Almacenes de columnas

Procesamiento de consultas

Jordi Conesa i Caralt
M. Elena Rodríguez González

EIMT.UOC.EDU

Bienvenidos a la cuarta presentación de la asignatura de Bases de datos para *Data Warehouse* donde explicaremos cómo se gestionan las consultas en los almacenes de columnas.

Procesamiento de consultas

- Operar con datos comprimidos
- Materialización tardía
- Modelos de ejecución de consultas
- *Database Cracking*

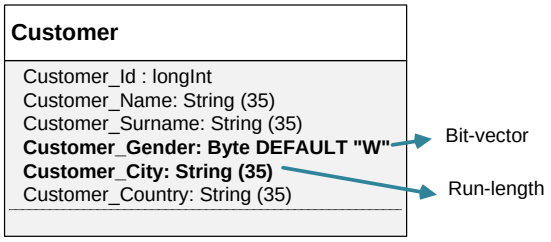
EIMT.UOC.EDU

Esta presentación está dividida en 4 partes:

- Operar con datos comprimidos: en algunos casos los almacenes de columnas pueden realizar las operaciones sobre la base de datos (BD) directamente sobre los datos comprimidos. Veremos las ventajas que esto tiene, así como algunos ejemplos donde se opera directamente sobre datos comprimidos.
- Materialización tardía: en este apartado se verán algunas estrategias utilizadas por los almacenes de columnas para resolver las consultas formuladas sobre la BD. En particular, se hablará de la materialización temprana, de la materialización tardía, de las ventajas de una u otra opción en los almacenes de columnas, y de algunas técnicas de optimización para cuando se utiliza materialización tardía.
- Modelos de ejecución de consultas: en esta sección se presentarán los distintos modelos de ejecución de consultas que un sistema gestor de la base de datos (SGBD) puede utilizar para resolver las consultas, y los contextualizaremos para el caso particular de los almacenes de columnas. Existen 3 modelos: los modelos que operan sobre valores individuales (registro a registro), los que operan directamente sobre las columnas en bloque (materialización completa) y los que operan sobre fragmentos de columnas (vectorización).
- Finalmente hablaremos del *Database Cracking* o índices adaptativos, que son un mecanismo que nos permite indexar de forma dinámica los datos de la BD en función del tipo de consultas que se formulan.

Operar con datos comprimidos

- Desaparece el sobrecoste de la CPU.
- Reduce la cantidad de datos a mover a la CPU.
- Permite ejecutar operaciones sobre múltiples registros a la vez.
- Los operadores son más complejos.




```
SELECT Customer_Gender, count(*)
  FROM Customer
 WHERE Customer_City = "Jaramillo Quemado"
GROUP BY Customer_Gender
```

EIMT.UOC.EDU

Tal y como ya hemos comentado en otras presentaciones, en algunos casos, los almacenes de columnas permiten que las operaciones que necesita ejecutar el SGBD para responder a las peticiones formuladas por los usuarios se puedan resolver directamente sobre datos comprimidos. Las ventajas de operar directamente sobre datos comprimidos son las siguientes:

1) Desaparece el sobrecoste (o sobrecarga) de CPU necesario para descomprimir los datos antes de operar sobre ellos.
2) Se reduce la cantidad de datos que se desplazan (o se transfieren) desde la memoria a la CPU para operar con ellos.
3) Se puede ejecutar la operación con menos ciclos de CPU, ya que permite ejecutar operaciones sobre múltiples valores (registros) a la vez.

No obstante, se debe tener en cuenta que procesar directamente datos comprimidos tiene un coste en la implementación de los operadores (u operaciones) de la BD, que acabarán siendo más complejos y menos escalables. Vamos a explicar el porqué con un par de ejemplos. Supongamos que queremos añadir un esquema de compresión al SGBD (un *dictionary* encoding, por ejemplo). En dicho caso, los operadores unarios (por ejemplo, la proyección) deberán ser capaces de operar sobre columnas no comprimidas y sobre columnas comprimidas mediante *dictionary encoding*. Por lo tanto, deberemos crear dos versiones de cada operador unario, una versión del operador que trabaje sobre columnas no comprimidas y otra versión que opere sobre columnas comprimidas mediante el algoritmo *dictionary*

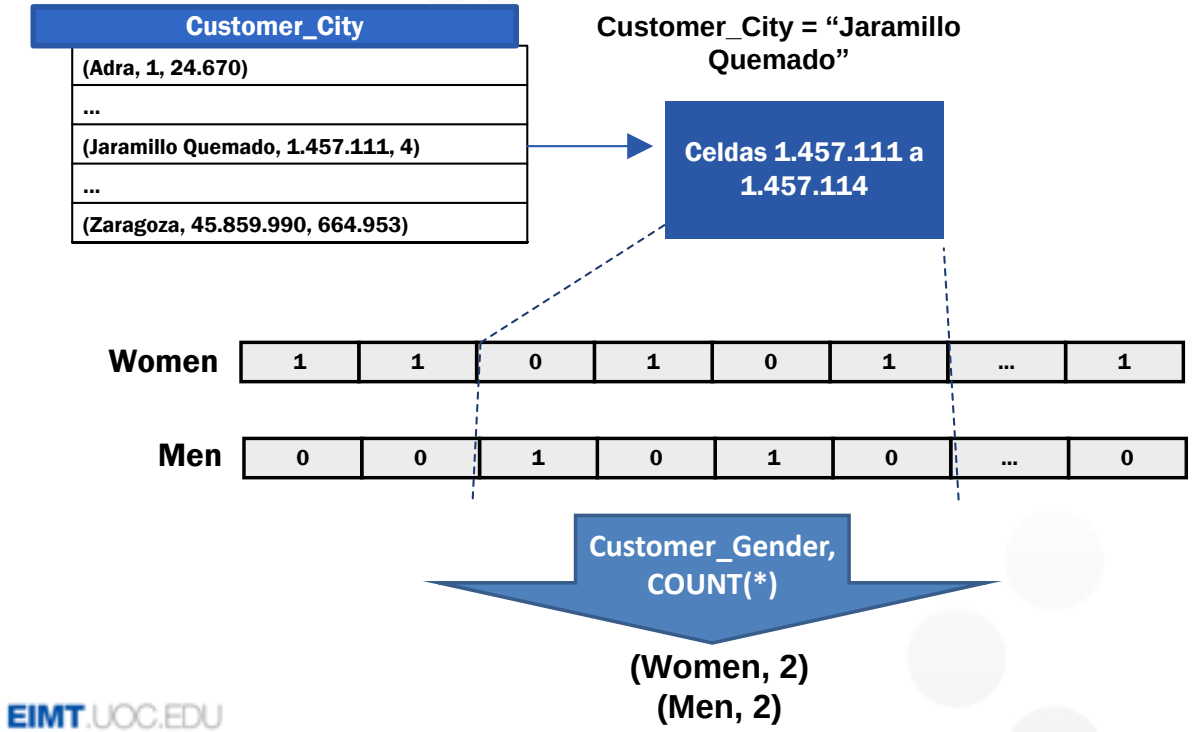
encoding. Supongamos ahora que queremos añadir un nuevo esquema de compresión (por ejemplo, un *run-length encoding*). En dicho caso, los operadores unarios deberían tener una nueva versión que permita operar con columnas comprimidas mediante *run-length encoding*. Como norma general, si *N* es el número de esquemas de compresión soportados en un SGBD, cada operador unario deberá tener *N+1* implementaciones, una para cada esquema de compresión y una para operar sobre datos no comprimidos.

En el caso de los operadores binarios (como sería el caso, entre otros, de la resolución de operaciones de combinación –o *join*–) el tema es mucho más dramático, ya que el número de implementaciones de cada operador será $(N+1)^2$, que son las posibles combinaciones de tipos de codificación de entrada por cada operador. Está claro, pues, que los costes en implementación y su futuro mantenimiento no son despreciables. Cabe decir que los números proporcionados apuntan a los casos peores. En algunos casos, es posible utilizar una misma implementación para distintos esquemas de compresión. También existen algunas técnicas, como por ejemplo, la de los *compression blocks* (ver trabajo de tesis de Abadi referenciado en esta presentación) que permiten simplificar las implementaciones de los operadores y reducir el número de versiones a implementar. No obstante, aún con estas mejoras, las implementaciones de los operadores continúan siendo más complejas que sus homólogas que tratan con datos no comprimidos.

Otra pregunta (relacionada con lo que acabamos de discutir) que podemos plantearnos es, ¿qué tipo de operaciones son capaces de trabajar con datos comprimidos? Según la bibliografía (ver Graefe y Saphiro y la tesis de Abadi), las operaciones de *natural join* pueden realizarse sobre datos comprimidos siempre y cuando los participantes en la operación de *natural join* utilicen el mismo esquema de compresión. Una posibilidad para garantizar dicha condición en un SGBD, podría ser requerir que todos los atributos (o columnas) de un mismo dominio (es decir, que representen un mismo concepto del mundo real) utilicen el mismo esquema de compresión. Otras operaciones permitidas son las búsquedas por valor (presentes en las operaciones de selección, por ejemplo), que podrán ejecutarse sobre datos comprimidos cuando el esquema de compresión utilizado preserve el orden de los elementos. Finalmente, la proyección y la eliminación de duplicados también pueden ejecutarse sobre datos comprimidos. El operador de agrupación parece no poder ejecutarse siempre sobre datos comprimidos. El atributo (o columna) a agrupar sí que puede generarse a partir de datos comprimidos, pero no hay garantía de que el atributo (o atributos) sobre el que se realicen las operaciones de agrupación (ejecución de operaciones `COUNT`, `SUM`, `AVG`, etc.) pueda calcularse mediante datos comprimidos.

