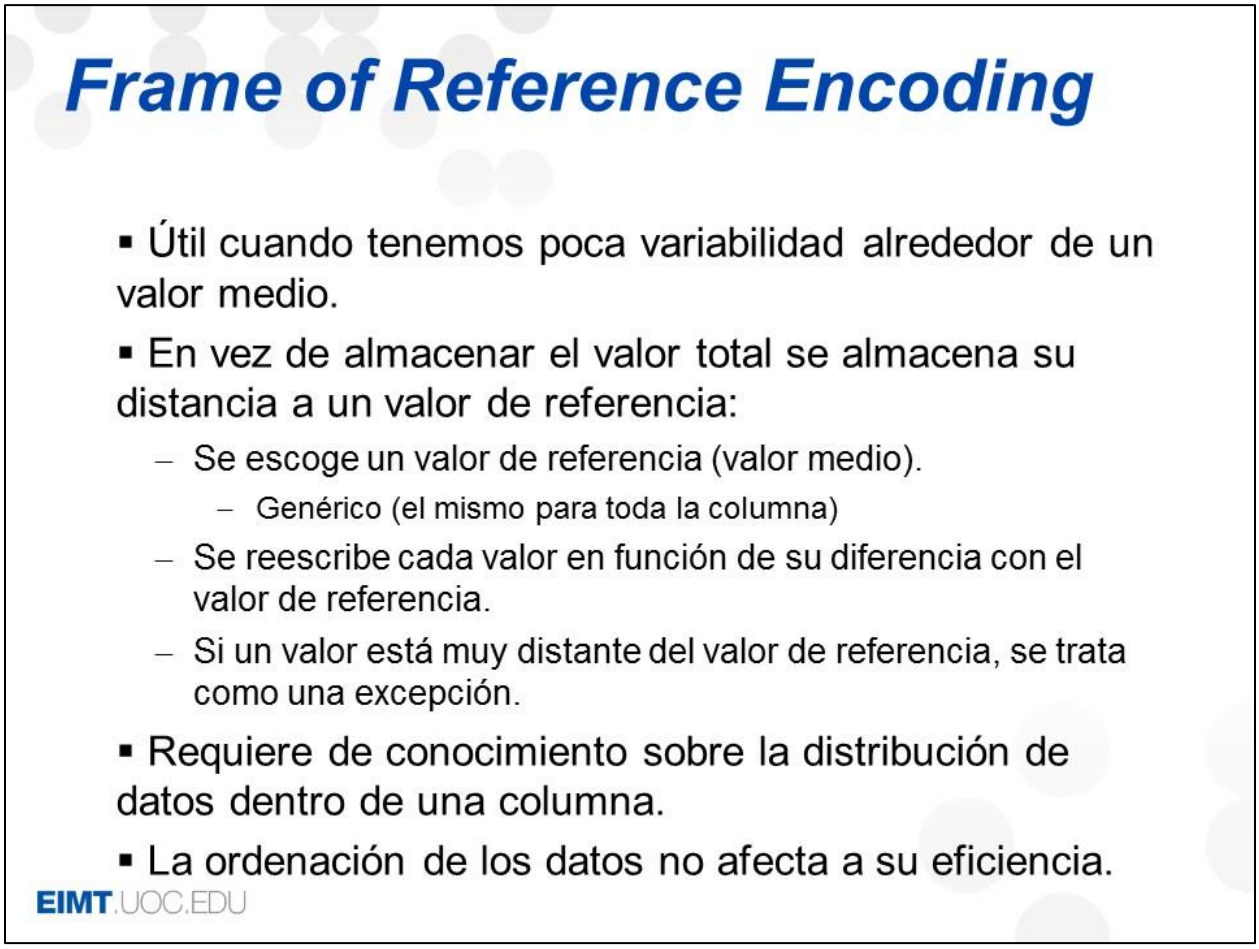
Por su parte, el tamaño de la nueva columna vendrá dado por el tamaño de todos los vectores de bits creados. Dichos tamaños pueden calcularse de la siguiente forma:

Tamaño comprimido(Customer_Gender) = 46.524.943 valores x 1 bit x 2 vectores = 93.049.886 bits ≈ 11 MBytes

Por lo tanto, el tamaño final de la columna comprimida será de 11 MBytes. El ahorro final vendrá dado por la diferencia entre el tamaño inicial y el comprimido:

Ahorro(Customer_Gender) = 44 MBytes – 11 MBytes = 33 MBytes

Podemos observar que el ahorro obtenido en este caso es significativo. No obstante, en el caso de que la columna se hubiera configurado inicialmente para ocupar sólo un bit no habría habido ganancia, más bien al contrario.



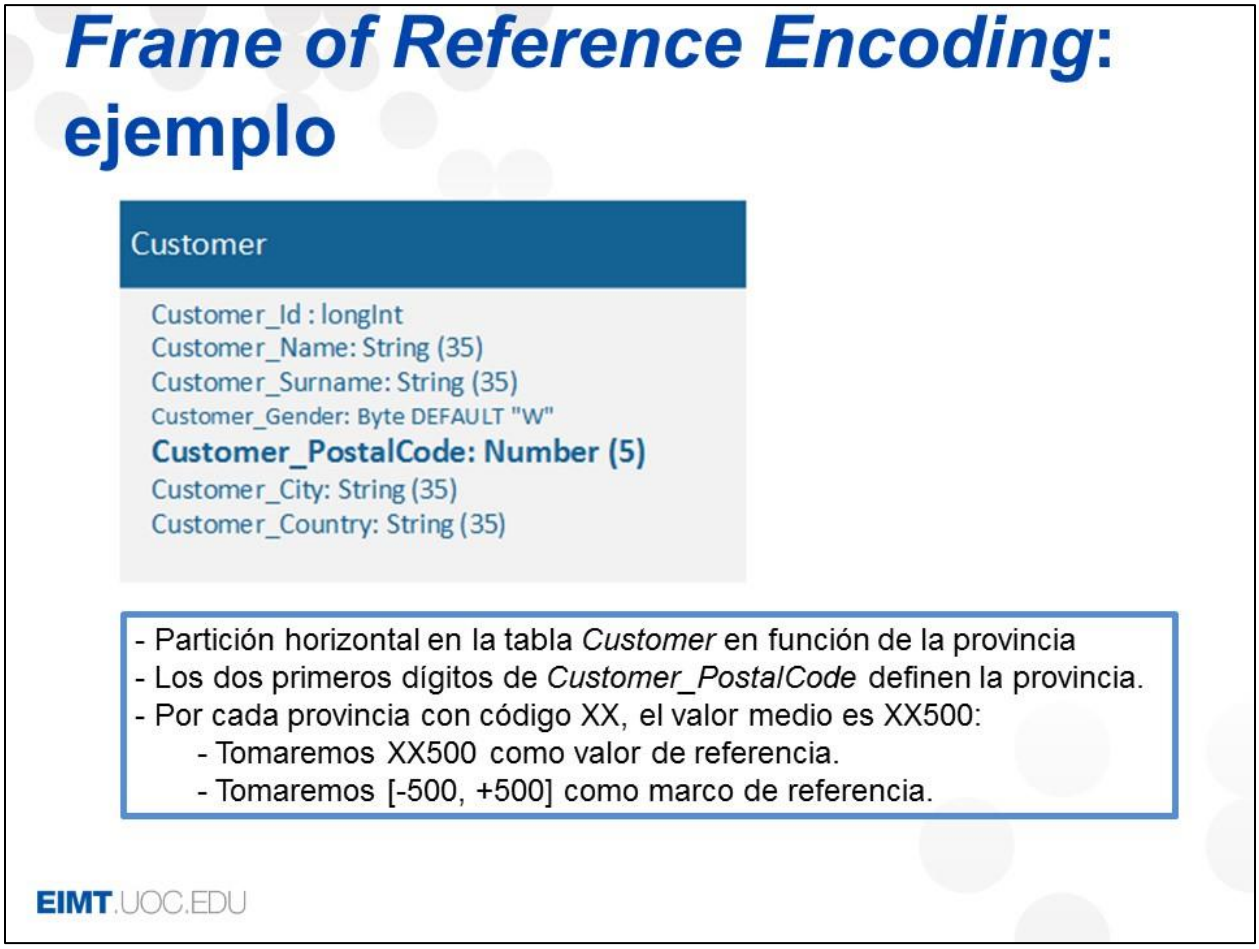
El algoritmo *Frame of Reference Encoding* comprime los datos de una columna reduciendo el tamaño necesario para representar sus valores. Por lo tanto, no reduce el tamaño en función de las repeticiones de sus valores, ni tampoco tendrá una mejor ganancia (o ratio de compresión) en el caso de que los datos estén ordenados.

Este algoritmo es aplicable cuando los valores de una columna están muy próximos a un valor concreto y muestran poca variabilidad. En dicho caso, los valores podrían reescribirse en función de su distancia a ese valor de referencia y así reducir, potencialmente, el espacio necesario para representarlos. Esta es la filosofía de trabajo que subyace detrás de este algoritmo.

Para ejecutar la compresión se escoge un valor de referencia, que es fijo (o sea, el mismo) para toda la columna. Este valor de referencia pretende ser el valor medio sobre el que se distribuyen los datos. Después se reescribe el valor de cada celda en función de la distancia que lo separa del valor de referencia. Para reducir el espacio necesario para almacenar los valores, se define una ventana (o marco) de referencia en función de los bits que quieran utilizarse para representar la distancia de los valores. Es posible que haya valores extremos, es decir, valores que están lejos del valor de referencia, y por lo tanto su diferencia queda fuera de la ventana establecida. En tal caso, se pueden representar como excepciones, indicando su valor original y un metadato (es decir, una información de control) que

indique que dicho valor es una excepción (o sea, no es la distancia con respecto al valor de referencia, sino el valor original).

Es necesario escoger un buen valor y un marco de referencia para que el algoritmo consiga resultados satisfactorios. Eso implica conocer cómo son y cómo se distribuyen los datos dentro de la columna. En el mejor de los casos el SGBD proporcionará cierta flexibilidad para que el usuario defina el valor y marco de referencia (al menos a nivel de byte).

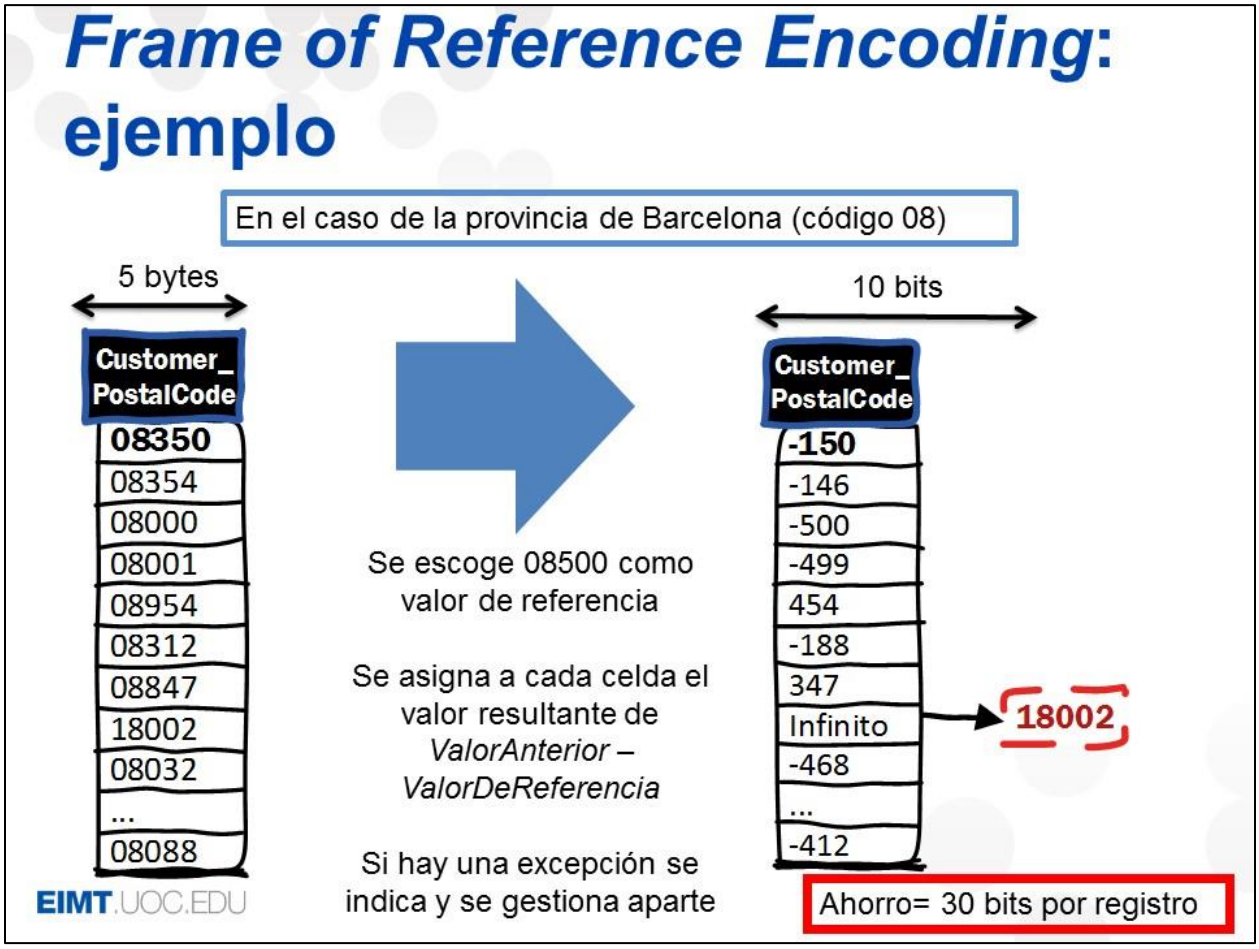


Vamos a ver cómo funciona este algoritmo mediante un ejemplo. De nuevo nos basaremos en la tabla *Customer*. En este caso, procederemos a comprimir la columna *Customer_PostalCode*, cuyo rango de valores estará entre el *00000* y el *99999*.