Una vez explicadas las ventajas y desventajas de trabajar con datos comprimidos, vamos a ver un ejemplo donde una sentencia `SELECT` opera directamente sobre datos comprimidos. El ejemplo utilizará la tabla de clientes (*Customer*) que hemos usado a lo largo de este material didáctico dedicado a almacenes de columnas. La operación a realizar será una consulta que nos devuelva el número clientes, agrupados por género, de la población de Jaramillo Quemado (se ha escogido esta población porque sólo tiene 4 habitantes). Para el ejemplo, supondremos que la columna *Customer_Gender* está comprimida mediante *bit-vector encoding* y la columna *Customer_City* mediante *run-length encoding*.

Operar con datos comprimidos: ejemplo



EIMT.UOC.EDU

Tal y como se ha comentado, la columna *Customer_City* contendrá los datos de la ciudad donde viven los clientes de la empresa comprimidos mediante un esquema de *run-length encoding*. Recordemos que la empresa tiene 46 millones de clientes (la población española). Supongamos que la columna está ordenada alfabéticamente por población. Por lo tanto, cada celda de la columna estará compuesta por una tripleta con el nombre de una ciudad, el número de celda de su primera ocurrencia en la columna original y el número de veces que se repite. Al estar ordenada alfabéticamente por el nombre de ciudad, sólo habrá una tripleta para cada ciudad.

Por otro lado, tal y como vimos en la presentación de compresión de datos, la columna *Customer_Gender* se ha comprimido mediante dos vectores de bits con tantas posiciones como registros (o filas) tiene la tabla *Customer*. El primer vector, llamado *Women*, contiene un valor binario en cada celda que indica si el cliente asociado al número de registro *i* de la tabla es una mujer (valor 1 para la posición *i*-ésima) o no (valor 0 para la posición *i*-ésima). El vector *Men* contendrá un valor binario que indique lo contrario, si el cliente es un hombre.

En el caso que nos ocupa, el SGBD podría ejecutar la consulta directamente sobre los datos comprimidos. En la transparencia se muestra de forma simplificada cómo se realizaría. Primero, se seleccionaría la posición de los registros (o filas) de la tabla que satisfagan la condición indicada en el

`WHERE` (clientes que viven en Jaramillo Quemado). Al estar ordenada la columna *Customer_City*, el SGBD podría obtener eficientemente la celda asociada a la ciudad de Jaramillo Quemado:

(Jaramillo Quemado, 1.457.111,4)

Supongamos que el resultado es la celda anterior. Asumiendo que el algoritmo que implementa la operación de selección es capaz de interpretar datos comprimidos mediante *run-length encoding*, el SGBD interpretará que los clientes de Jaramillo Quemado son 4 y están ubicados en las celdas 1.457.111, 1.457.112, 1.457.113 y 1.457.114. Para obtener el resultado de la consulta, el SGBD deberá contar cuantos hombres y mujeres hay en dichas posiciones y devolver ese valor. Para hacer este cálculo, asumiendo que es capaz de operar con datos comprimidos mediante *bit-vector encoding*, simplemente debe contar los unos (1) que hay en las posiciones 1.457.111, 1.457.112, 1.457.113 y 1.457.114 de los vectores *Women* y *Men*. El resultado es *Men* 2 y *Women* 2. Ese sería justamente el resultado que devolvería la consulta.

Tal y como se puede apreciar en este ejemplo, operar sobre datos comprimidos resulta ser eficiente, ya que evitaría descomprimir las columnas *Customer_City* y *Customer_Gender*. Además, permite reducir el número de operaciones a realizar, ya que se han podido seleccionar 4 registros haciendo una sola comparación sobre la celda comprimida. También reduce la cantidad de datos que se deben mover desde el dispositivo de almacenamiento externo (disco) a la CPU.

Para resolver la consulta sobre datos comprimidos, el SGBD debería tener implementada una versión del operador de comparación (que resuelve la operación de selección planteada en la consulta, *Customer_City="Jaramillo Quemado"*) que soporte un esquema de compresión de *run-length encoding*. Por otro lado, la operación proyección (que proyecta la columna *Customer_Gender*) y la operación `COUNT` deberían permitir operar sobre datos comprimidos mediante *bit-vector encoding*.

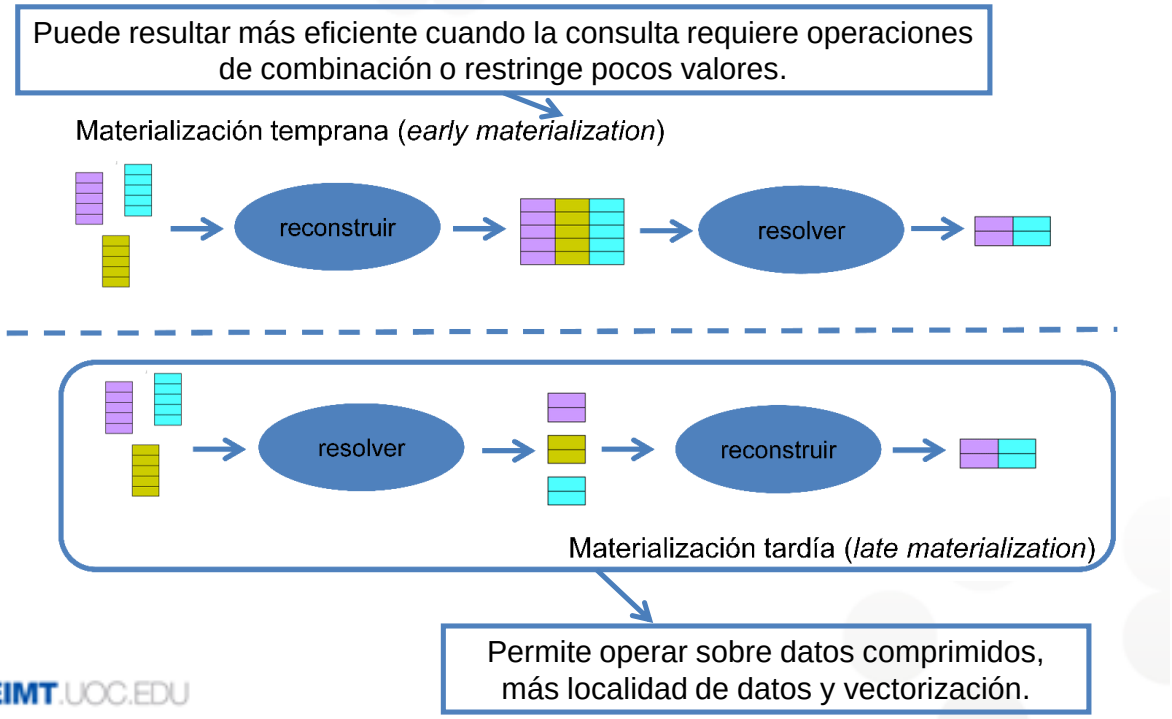
Procesamiento de consultas

- Operar con datos comprimidos
- **Materialización tardía**
- Modelos de ejecución de consultas
- *Database Cracking*

EIMT.UOC.EDU

Bien, una vez visto cómo operar con datos comprimidos vamos a ver cómo se pueden procesar las consultas formuladas sobre la BD en los almacenes de columnas. Para ello, empezaremos viendo los distintos tipos de materialización que existen, y cuál de ellos es el más adecuado en los almacenes de columnas. Recordad que la materialización es el proceso que reconstruye las filas de interés de una tabla a partir de sus columnas.

Materialización temprana y tardía



Tal y como ya hemos comentado en este material didáctico (en concreto, en la presentación dedicada a las características de los almacenes de columnas), la materialización tardía es una de las características propias de los almacenes de columnas que puede mejorar el rendimiento de la BD, en el caso de utilizarse adecuadamente.

El hecho es que, en un almacén de columnas, los datos de cada tabla se almacenan por columnas, y en diversas ubicaciones en disco (en definitiva, cada columna de una tabla, al menos, se almacena en un fichero separado sin mezclarse con datos de otras columnas). Sin embargo los usuarios, los programas e incluso los *drivers* (ADO.NET, JDBC, etc.) que acceden a la BD esperan que el resultado de una consulta sea un conjunto de filas. Por lo tanto, en el plan de acceso de la mayoría de consultas, los datos de varias columnas se deberán combinar en filas de información. Este proceso de reconstrucción de filas se denomina materialización de filas, y es muy similar (desde un punto de vista conceptual) a una operación de combinación.

Cabe destacar que en el caso de los almacenes de columnas, al materializar las filas, se deben descomprimir los datos cuando los algoritmos de compresión aplicados hayan sido de los que reducen el número de celdas de la columna (este es el caso, por ejemplo, de los esquemas de compresión *run-length encoding* y *cluster encoding*).

La decisión sobre cuándo realizar el proceso de materialización de filas en almacenes de columnas es similar a la decisión de cuándo ejecutar la ejecución de operaciones de proyección en los almacenes de

filas, que también puede realizarse de forma temprana durante el proceso de ejecución de una consulta o de forma más tardía.

La cuestión fundamental es en qué momento, durante el proceso de resolución de consulta, se realiza la reconstrucción o materialización de filas. En principio, existen dos enfoques: la materialización temprana (en inglés, *early materializacion*) o la materialización tardía (*late materialization* en inglés).

La materialización temprana realiza la reconstrucción de filas a partir de los datos de las columnas relevantes al principio del plan de acceso de la consulta. En este caso, al tener las filas creadas antes de la resolución de la consulta, se pueden aplicar los algoritmos existentes en los almacenes de filas para la resolución de las operaciones indicadas en la consulta (selección, proyección, agregación, combinación, etc.). Si bien esto permite ahorrar costes de desarrollo, desaprovecha algunas de las posibilidades de los almacenes de columnas a efectos de rendimiento. Entre otros, no permite la ejecución de operaciones en bloque (tal y como veremos en la siguiente sección), no permite la ejecución de operaciones directamente sobre datos comprimidos, no aprovecha la localidad de los datos, requiere de más traspaso de datos entre disco, memoria principal y CPU, y en muchos casos materializa registros (o filas) innecesarios, ya que las consultas pueden requerir procesar menos filas (en las operaciones de selección, los predicados que incorporan, pueden reducir el número de filas de salida y las agregaciones combinan distintas filas en una de sola). Por estos motivos, aun siendo más compleja, la materialización tardía se suele utilizar de forma intensiva en los almacenes de columnas.

A bote pronto, podemos pensar que la materialización tardía siempre genera mejores resultados, o sea, que es más eficiente. Esto no es del todo cierto. En algunos casos, la materialización temprana es más conveniente. El principal problema de la materialización tardía es que, en algunos casos, las columnas necesitan ser accedidas múltiples veces en el plan de consulta. Por ejemplo, supongamos que una columna se accede una vez para obtener las posiciones que satisfacen un predicado y una segunda vez (más adelante en el plan) para recuperar sus valores. El sobrecoste puede incrementarse aún más en el caso de que estemos haciendo una combinación y se deba ordenar la columna según la clave de la combinación. En la materialización temprana se genera un conjunto de filas la primera vez que se consulta la columna, eliminando la necesidad de accesos futuros a la columna.

Por lo tanto, y aunque la materialización tardía permite algunas optimizaciones, si el coste de acceder repetidamente a unas mismas columnas es elevado, puede ser más conveniente utilizar materialización temprana. Por otro lado, cuando los predicados que incorpora una operación de selección de una consulta no son muy restrictivos (en otras palabras, generan un número elevado de filas), también puede ser más conveniente utilizar materialización temprana.

Materialización tardía

- Las operaciones se ejecutan directamente sobre las columnas relevantes, generando:
  - Listas de posiciones de los valores de una columna que cumplen una determinada condición:
    - En forma de vectores de bits o a través de conjuntos de rangos de valores
    - Se pueden realizar AND (o NOT o OR) lógicos sobre las listas de posiciones para evaluar condiciones compuestas.
  - Reconstrucción de las columnas

Customer
Customer_Id : longInt
Customer_Name: String (35)
Customer_Surname: String (35)
Customer_Gender: Byte DEFAULT "W"
Customer_City: String (35)
Customer_Country: String (35)

```
SELECT Customer_Name
  FROM Customer
 WHERE Customer_City = "Jaramillo Quemado"
       AND Customer_Gender= "W"
```

EIMT.UOC.EDU

Cuando se utiliza materialización tardía, las operaciones implicadas en el plan de consulta se ejecutan directamente sobre las columnas relevantes. Por norma general, cada operación se ejecuta sobre una o columna (operaciones unarias) o dos columnas (operaciones binarias).

Al aplicar operaciones de selección en una consulta (estas operaciones aparecerán en cláusulas `WHERE`) se suelen generar listas de posiciones que indican el resultado de las operaciones de selección sobre las columnas. Cada lista de posiciones indica las posiciones de los valores de la columna que satisfacen la condición (o predicado) evaluada. Estas listas pueden tener forma de vectores de bits o de conjuntos de rangos de posiciones (*rowid* o identificadores de filas), en función de las operaciones a considerar. En el caso de que tengamos condiciones sobre más de una columna (o sea, en el caso que se incluyan diferentes operaciones de selección en la consulta), se pueden utilizar operaciones *AND* y *OR* lógicas sobre las listas de posiciones (cuando éstas toman la forma de vectores de bits) para calcular rápidamente el conjunto de filas que cumplen las distintas condiciones.

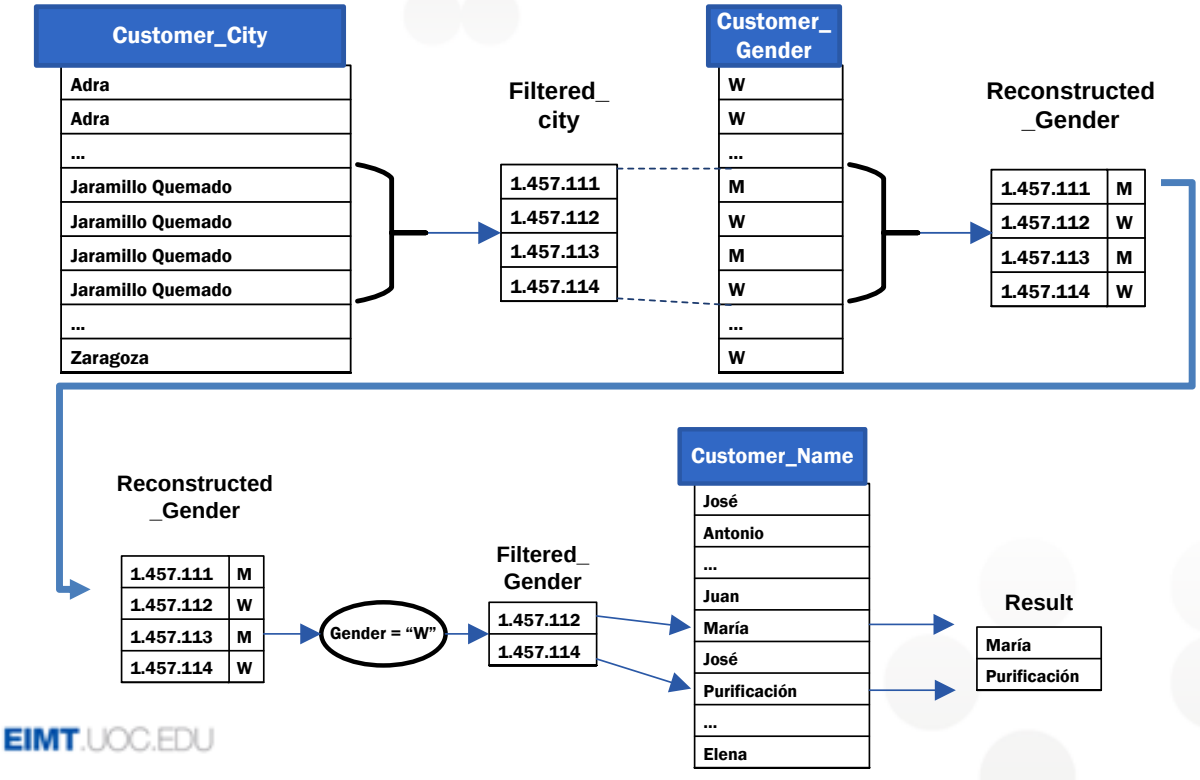
Por ejemplo, supongamos una consulta que realiza un filtro (o sea, impone condiciones) sobre dos atributos (o columnas) y proyecta un tercer atributo de la tabla que cumpla las dos condiciones indicadas. En este caso, se podría generar un vector de bits para cada columna filtrada, indicando las celdas de cada columna que satisfacen la condición (con un 1). Posteriormente, se realizaría un *AND* lógico de ambos vectores de bits para obtener un nuevo vector de bits que representa las celdas de la tercera

columna que deben ser seleccionadas en la consulta. Al final de la consulta, y antes de devolver el resultado, se combina el vector de bits que resulta de la operación de *AND* lógico con la columna (o atributo) de interés en la consulta para generar el resultado de la consulta (un conjunto de filas que contienen un único valor). Al retrasar la creación de filas tan tarde como sea posible, se consigue mayor localidad de datos, aumentar la posibilidad de operar sobre datos comprimidos, de operar con datos en bloque y de evitar generar filas innecesarias.

Cada vez que se deben consultar los valores de una columna de acuerdo a una lista de posiciones decimos que se realiza una reconstrucción de la columna. En la reconstrucción se crea una nueva columna que constituye un subconjunto de la columna original, puesto que contiene sólo los valores que cumplen las condiciones representadas por la lista de posiciones. Las reconstrucciones de columnas suelen ejecutarse múltiples veces dentro del plan de cada consulta.

A continuación vamos a ver un ejemplo de materialización tardía utilizando nuestra BD de ejemplo. En este ejemplo supondremos que se realiza una consulta que pretende obtener los nombres de las clientas que viven en la ciudad Jaramillo Quemado. Para simplificar el ejemplo supondremos que los datos no están comprimidos, aunque esta consulta podría realizarse igualmente sobre datos comprimidos.

Materialización tardía



En el caso de utilizar materialización temprana, lo primero que haríamos al ejecutar la consulta sería la materialización. Por lo tanto, se crearían más de 46 millones de filas de clientes con los valores de las columnas *Customer_City, Customer_Gender* y *Customer_Name*, que son las tres columnas implicadas en la consulta que se pretende resolver. A partir de ese momento, se ejecutarían las diferentes operaciones de la consulta (en nuestro ejemplo se trata de operaciones de selección y proyección).