Para que el ejemplo sea más claro, supondremos que la tabla esta particionada (o fragmentada) horizontalmente por provincia. Cada partición horizontal contendrá entonces todos los clientes de una provincia determinada, siendo los dos dígitos de más peso de la columna los que identifican la provincia (*08XXX* corresponde a Barcelona, *17XXX* a Girona, *36XXX* a Pontevedra etc.).

Supongamos que queremos comprimir la columna en cada una de las particiones utilizando el algoritmo *Frame of Reference Encoding*. Sabemos que dada una provincia *XX* todos los valores oscilaran entre *XX000* y *XX999*. En consecuencia, podríamos definir el valor de referencia en *500* para cada provincia y el marco de referencia en una distancia de *[-500, 500]*.

Una vez presentado el contexto, vamos a ver cómo se comprime esta columna.



En concreto, vamos a ver en esta transparencia cómo se comprimiría la columna relativa a la partición horizontal que corresponde a la provincia de Barcelona.

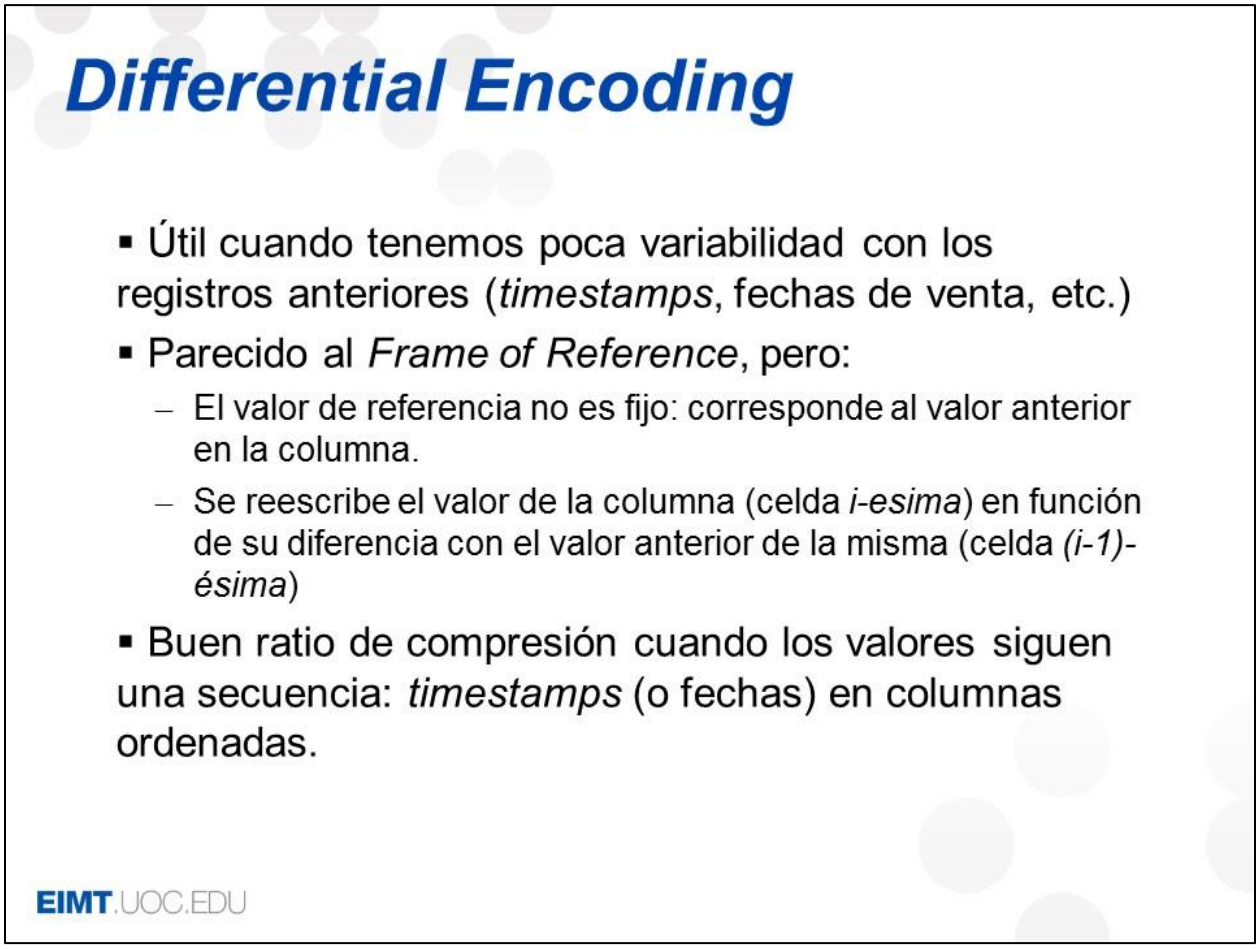
El código de Barcelona es *08*, en consecuencia todos los códigos postales serán del tipo *08XXX*, donde el valor *XXX* irá desde *000* a *999*. Tal y como hemos comentado, se establecerá el valor de referencia en *08500* y el marco de referencia en el intervalo *[-500,500]*. Eso quiere decir que la columna comprimida deberá ser capaz de almacenar valores que vayan de *-500* a *500*. Para almacenar dichos valores es necesario usar 10 bits. Por lo tanto, la columna comprimida contendrá 10 bits en el caso general.

Para cada una de las celdas de la columna original, se calcularía su homóloga en la columna comprimida. Para hacerlo se calculará la distancia del valor original con el valor de referencia (*valor original – valor de referencia*). El valor resultante se almacenará en la columna comprimida, tal y como podemos ver en la transparencia.

En el ejemplo de la transparencia también podemos ver como el valor *8350* se convierte en el valor *-150* (resultado de la operación 8350-8500), el valor *8354* en el valor comprimido *-146*, el valor *8000* en el valor *-500*, y así sucesivamente. En el caso de que un valor esté más distante del valor de referencia de lo esperado, se creará una excepción. Esta excepción se podría representar de distintas formas, una de ellas sería utilizar un valor por defecto para indicar que la distancia es infinita, y guardar en otra

estructura alternativa (o auxiliar) el valor original. En la transparencia podemos ver que por error un cliente de otra provincia o con un código erróneo (con código que comienza por *18*) se encuentra en la partición de Barcelona. Este valor está claramente fuera del marco de referencia, ya que su distancia al valor de referencia es de *9502* (valor muy superior al valor *512* representable con 10 bits). En este caso, se ha indicado que la distancia es infinita (está más allá del marco de referencia), y el valor original se almacena aparte.