No obstante, si utilizáramos una estrategia de materialización tardía, primero ejecutaríamos las operaciones de la consulta sobre las columnas relevantes y, cuando tuviéramos el resultado esperado, podríamos materializar las filas que conforman el resultado final de la consulta (sobre nuestro ejemplo, y dado que sólo proyectamos los datos de la columna *Customer_Name*, cada fila contendría un solo valor). Partiendo del supuesto de que usamos materialización tardía, podríamos descomponer la consulta de ejemplo en las siguientes operaciones (cabe tener en cuenta que podrían existir otras estrategias de resolución, con diferente propuesta de operaciones, igualmente válidas para resolver la consulta):

1. Identificar las filas que cumplen la condición *Customer_City = "Jaramillo Quemado"*.
2. Reconstruir la columna *Customer_Gender* de acuerdo a las filas seleccionadas en la anterior operación.
3. Identificar las filas de la columna creadas en el paso anterior que contienen el valor *W* (género femenino).

4. Reconstruir la columna *Customer_Name* a partir de la lista de posiciones obtenida en el paso anterior.

Después del paso 4 ya tendríamos el resultado esperado. En el caso de que la consulta requiriera más columnas, se reconstruirían las columnas afectadas después del paso 3 y se materializarían posteriormente.

Para el ejemplo propuesto asumiremos que las listas de posiciones se generan mediante una lista que contiene las posiciones (es decir, los *rowid* o identificadores de filas) de los valores que satisfacen la condición.

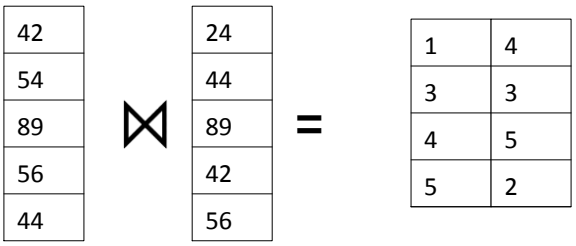
En el ejemplo podemos ver que la primera operación identificaría las posiciones de los valores de la columna *Customer_City* que contienen el valor Jaramillo Quemado. Como resultado, se generará una lista de 4 posiciones, con los valores 1.457.111, 1.457.112, 1.457.113 y 1.457.114 que identifican los valores de interés. Obviamente, la ejecución de esta operación sería más o menos eficiente en función de si la columna está ordenada, de si sus valores están comprimidos y de otros factores que veremos en el siguiente apartado.

Posteriormente se reconstruirá la columna *Customer_Gender* (afectada en el siguiente predicado de la consulta) con los valores de la lista de posiciones creada. Como resultado se generará una nueva columna con los valores del género de las personas que residen en Jaramillo Quemado. Cómo sólo deseamos obtener los nombres de las mujeres, realizaremos una selección de los valores de la nueva columna, seleccionando los que tienen un valor igual a *W*. El resultado de esta selección generará una nueva lista de posiciones. La nueva lista contendrá las posiciones 1.457.112 y 1.457.114, que son las posiciones que corresponden a las mujeres que residen en Jaramillo Quemado. Esas posiciones indican las filas que queremos obtener. Por lo tanto, la última operación será la de reconstruir la columna *Customer_Name* para los valores seleccionados. Una vez hecho esto tendríamos el resultado final de la consulta.

Como se puede observar, el hecho de realizar una materialización tardía, permite optimizar la consulta en este caso de ejemplo. El motivo es que en la primera operación se pueden descartar más del 99% de los valores de la tabla. En el caso de haber utilizado materialización temprana, dicho descarte no se hubiera podido hacer hasta después de materializar los 46 millones de filas con los valores de las tres columnas de interés para la consulta. Por lo tanto, se habría generado un gran número de filas que son irrelevantes para la resolución de la consulta.

Combinaciones en materialización tardía

- Las operaciones de combinación pueden ser problemáticas:
  - Las listas de posiciones resultantes de una combinación pueden no estar ordenadas.
  - Extraer datos requiere saltar de forma aleatoria a través de las posiciones de la columna:
    - Se pierde localidad de datos
    - Las lecturas aleatorias son más lentas que las secuenciales.



EIMT.UOC.EDU

Cuando se usa materialización temprana, los registros (o filas) se crean antes de realizar las operaciones de combinación, y por ello se pueden utilizar los operadores de combinación diseñados y optimizados para los almacenes de filas.

Pero, ¿qué sucede cuándo se desea utilizar materialización tardía? La manera más sencilla de realizar una combinación en un almacén de columnas es generar un par de listas de posiciones, una lista por cada columna a combinar. Las listas de posiciones indicarán las posiciones (los *rowid*) de los valores equivalentes en ambas columnas, de forma parecida a lo que se hace en una operación de *hash join*. En la transparencia podemos ver un ejemplo, en el que hacemos la combinación de dos columnas (dichas columnas y sus valores, en el ejemplo, se muestran a la izquierda). El resultado de la combinación serían 2 listas de posiciones (en la parte derecha del ejemplo) de 4 filas cada una, una para la posición del valor 42, otra para la posición del valor 89, otra para la posición del valor 56 y otra para la posición del valor 44.

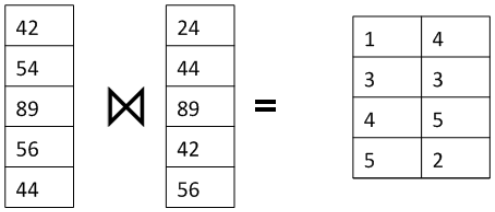
Por lo tanto, las listas de posiciones indicarán, para cada valor coincidente (o sea, para los valores que forman parte del resultado de la operación de combinación), su posición en ambas columnas. En consecuencia, la primera posición de las listas contendrá un 1 para la primera columna y un 4 para la segunda, ya que el valor 42 se encuentra en la primera posición en la primera columna y en la cuarta posición en la segunda. La segunda posición de las listas contendrá un 3 para la primera columna y un 3 para la segunda, y así sucesivamente. Para la mayoría de los algoritmos de combinación que podamos utilizar, se generaría la primera lista de posiciones ordenada (la de la izquierda) mientras que la segunda

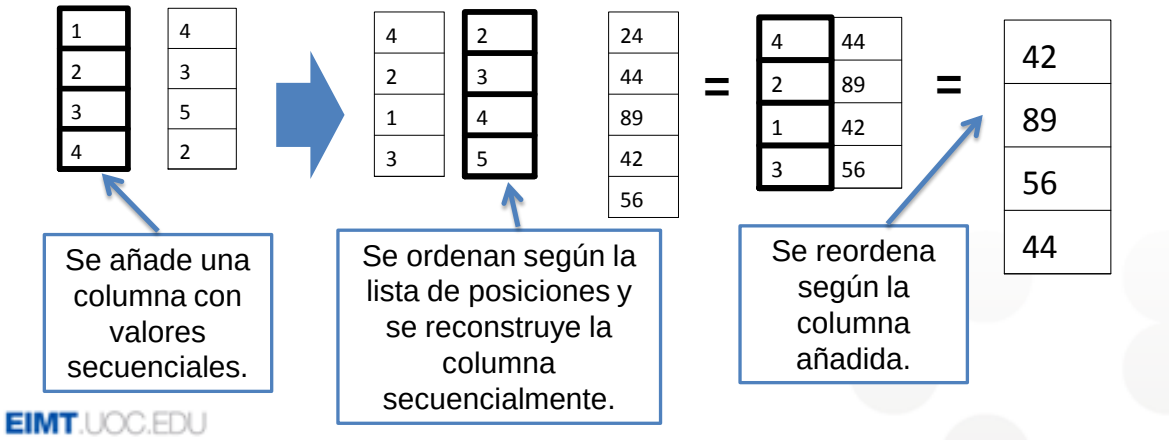
lista de posiciones no lo estaría (la de la derecha). Eso es así porque sobre la columna de la izquierda se itera de forma secuencial, mientras que para la columna de la derecha se requiere buscar los valores de la columna que satisfacen el valor de la columna de la izquierda.

Tener la lista de posiciones de la segunda columna desordenada es problemático, dado que implica realizar un conjunto de accesos directos por posición para reconstruir la porción de la columna que nos interesa. Los accesos directos por posición, en otras áreas de la informática como sería la de sistemas operativos, se denominan accesos o lecturas aleatorias, tal y como se indica en la transparencia, y pueden tener un coste prohibitivo dependiendo del número de elementos incluidos en la lista de posiciones. Por lo tanto, es mejor acceder a esos elementos realizando un recorrido secuencial por posición. El problema es que no se podrá reconstruir la porción de la columna que nos interesa a partir de esa lista haciendo un único recorrido secuencial sobre el fichero de datos donde se almacena la columna en cuestión. En el caso peor, serán necesarios realizar tantos recorridos secuenciales sobre el fichero como valores deseemos recuperar (sobre nuestro ejemplo, en concreto, podría ser necesario tener que realizar hasta 4 recorridos secuenciales).

Jive Join

- Existen alternativas:
  - *Jive Join*, *Radix Join*, etc.
  - *Multi-column Blocks*





Hay algoritmos que abordan este problema y proporcionan soluciones satisfactorias. Un par de ellos son el *Jive Join* y el *Radix Join*. Otras alternativas que utilizan los SGBD son la materialización de las columnas relevantes antes de realizar la combinación o el uso de bloques multicolumna.

Vamos a ver de forma sintética un par de ejemplos de estas estrategias. Podéis encontrar más información en las referencias indicadas al final de la presentación.

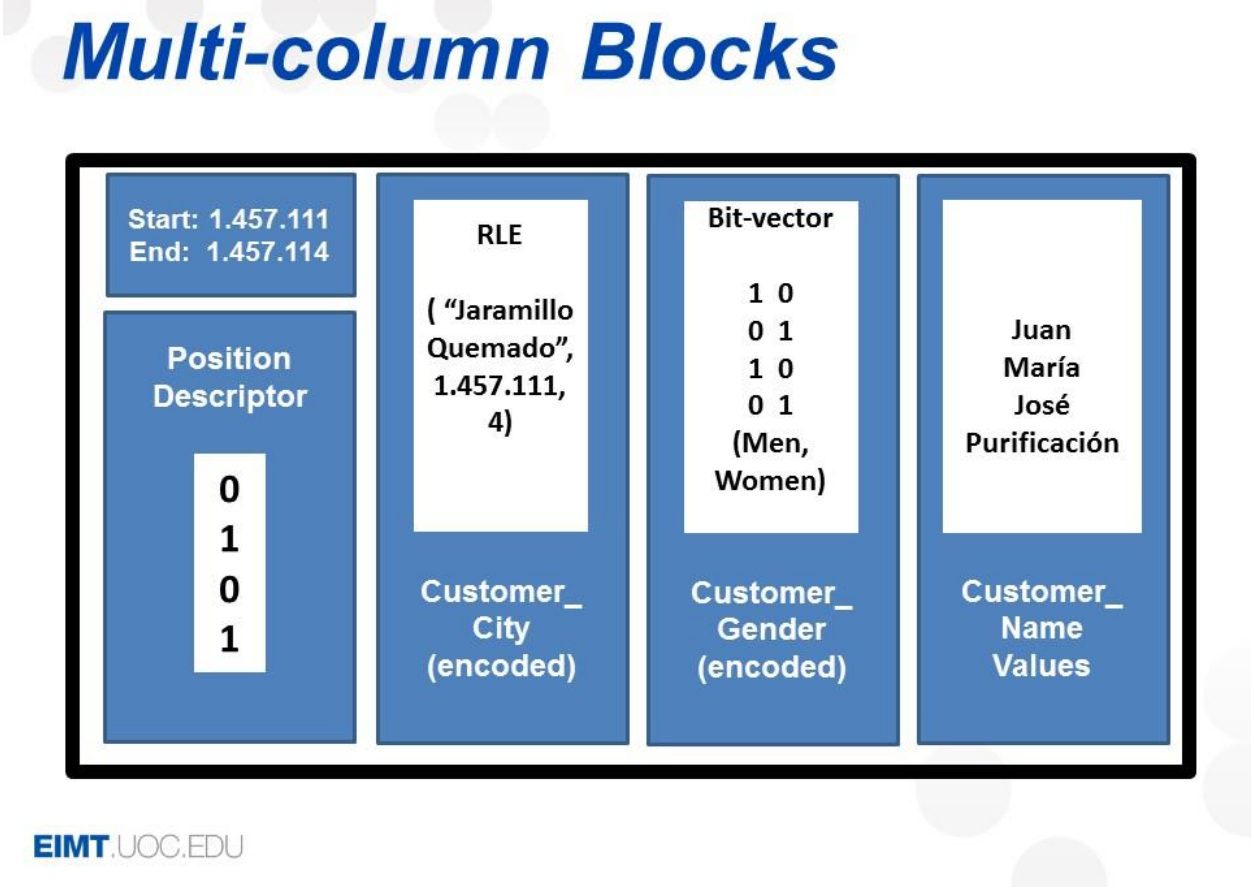
Empezaremos viendo un ejemplo del uso del algoritmo de *Jive Join* para obtener una reconstrucción de la segunda columna de forma ordenada.

Asumamos que se realiza la combinación que se muestra en la parte superior derecha de la transparencia, y que como resultado se obtienen las dos listas de posiciones mostradas. La segunda lista de posiciones contiene las posiciones de los valores de interés de forma desordenada.

Supongamos que aplicamos el algoritmo de *Jive Join* para materializar la columna a partir de su lista de posiciones. Para ello será necesario seguir una serie de pasos. El primer paso consiste en añadir una nueva columna a la lista de posiciones, que contenga la secuencia *1, 2, … N*, donde *N* es el número total de posiciones de la lista de posiciones, tal y como se muestra en la figura de más a la izquierda de la parte inferior de la transparencia. En el caso de que queramos obtener los valores de la columna (es decir, si queremos reconstruir la columna) se podría (como segundo paso) ordenar la lista de posiciones de acuerdo a las posiciones de los elementos relevantes (véase la parte central de la figura de la parte

inferior de la transparencia). Una vez hecho esto, tendríamos las posiciones de interés de los valores de la columna a reconstruir de forma secuencial y podríamos reconstruir la columna fácilmente haciendo una sola lectura secuencial del fichero donde están los datos de la columna, tal y como se muestra en la transparencia.

Notad que al ordenar por posición, la secuencia que hemos añadido en el paso anterior ha quedado desordenada. Si más adelante debemos reconstruir la ordenación original, por ejemplo, para realizar una materialización de columnas de distintas tablas, únicamente tendríamos que reordenar la columna resultante en función de la secuencia introducida.



Otra opción para minimizar las problemáticas asociadas a la materialización tardía es continuar trabajando con columnas, pero almacenar los resultados intermedios de las operaciones incluidas en las consultas de forma agrupada en función de las columnas y las filas de interés. De esta forma se almacenan juntos fragmentos de columnas que potencialmente se van a consultar juntos más adelante. La filosofía es mantener los datos agrupados por columnas y comprimidos pero en una estructura que asemeje (o simule) que se ha realizado una materialización temprana, para así minimizar los problemas de posteriores accesos a las columnas. Estas estructuras se denominan bloques multi-columna (*multi-column blocks* en inglés).

En la transparencia podemos ver el bloque multi-columna relevante para la consulta del ejemplo anterior (obtener el nombre de las clientas de Jaramillo Quemado). Los bloques multi-columna contienen:

1. Información sobre el rango de valores contenidos en el bloque, por ejemplo en la transparencia se indica que el bloque contiene las columnas que van de la fila 1.457.111 a la fila 1.457.114, que corresponden con las filas de los clientes que residen en Jaramillo Quemado. El bajo número de valores responde al ejemplo concreto que estamos utilizando, tal y como ya hemos discutido con anterioridad. En situaciones normales, el número de valores contenidos, potencialmente, será más elevado.
2. Un *array* de "mini-columnas", que contiene los fragmentos de distintas columnas que corresponden al rango indicado en el punto anterior. Por lo tanto, en el ejemplo tendríamos

fragmentos de las columnas implicadas en la consulta (*Customer_City*, *Customer_Gender* y *Customer_Name*). Los fragmentos de las columnas se pueden guardar comprimidos. En la figura de ejemplo se guarda un fragmento de la columna *Customer_City* comprimida mediante *run length encoding*, un fragmento de la columna *Customer_Gender* comprimida mediante *bit-vector encoding* y un fragmento de la columna *Customer_Name* sin comprimir.

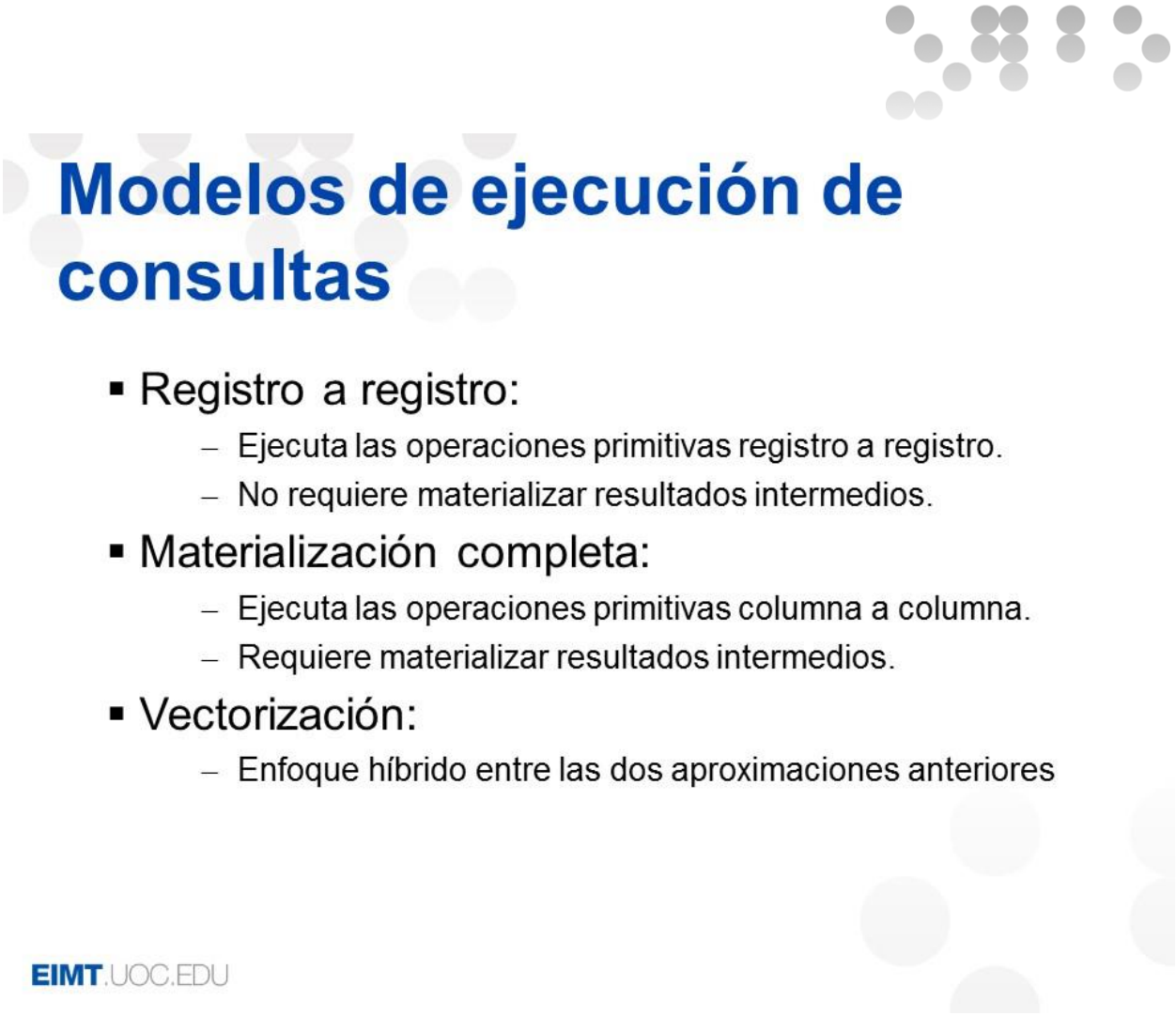
3. Un descriptor de posiciones que indica las posiciones del bloque que son válidas. En el ejemplo, una vez aplicado el filtro *(Customer_Gender="W")* el descriptor de posiciones indicaría que las posiciones válidas son la segunda y la cuarta.