Al no saber con certeza el número de excepciones que nos vamos a encontrar, no tiene mucho sentido analizar la ganancia total de la compresión de la columna. En este caso particular, creemos que es más útil indicar la ganancia por celda. Para cada valor original de la columna dentro del marco de referencia, tendremos un ahorro de 30 bits (en el caso de que el código postal se almacene mediante 5 caracteres) o de 7 bits (en el caso de que el código postal se almacene como un valor numérico).



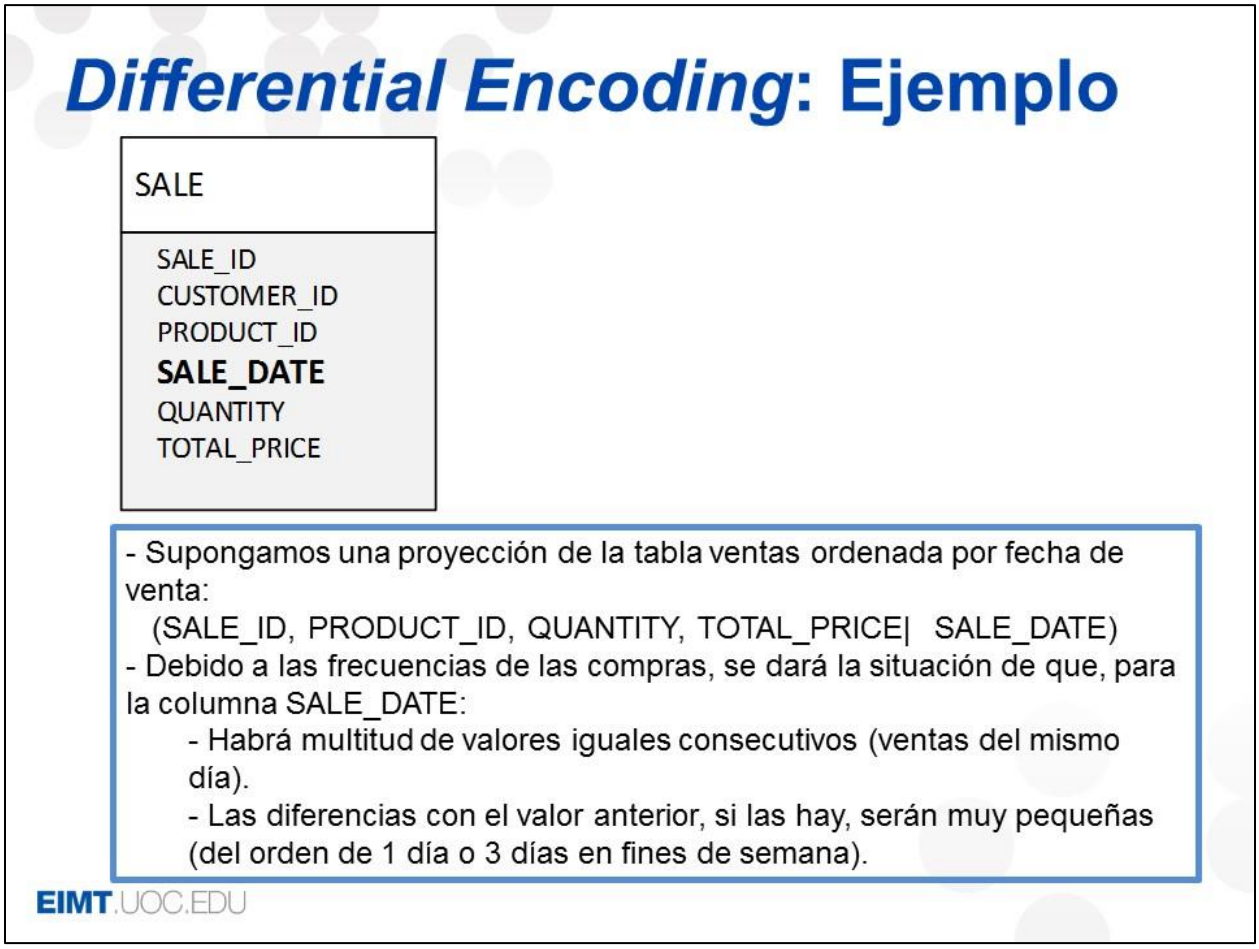
El algoritmo *Differential Encoding* comprime los datos de una columna reduciendo el tamaño necesario para representar sus valores. No obstante, en este caso, por la forma de realizar la compresión se obtiene un mejor ratio de compresión cuando hay repeticiones y los valores están ordenados.

Este algoritmo es aplicable cuando los valores de una columna siguen una secuencia, ya sea creciente o decreciente. Cuando esto ocurre, podemos utilizar un algoritmo parecido al *Frame of Reference Encoding*, pero utilizando el valor de la celda anterior como valor de referencia. En consecuencia, sería como utilizar un *Frame of Reference Encoding* pero con un valor de referencia dinámico que cambia para cada valor de la columna. En resumen, la filosofía no es almacenar la distancia al valor de referencia, sino almacenar la diferencia con el valor anterior.

Este algoritmo tiene ratios de compresión muy altos cuando comprimen fechas, marcas de tiempo (*timestamps*) u otras secuencias de valores que están ordenadas.

El algoritmo siempre empieza replicando el valor de la primera celda de la columna original en la columna comprimida. A partir de ahí, se empieza a realizar la compresión. En las siguientes celdas se calcula el valor comprimido de la celda a partir de la diferencia entre su valor y el valor de la celda anterior. En el caso de que los datos estén ordenados es muy probable que las diferencias sean mínimas, y en consecuencia, se podrá reducir mucho el tamaño requerido. Al igual que en el algoritmo

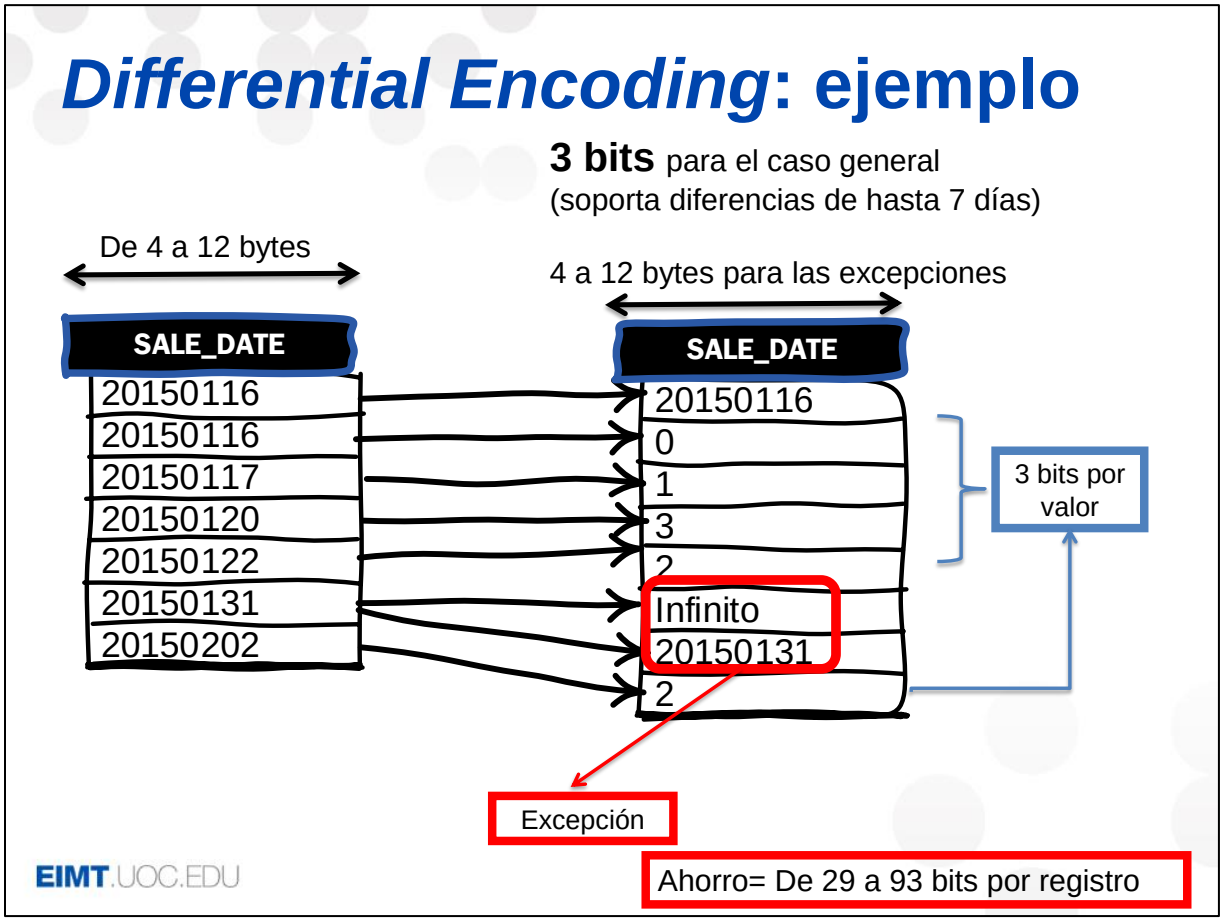
anterior, es posible que haya valores extremos (*outliers*), es decir, valores que están lejos del valor de referencia, y en consecuencia fuera de la ventana establecida. En este caso también se establece un marco de referencia que indica qué valores serán considerados extremos. Los valores extremos se pueden representar siguiendo una estrategia similar al algoritmo anterior (el *Frame of Reference Encoding*).



A continuación vamos a ver cómo funciona el algoritmo *Differential Encoding* a través de un ejemplo. Esta vez lo haremos mediante la tabla *SALE*, que si recordáis almacena las ventas realizadas. En particular, procederemos a comprimir la columna *SALE_DATE*, que indica la fecha cuando se efectúa la venta. Adicionalmente, supondremos que los datos están ordenados ascendentemente en función de la fecha de venta.

Teniendo en cuenta que las fechas están ordenadas, los valores de la columna seguirán una secuencia ascendente, minimizando las distancias entre valores consecutivos dentro de la columna. Este hecho, junto con el volumen de ventas de la empresa (es de esperar que haya más de una venta por día) implicará qué, en la mayoría de los casos, la distancia entre una fecha y la siguiente será muy pequeña. De 0 días en el caso general, y de 1 día a 3 días (como mucho) en el caso de los fines de semana.

Teniendo en cuenta todo esto, vamos a ver cómo se comprime esta columna mediante la compresión *Differential Encoding*.



Tal y como hemos comentado, la columna *SALE_DATE* estaría ordenada ascendentemente. Al haber más de una venta por día, la distancia entre valores contiguos sería de 0 en la mayoría de los casos. En este ejemplo hemos añadido más distancia entre fechas para ver ejemplos más distintos.

Lo primero que hay que hacer es identificar el marco de referencia, es decir, elegir a partir de qué distancia consideraremos los elementos como elementos atípicos. En este caso hemos decidido, de forma arbitraria, que la distancia máxima entre fechas será 8. Es decir, las fechas con más de 7 días de diferencia se representarán como excepciones.

Al tener un marco de referencia de 8 días, necesitamos sólo 3 bits para representar los valores de la columna comprimida para el caso general. En los casos excepcionales, necesitaremos el mismo tamaño que la fecha original (de 4 a 12 bytes, que es lo que puede ocupar una fecha en función de su tipo).

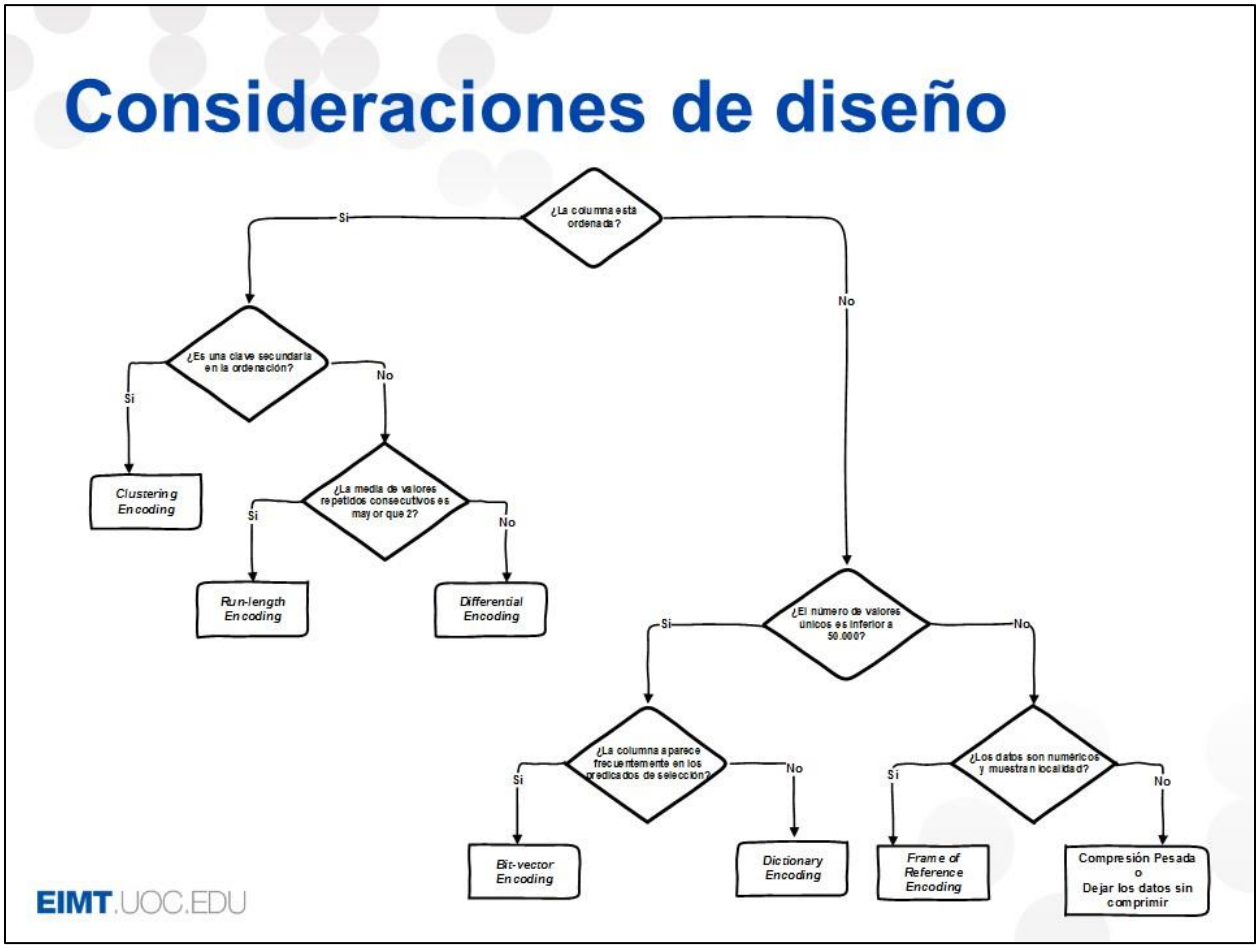
Tomando como base la columna de la parte izquierda de la figura, la compresión se realizaría de la siguiente forma. Se copiaría el primer valor de la columna en la columna comprimida. Para calcular el segundo valor se calcularía la diferencia entre el primer valor de la columna (*16 de enero de 2015*) y el segundo (*16 de enero de 2015*). Como ambas fechas son iguales, la diferencia es 0, y se añadiría el valor *0* en la segunda celda de la columna comprimida. El tercer valor de la columna original (*17 de enero de 2015*) es un día posterior al *16 de enero de 2015* (el segundo valor de la columna original). En