Procesamiento de consultas

- Operar con datos comprimidos
- Materialización tardía
- **Modelos de ejecución de consultas**
- *Database Cracking*

EIMT.UOC.EDU

A continuación, veremos los distintos modelos de ejecución de consultas y cómo algunos de ellos pueden utilizar la ejecución en bloque para optimizar las consultas en almacenes de columnas.



La vectorización es una técnica compleja. Conocer su funcionamiento de forma detallada queda fuera del alcance de este material didáctico. A pesar de ello, a continuación vamos a presentar una pequeña introducción (desde un punto de vista conceptual y a grandes rasgos) de la vectorización y una enumeración de las ventajas que puede ofrecer.

La ejecución de una consulta en un SGBD se puede hacer de distintas formas: 1) registro a registro (o si preferís, fila a fila o tupla a tupla), 2) mediante materialización completa y 3) vía vectorización.

La ejecución fila a fila, también conocida como *Volcano-style* o *tuple-at-time pipelining*, es la más clásica desde un punto de vista histórico, y también la más elegante desde un punto de vista de ingeniería del software. En este modelo de ejecución, las distintas operaciones involucradas en el plan de una consulta se ejecutan de forma sincronizada y siguiendo la estructura (normalmente de árbol) definida en el plan de consulta. En este caso, cada operador (u operación) relacional implicado en el plan de consulta produce una nueva tupla (o fila o registro) cada vez, a partir de los datos recibidos de la ejecución de sus operaciones hijas en el árbol, y genera una nueva tupla que envía a su operación padre. Por lo tanto, tenemos un proceso en forma de árbol (cada nodo del árbol representa una operación relacional que es necesario ejecutar para resolver la consulta) que gestiona los registros (o filas o tuplas) uno a uno, y en donde el control del flujo de ejecución (la sincronización) se realiza desde los niveles superiores del árbol (en otras palabras, desde las operaciones padre). Al procesar registros uno a uno, las operaciones del

SGBD trabajan con valores individuales, y también se minimiza la necesidad de almacenar resultados intermedios.

El segundo modelo de ejecución de consultas, la materialización completa, ejecuta las operaciones presentes en el plan de ejecución de la consulta de forma secuencial. En concreto, cada operación presente en el plan de la consulta se ejecuta consumiendo totalmente los datos de entrada (el conjunto de filas a procesar, en definitiva), generando una salida (o resultado intermedio) que, a su vez, se almacena (o sea, se materializa). La ejecución de las operaciones se encadena secuencialmente una tras otra para generar la solución final (en otras palabras, para generar el conjunto de filas que constituyen el resultado de la consulta).

Este modelo de ejecución, por ejemplo, es el utilizado por MonetDB. Se trata de un modelo que consigue ejecuciones más eficientes en términos de CPU. No obstante, puede requerir un uso de recursos excesivo al generar resultados intermedios que va a ser necesario almacenar, especialmente en consultas que contengan operaciones de selección que incluyan predicados poco restrictivos. Por ejemplo, suponed una consulta que calcule la media de todas las ventas con un valor de más de 1€. En el caso de que la gran mayoría de nuestras ventas (imaginemos que alrededor del 99%) fueran superiores a 1€, la operación que filtra las filas con ventas superiores a 1€ generaría una materialización (o resultado intermedio) que contendría alrededor del 99% de filas de la tabla original. Eso podría ser demasiado costoso en entornos que gestionan una gran cantidad de datos. Al procesar toda la entrada a la vez, las operaciones del SGBD trabajan con conjuntos de valores.

El modelo de ejecución vectorial (o vectorización) es una solución de compromiso entre las dos aproximaciones anteriores, donde la ejecución se continúa realizando de forma iterativa, como en la ejecución fila a fila, pero los operadores se ejecutan sobre un conjunto de valores en vez de sobre un valor individual. Esto permite ejecutar las operaciones en bloque y reducir el número de iteraciones necesarias. Por lo tanto, cada vez que se itere, se procesará un vector de *N* tuplas que contiene los datos sobre los que se operará. Respecto a las operaciones, en este caso, procesarían vectores de longitud *N.* El tamaño del vector normalmente se define de manera que quepa confortablemente en la caché. VectorWise, por ejemplo, utiliza vectorización usando vectores de 1.000 posiciones por defecto.

Actualmente, las técnicas de materialización completa y vectorización pueden representar importantes mejoras en la eficiencia, debido a la capacidad de ejecutar operaciones con múltiples datos (SIMD) de los procesadores actuales.

Los procesadores actuales soportan ejecuciones de tipo SISD (*Single Instruction Single Data*) y SIMD (*Single Instruction Multiple Data*). Básicamente estas siglas indican cómo se ejecutaran las operaciones a nivel de procesador. Si utilizamos SISD, las operaciones del procesador se ejecutarán sobre un solo dato. Las técnicas SIMD permiten ejecutar una operación sobre un conjunto de datos. Por ejemplo, si queremos incrementar en una unidad todos los valores de una columna y utilizamos técnicas SISD, deberemos ejecutar la operación de incremento sobre cada valor de la columna. No obstante, si utilizamos técnicas SIMD, podremos ejecutar la operación sobre toda la columna (o parte de la misma) de forma atómica. Utilizar operaciones SIMD para ejecutar operaciones de la BD permite mejorar el rendimiento de las consultas.

La arquitectura SIMD puede aplicarse fácilmente en los almacenes de columnas para ejecutar una sola operación sobre una columna (o un subconjunto de ella) en bloque. Esta técnica recibe el nombre de

ejecución en bloque (en inglés, *block iteration*). Como los valores de las columnas están almacenados de forma consecutiva y las operaciones de la BD se ejecutan sobre los valores de las columnas, es fácil mover los datos de una columna juntos a la CPU y aplicar operaciones de tipo SIMD que ejecuten una misma operación para todos los valores de la columna.

Ejemplo de materialización completa

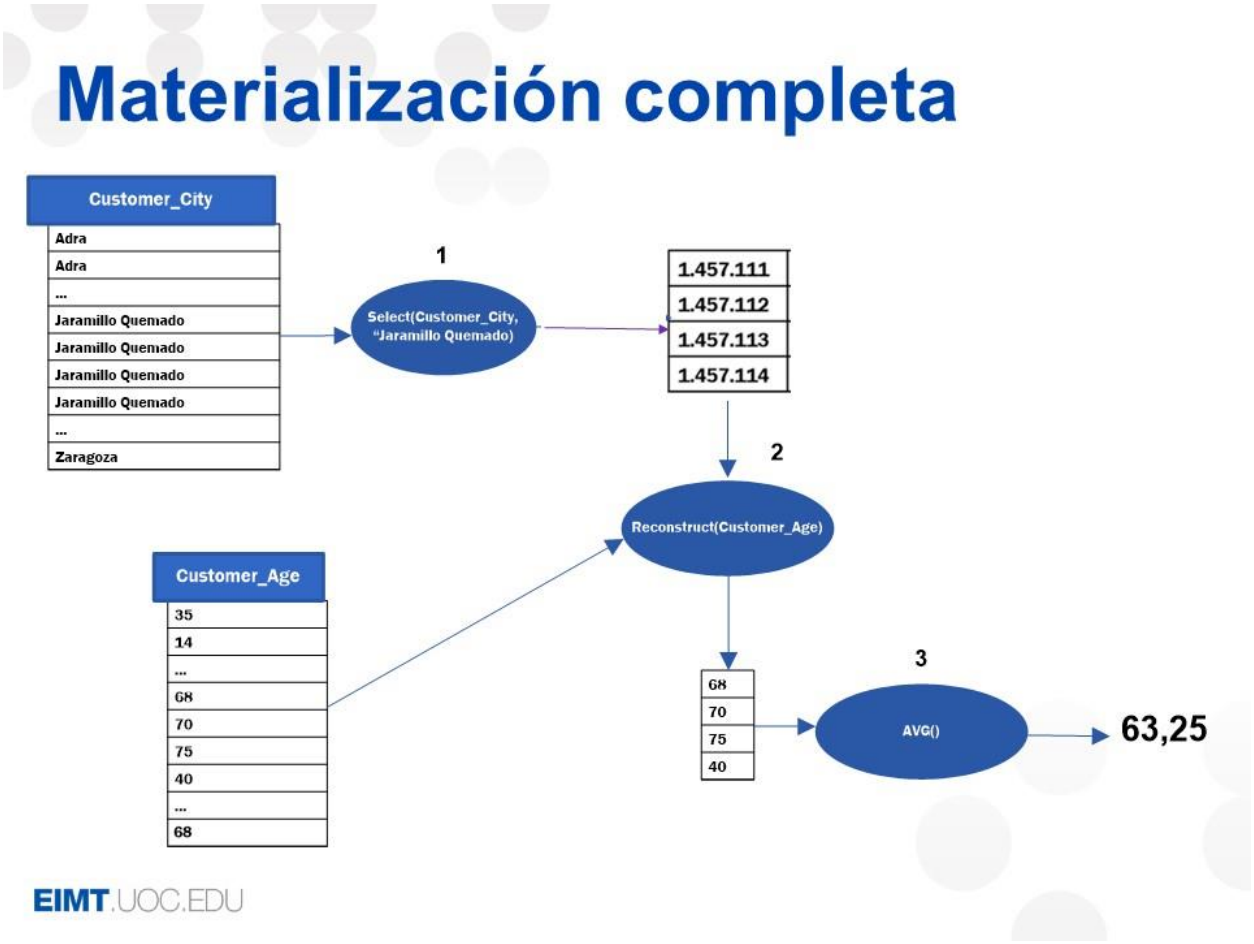
Customer
Customer_Id : longInt Customer_Name: String (35) Customer_Surname: String (35) **Customer_Age: Integer** Customer_Gender: Byte DEFAULT "W" Customer_City: String (35) Customer_Country: String (35)

```
SELECT AVG(Customer_Age)
  FROM Customer
 WHERE Customer_City = "Jaramillo Quemado"
```

EIMT.UOC.EDU

A continuación vamos a ver un ejemplo de cómo funciona la materialización completa. No vamos a entrar en detalle a hablar de vectorización pero, conceptualmente, podemos entenderla como un caso particular de materialización completa que requiere menos recursos para la materialización de resultados intermedios y que, a la vez, puede sacar provecho del uso de técnicas SIMD.

En el siguiente ejemplo vamos a ver cómo se ejecutaría, utilizando materialización completa, la consulta que se indica en la transparencia. En este caso particular, la consulta pide la media de edad de los habitantes de la población Jaramillo Quemado. Para ello hemos añadido un atributo (o columna) edad (*Customer_Age*) a la tabla *Customer*.



Supondremos que la consulta de ejemplo se ejecuta utilizando estrategias de materialización tardía. En dicho caso, un posible plan de ejecución para la consulta podría ser el siguiente:

1. Selección de los clientes que residen en Jaramillo Quemado.
2. Reconstrucción de la columna *Customer_Age* de los clientes que residen en Jaramillo Quemado (a partir de la lista de posiciones obtenida en la operación anterior).
3. Calcular la edad media (mediante una operación `AVG`) de los clientes seleccionados

En la transparencia podemos ver cómo se ejecutarían las tres operaciones. Para ejecutar la primera operación asumiremos que la BD ofrecería una operación que, dada una columna y un valor concreto, obtiene las posiciones de una columna que contienen dicho valor. Internamente la operación del SGBD podría implementarse mediante operaciones SIMD en el procesador, mediante una operación de igualdad en bloque por ejemplo. Ello podría acelerar enormemente el tiempo de respuesta. Como resultado de dicha operación obtendríamos la lista de las 4 posiciones que corresponde a los clientes que residen en Jaramillo Quemado. Esa lista de posiciones, junto con la columna *Customer_Age*, serán las entradas de la siguiente operación, que dada una lista de posiciones y una columna reconstruye la columna con los valores de las posiciones indicadas. Dicha operación devolvería una columna con 4 celdas que contendría las edades de los clientes de interés: 68, 70, 75 y 40. Finalmente, la operación de media (`AVG`) calcularía la media de los valores de la columna, devolviendo el valor final: 63.25 años.

Características

- La materialización completa es más habitual en los almacenes de columnas.
- La vectorización funciona tanto en:
  - Almacenes de filas
  - Almacenes de columnas
- Ventajas:
  - Reducción de la carga de CPU
  - Aprovecha las operaciones SIMD.
  - Aprovecha la localidad de los datos.
  - Mejor optimización y ejecución adaptativa

EIMT.UOC.EDU

La vectorización puede soportar tanto representaciones de datos por columnas como por filas. Según la bibliografía, el rendimiento de las operaciones que requieren recorridos secuenciales (como sería el caso de proyección y selección, por ejemplo) es mejor cuando se utiliza sobre representaciones columnares (ya que aprovechan las características SIMD de los procesadores actuales), mientras que las operaciones que se basan en un acceso aleatorio a los datos (*hash-join* o agregación) tienen un rendimiento mejor al usar representaciones por filas.

En los almacenes de columnas es conveniente retrasar la materialización lo más tarde posible al utilizar vectorización para permitir ejecuciones en bloque. La vectorización soporta tanto datos comprimidos como datos no comprimidos, pero conviene que los datos tengan un tamaño fijo para poder facilitar su ejecución vectorial (en otras palabras, es conveniente que los tipos de datos asociados a las columnas sean de longitud fija y, en algunos casos , que el número de celdas por cada columna sea también fijo ya que si no puede dificultar la materialización).

Algunas de las ventajas de la vectorización son:

- Se reduce la carga de CPU: el número de llamadas realizadas sobre el intérprete de una consulta se reduce en función del tamaño del vector a utilizar. Por ejemplo si tenemos vectores de 1.000 valores, podremos reducir en un factor de 1.000 el número de operaciones ejecutadas.
- Oportunidades de optimización: algunas operaciones de la BD podrán traducirse directamente en operaciones SIMD del procesador o combinaciones de las mismas.

- Ejecución adaptativa: la implementación vectorial de las operaciones permite tener información más detallada del rendimiento de la consulta y del coste de CPU para cada operación. Esta información puede ser aprovechada por el SGBD para cambiar el plan de la consulta de forma dinámica, en función del coste esperado de una operación. Por ejemplo, supongamos que debemos multiplicar ciertos valores de una columna por 5. Supongamos que el planificador, en tiempo de ejecución (después de aplicar una operación de selección), comprueba que el número de valores de interés es muy bajo (10 valores por ejemplo) mientras que el número de valores de la columna es muy elevado (hay muchos valores irrelevantes). En este caso, el planificador puede optar por ejecutar la operación de forma individual para cada valor de interés (lo que parecería conveniente en este caso) en vez de ejecutar la multiplicación de la columna en bloque (que probablemente tendría un coste de ejecución mayor).

Procesamiento de consultas

- Operar con datos comprimidos
- Materialización tardía
- Modelos de ejecución de consultas
- ***Database Cracking***

EIMT.UOC.EDU

Una vez llegados a este punto vamos a hablar de una de las últimas técnicas añadidas a los almacenes de columnas. El *database cracking* o indexación adaptativa.

Indexación

- Usar ordenación o índices es más eficiente:
  - La ordenación, en términos generales, es más eficiente.
  - *Los zonemaps* permiten optimizar consultas mediante el uso de metadatos simples a nivel de página.
- Usar ordenación o índices requiere:
  - Conocimiento sobre el dominio y el consumo de datos
  - Tiempo para generar las estructuras necesarias
    - Son complejos de mantener y poco adaptables a cambios
- *Database Cracking:*
  - Las consultas recibidas se utilizan como base para crear los índices.
  - Se adapta a cambios en los datos o en el consumo.

EIMT.UOC.EDU

El recorrido de datos en los almacenes de columnas es, por defecto, más eficiente que los recorridos en los almacenes de filas. No obstante, hay maneras de mejorar aún más su eficiencia explotando la ordenación/indexación de forma adecuada.

Existen trabajos que demuestran que la utilización de índices en los almacenes de columnas consiguen mejoras en la eficiencia en los recorridos de uno o dos órdenes de magnitud. Sin embargo, otros estudios demuestran que, en términos generales, es más eficiente el acceso sobre las columnas cuando éstas están físicamente ordenadas, haciendo innecesario el uso de índices como los utilizados en los almacenes de filas.

Por lo tanto, una opción para mejorar el rendimiento, es almacenar las columnas de forma ordenada, en función del valor de una columna o conjunto de columnas (que reciben el nombre de clave de ordenación). Adicionalmente, una misma columna o grupo de columnas (cada grupo recibe el nombre de proyección) se pueden almacenar diversas veces, de acuerdo a diferentes criterios de ordenación (o sea, en función de distintas claves de ordenación). Si recordáis estas cuestiones fueron tratadas en profundidad en la presentación dedicada a las características de los almacenes de columnas.