consecuencia, se insertará un *1* en la tercera celda de la columna comprimida. El nuevo valor de referencia es la fecha *17 de enero de 2015*, que es 3 días anterior al valor de la cuarta celda de la columna (*20 de enero de 2015*). Por lo tanto, se asignaría un *3* en la cuarta celda comprimida. El mismo procedimiento se usaría para la quinta celda, que implicaría escribir el número *2* en la columna comprimida.

Por su parte, el valor de la sexta celda supone una excepción, ya que su diferencia con el valor de la celda anterior es *9*, que es superior a *7*, el valor máximo que podemos representar con 3 bits. En este caso, se indicaría que es una excepción, y se almacenaría el valor original de la celda, pasando a ser este valor el nuevo valor de referencia. En el siguiente caso, el valor corresponde al *2 de febrero del 2015*, que difiere 2 días del valor de referencia. En consecuencia, se almacenaría el valor *2* en la celda comprimida y finalizaría el proceso de compresión.

Al no saber con certeza el número de excepciones que nos vamos a encontrar, tampoco tiene mucho sentido en este caso analizar la ganancia total (en términos de ahorro de espacio de almacenamiento) de la compresión de la columna. De nuevo, como en el caso del *Frame of Reference Encoding*, creemos más útil indicar la ganancia por celda. Para cada valor original de la columna dentro del marco de referencia, tendremos un ahorro de 29 a 93 bits (en función de cómo se haya almacenado la fecha en la columna original).

Una vez se conocen los algoritmos de compresión más representativos es importante saber cuándo es aconsejable aplicar cada uno de ellos. A continuación vamos a presentar algunas reglas básicas basadas en las referencias presentadas al final de la presentación.



Tal y como se ha apuntado en estos materiales didácticos, cuando se usan métodos de compresión ligera, es muy importante conocer las características de los datos a comprimir para garantizar una buena compresión de los mismos. Recordemos que en los casos extremos, los algoritmos de compresión pueden tener el efecto contrario al deseado, y pueden generar columnas de más tamaño si se aplican en contextos inadecuados.

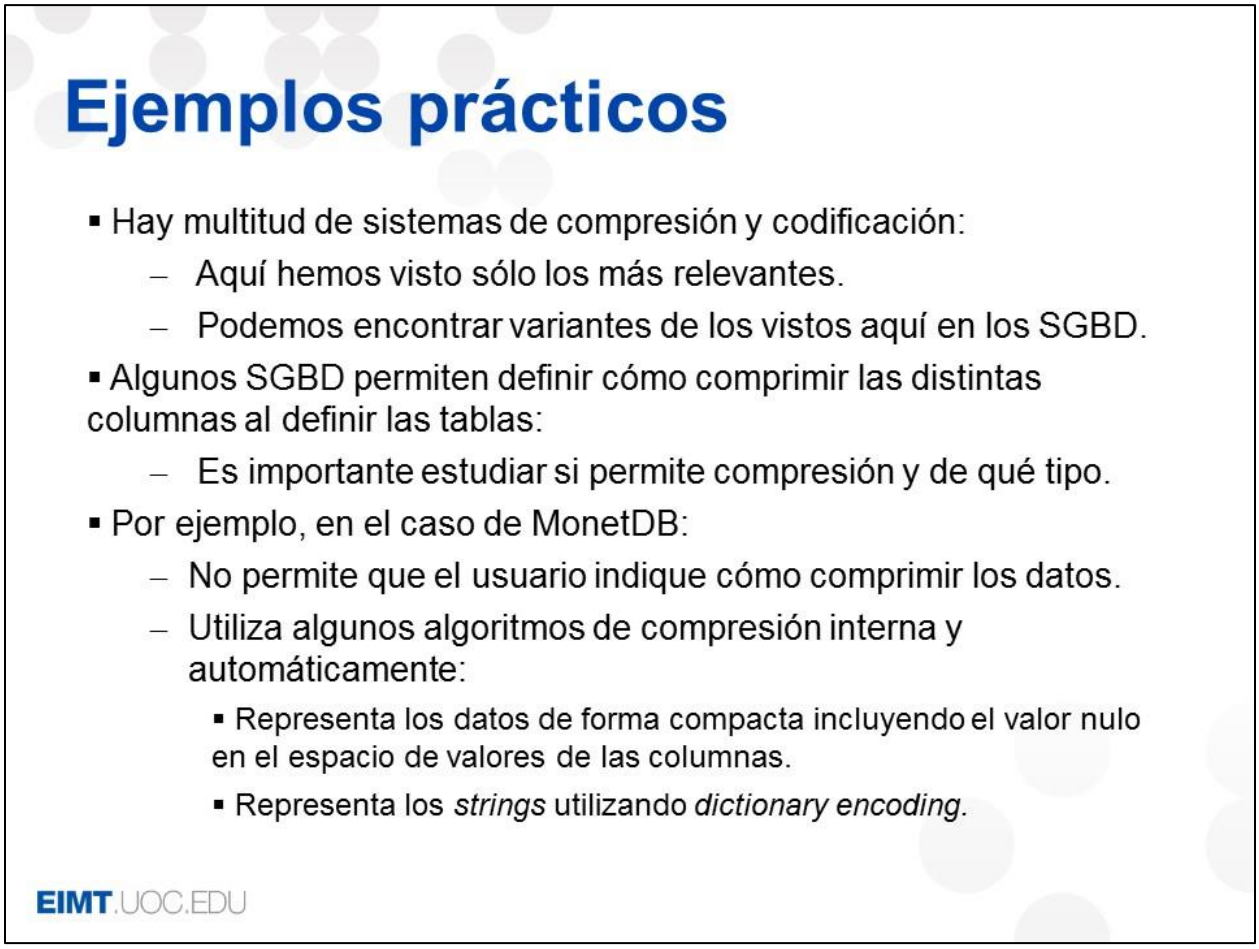
El árbol de decisión presentado en esta transparencia permite, de forma rápida, hacerse una idea de las características principales a tener en cuenta antes de escoger qué algoritmo de compresión es el más adecuado para cada columna. Este árbol ha sido adaptado del trabajo de tesis de Daniel Abadi y enriquecido con las sugerencias y comentarios de los autores indicados en el apartado de referencias bibliográficas que tenéis al final de esta presentación. Se trata de consideraciones generales que será necesario matizar en cada caso o contexto concreto. También pueden cambiar con el trascurso del tiempo, ya que algunas de las condiciones propuestas indican valores constantes que pueden verse condicionados por avances en la tecnología (*hardware*) y los SGBD (*software*).

Dicho esto, la primera pregunta que debemos hacernos antes de decidir cómo (o si debemos) comprimir una columna es la siguiente: ¿la columna está ordenada?. En caso afirmativo, parece que podremos aprovechar la localidad de datos y repeticiones para comprimir la columna (parece interesante usar *Run-Length Encoding*, *Cluster Encoding* y *Differential Encoding*).

A partir de aquí, habría que estudiar cada situación más a fondo, pero algunas reglas que podríamos considerar serían las siguientes: en el caso de que la columna no sea la clave primaria en la ordenación, sino la clave secundaria, puede ser interesante utilizar *Cluster Encoding*. Un ejemplo sería cuando queremos comprimir una columna que indique el apellido de nuestros clientes, y la tabla de clientes (*Customer*) estuviera ordenada primero por el nombre y después por apellidos. En el caso de que la media de valores consecutivos sea mayor que 2, la bibliografía propone utilizar *Run-Length Encoding*. En el caso contrario, se aconseja utilizar *Differential Encoding*.

Cuando los datos de la columna no estén ordenados, no podemos garantizar que los valores iguales aparecerán de forma consecutiva en las estructuras de almacenamiento, y en consecuencia, no tenemos garantías de que los algoritmos anteriores funcionen adecuadamente. En estas situaciones habría que considerar el número de valores únicos que aparecen en la columna. En el caso de que el número de valores no sea muy elevado (se considera que el número de valores es muy elevado cuando supera los 50.000 valores distintos) será factible utilizar compresión utilizando diccionarios y mapas de bits. El uso de uno u otro dependerá básicamente de si la columna se utiliza frecuentemente en los predicados de selección (por ejemplo, en las cláusulas `WHERE` de las sentencias SQL que se ejecuten sobre la BD). Si esto fuese así, de acuerdo a la bibliografía, la mejor alternativa sería el uso del algoritmo *Bit-Vector Encoding*. En el caso contrario, se propone usar *Dictionary Encoding.*

En columnas no ordenadas donde haya más de 50.000 valores distintos, no se recomienda usar diccionarios ni vectores de bits. Gestionar más de 50.000 vectores de bits o diccionarios de muchos elementos puede ser muy complejo, y puede exceder el espacio que se puede alojar en memoria, complicando la gestión de los datos. En estos casos se propone utilizar el algoritmo *Frame of Reference Encoding* cuando los datos sean numéricos y muestren localidad. En el caso contrario, se propone dejar los datos sin comprimir o usar algoritmos de compresión pesada.

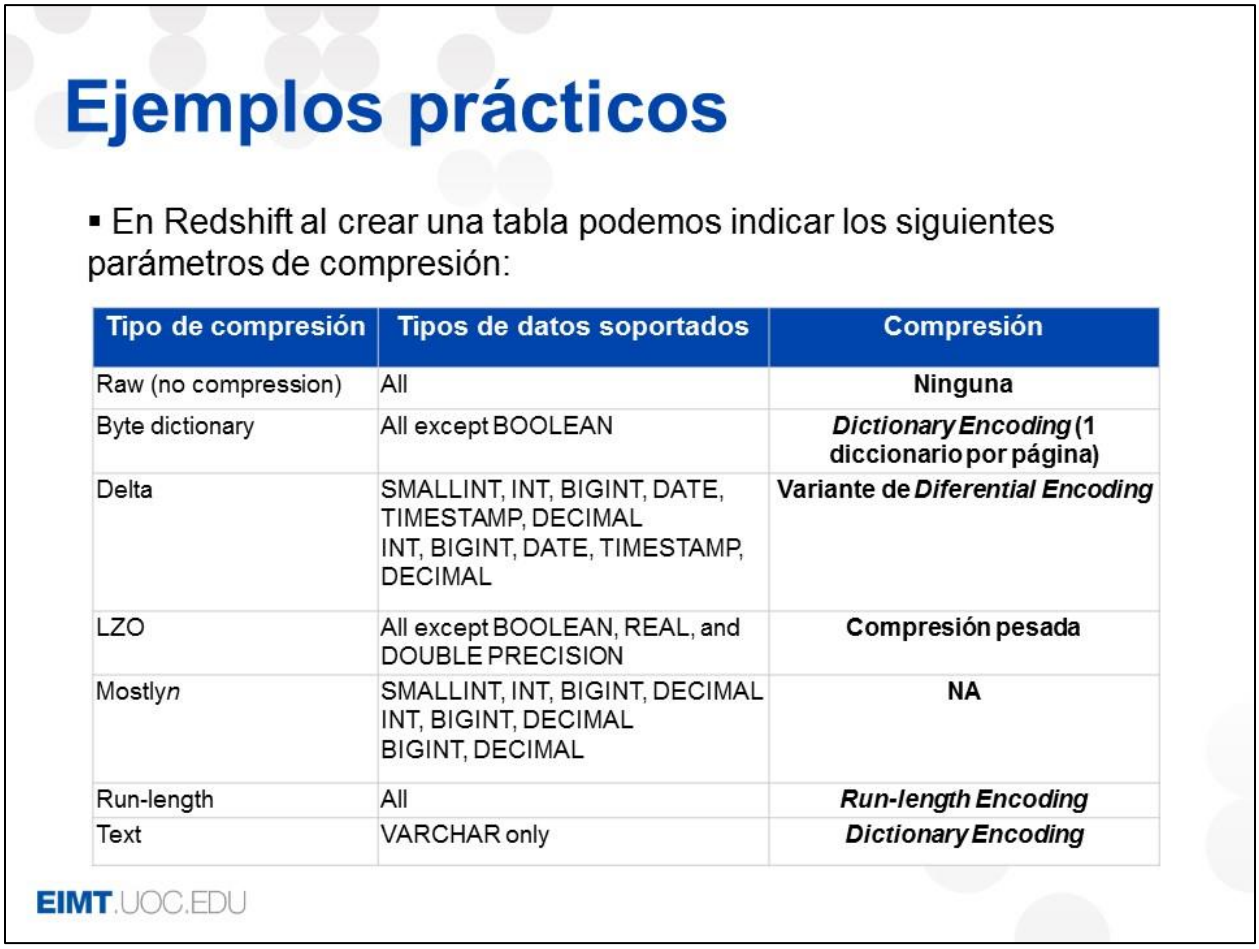


Ya para acabar, vamos a hablar un poco de cómo podemos encontrar estos algoritmos de compresión en los SGBD con los que trabajemos. Lo primero es tener en cuenta que aquí hemos visto algunos algoritmos de compresión, pero que existen muchos más y no sólo eso, sino que existen muchas versiones distintas de cada algoritmo. De hecho, es difícil encontrar en los SGBD exactamente la misma implementación de cada uno de los algoritmos de compresión que hemos explicado aquí.

En algunos casos, los SGBD utilizan algoritmos de compresión de forma automática sin necesidad de que el usuario intervenga. Algunos ejemplos son los vectores de bits que puedan utilizar para la optimización de consultas distintos SGBD (ya sean almacenes de filas o de columnas), como PostgreSQL, por ejemplo.

En otros casos, el usuario no interviene en la elección de los algoritmos de compresión a utilizar. En otras palabras, los almacenes de columnas pueden utilizar compresión internamente, y sin control del usuario, para mejorar su eficiencia. Un ejemplo es MonetDB, que no permite que el usuario indique cómo comprimir los datos, pero que internamente aplica algunos algoritmos de compresión de forma automática. Uno de ellos es utilizar una variante del *Dictionary Encoding* para representar las columnas que tiene asociado un tipo de datos *string*.

Finalmente, los SGBD también pueden permitir que el diseñador y el administrador de la BD pueda especificar los algoritmos de compresión a aplicar a las columnas en el momento de creación de las tablas, o con posterioridad (sobre la base de que inicialmente se decidió que los datos de la tabla no estuvieran comprimidos). En estas situaciones, es importante identificar qué opciones de compresión permite el SGBD y analizarlas a fondo para ver en qué casos son adecuadas. Para cada algoritmo soportado habrá que ver qué margen de maniobra ofrece al usuario para indicar los parámetros de compresión (si permite definir el valor y tamaño del marco de referencia, el tamaño del diccionario, etc.). A continuación, y a modo de ejemplo, vamos a ver los distintos algoritmos de compresión que permite utilizar Redshift.



Tal y como se ha comentado en Redshift, al definir una tabla, podemos indicar si queremos que los datos de las distintas columnas se guarden comprimidos y bajo qué esquema de compresión. En la transparencia se pueden observar los tipos (o algoritmos) de compresión disponibles en este SGBD, sobre qué tipo de datos se pueden aplicar, y a cuál de los algoritmos que hemos visto en esta presentación se parecen más.

El *Byte Dictionary*, por ejemplo, permite crear un diccionario con 255 entradas por cada página de disco. Por lo tanto, y según la documentación del fabricante, sería una adaptación del algoritmo *Dictionary Encoding* donde hay un diccionario por página y con un límite de entradas. La compresión de tipo *Text* funciona de forma similar, pero permite aumentar el número de entradas del diccionario.

El algoritmo de compresión *delta* parece ser una adaptación del *Differential Encoding* donde el espacio de valores puede ser de 1 a 2 bytes, según se indique.

El método de compresión *Mostly* es parecido al *Frame of Reference* pero sin un valor de referencia, y se usa cuando los tipos de datos de una columna permiten un rango de valores superior al que realmente se almacenan en la columna. En dicho caso, el SGBD permite redefinir la columna como *MostlyN*, donde *N* indica el número de bits que se utilizará en la compresión (8, 16 o 32), y el SGBD utilizará menos bits

para representar cada uno de los valores cuando sea posible. Cuando no sea posible, se almacenará el valor original.

Por último, el método *Run-Lenght* de Redshift es una adaptación del algoritmo *Run-Length Encoding* explicado en este material didáctico.

Para el caso en que se decida que se quiere aplicar compresión de datos en el momento de creación de tabla, pero se desea que la elección de los esquemas de compresión sea decidida automáticamente por el SGBD, en el caso de Redshift, es necesario ejecutar el comando `COPY` con la opción `COMPUPDATE` activada (es decir, con opción `ON`) en el momento de insertar los datos en la nueva tabla.

Cuando se decide especificar manualmente los esquemas de compresión, y para ayudar en la toma de decisión, los SGBD incorporan utilidades que asisten al diseñador y/o el administrador de la BD. En el caso concreto de Redshift, por ejemplo, el comando `ANALYZE COMPRESSION`, sugiere esquemas de compresión para las columnas de una tabla que contiene datos. Puede ser utilizado para decidir los esquemas de compresión de una tabla para la que inicialmente se había decidido que los datos no estuviesen comprimidos, o para decidir los esquemas de compresión de una nueva tabla que tiene características similares a la que se examina.

Y hasta aquí estos materiales sobre compresión de datos en almacenes de columnas. Hemos visto lo que puede aportar la compresión de datos en los almacenes de columnas, qué tipos de compresión de datos existen y cuáles son los más usados en los almacenes de columnas. Después hemos explicado cómo funcionan algunos de los algoritmos más representativos para comprimir datos en los almacenes de columnas y qué criterios de diseño hay que tener en cuenta para escoger los algoritmos más adecuados en función de las características de cada columna. Finalmente, hemos visto un par de ejemplos de cómo SGBD concretos utilizan compresión de datos.

Esperamos que hayáis encontrado los materiales amenos, útiles y el tema interesante. A continuación encontraréis un conjunto de referencias relevantes sobre el tema, por si os interesa profundizar más en estas cuestiones.

Referencias

D.J. Abadi (2008). *Query Execution in Column-Oriented Database Systems*. PhD Dissertation in Computer Science and Engineering at the MIT. Advisor: Samuel Madden. Chapter 4.

D.J. Abadi, S. Madden, N. Hachem (2008). Column-stores vs. Row-stores: How different are they really? *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 967–980.

S. Harizopoulos, D. Abadi, Peter Boncz (2009). Column-oriented Database Systems. VLDB 2009 Tutorial. Transparencias 49 a 60.

D.J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden (2012). The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3), pages 197-280.

H. Plattner (2013). *A Course in In-Memory Data Management*. Editorial Springer. Capítulos 6 y 7.

G. Harrison (2015). *Next Generation Databases*. Editorial Apress. Capítulo 6.

EIMT.UOC.EDU

¡Qué tengáis un buen día!