Otra forma de mejorar el rendimiento de los almacenes de columnas (que puede ser usada conjuntamente con la ordenación física de los datos) es a través de *zonemaps*. Estas estructuras

auxiliares (que no dejan de ser índices no densos –en inglés, *sparse indexes*–) almacenan metadatos básicos a nivel de página, como por ejemplo (y entre otros), el valor mínimo y máximo almacenado en la página. Dichos metadatos son útiles para poder descartar las páginas que sean irrelevantes para una consulta dada, sin tener que acceder a su contenido. Estas estructuras también fueron discutidas en la presentación dedicada a las características de los almacenes de columnas.

Una desventaja de las estrategias basadas en la ordenación de los datos es que requiere un conocimiento elevado del dominio, para que sean efectivas. Por ejemplo, es necesario conocer la carga de ejecución que deberá soportar la BD, los datos que contendrá y cómo se distribuirán, cuál será la frecuencia de actualización y consulta de los datos, qué tipos de consultas se esperan, etc. Además, una vez se tiene esta información, hay que considerar los costes de creación y mantenimiento de las proyecciones, y del resto de estructuras auxiliares que se hayan podido crear (los *zonemaps* o índices no densos).

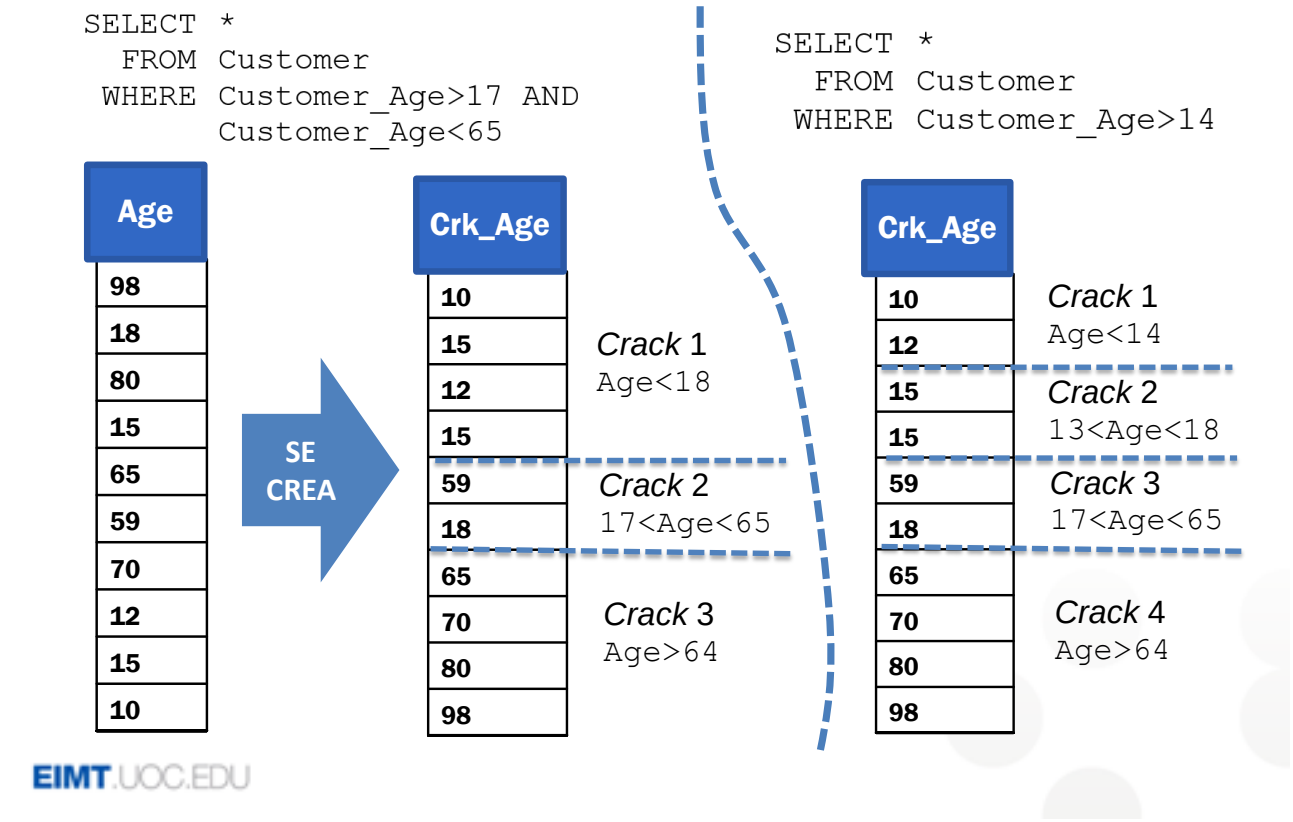
Es importante destacar que todo lo que acabamos explicar no está alineado con ciertas necesidades actuales, donde el reto es proveer de SGBD capaces de procesar cantidades masivas de datos de forma rápida (aunque no sea óptima), pero suficientemente flexibles para adaptarse a cambios en los datos o en sus hábitos de consumo. Para dar solución a estas nuevas necesidades surgen los índices adaptativos, normalmente llamados *database cracking*.

El *database cracking* es una aproximación donde la definición y la evolución de los índices está dirigido por el tipo de consultas que se realizan sobre los datos. Esta técnica utiliza normalmente las consultas para inferir las necesidades de consultas de los datos. Cuando se utiliza *database cracking*, cada consulta recibida se interpreta como una pista del tipo de consultas que se recibirán en el futuro. Para adaptarse a futuras consultas, agrupa físicamente los datos para ofrecer más eficiencia a consultas parecidas que se puedan efectuar en el futuro. Por lo tanto, el índice se crea dinámicamente, evolucionando después de cada consulta (y nunca después de las operaciones de inserción y de actualización) y adaptando la manera en que se agrupan los datos físicamente en el dispositivo de almacenamiento externo.

El *database cracking* es una técnica reciente y muy popular hoy en día (al menos en entornos de investigación), pero ¿es más conveniente que la indexación/ordenación? Bien, depende. Si tenemos un entorno donde sabemos de antemano qué datos son interesantes para los usuarios, qué datos se utilizarán en las consultas y podemos asumir los costes de indexar/ordenar los datos antes de recibir las consultas o entre una actualización y una consulta, la ordenación/indexación es una estrategia mejor. El *database cracking* es muy útil en entornos donde no hay conocimiento sobre qué datos son interesantes y/o no no hay tiempo suficiente para mantener el orden físico de los datos ante cambios en los datos.

Algunos ejemplos de almacenes de columnas que se basan en la ordenación de los datos (y en el uso de *zonemaps*) son Vertica (y C-Store, el prototipo en el que se basa), y Amazon Redshift. Por su parte, MonetDB es, posiblemente, el mejor ejemplo de almacén de columnas que usa *database cracking*.

Database Cracking



En esta transparencia vamos a ver cómo funciona el *cracking* en un almacén de columnas. Para verlo, utilizaremos la BD de ejemplo utilizada hasta ahora. En particular, vamos a utilizar *cracking* para indexar la columna *Age*.

La primera vez que se usa una consulta *q* por rango sobre un atributo (o columna) *A*, se crea una copia de la columna *A*. Esta copia se llamará la columna de *cracking de A* y cumplirá la condición de que los valores de *A* que satisfacen cada condición de *q* se almacenarán de forma consecutiva (es decir, se almacenarán de forma conjunta). En consultas subsiguientes sobre el mismo atributo, su columna de *cracking* se irá actualizando de acuerdo a la nueva condición. Por lo tanto, la columna se irá particionando (o subdividiendo) en más y más *cracks* (o zonas) a medida que lleguen más consultas.

Para tener controlados los distintos *cracks*, los valores qué contienen y cómo se distribuyen se utilizan índices. Estos índices pueden ser de distintos tipos, pero la idea es que permitan acceder a las diferentes zonas o *cracks* que se hayan creado.

Normalmente, el *cracking* se realiza en memoria principal, permitiendo evitar la copia de datos para cada *crack*. En condiciones normales los *cracks* contienen punteros (los *rowid*) a los valores asociados en la columna original (que estará almacenada en el dispositivo de almacenamiento externo), permitiendo una mayor eficiencia en el proceso.

En el ejemplo de la transparencia, supongamos que la primera consulta realiza un filtraje por los clientes mayores de 17 años y menores de 65. En este caso, se construiría una columna de *cracking* que estaría dividida en tres *cracks* o zonas. Una para los valores inferiores a 18 (primera condición de la consulta), otra con los valores superiores a 64 (segunda condición de la consulta) y la otra con el resto de valores, justamente los que son de interés para la consulta (mayores de 17 años y menores de 65 años, o sea clientes entre 18 y 64 años). A partir de ahora, cualquier consulta que busque valores con rangos similares a los utilizados podría acceder directamente a las zonas relevantes. Fijaos que los *cracks* no están totalmente ordenados ya que el algoritmo de *cracker* mueve sólo los valores necesarios para cumplir la condición indicada (los valores que satisfacen cada condición deben estar ubicados consecutivamente en un mismo grupo, pero no necesariamente ordenados). Obviamente, si se ejecutaran muchas consultas, podríamos acabar teniendo la columna de *cracking* totalmente ordenada.

Supongamos ahora que el SGBD recibe una nueva consulta que solicita los clientes de más de 14 años. En este caso el SGBD, de acuerdo a la nueva consulta (y las condiciones que ésta impone), partiría el primer *crack* de la columna de *cracking* en dos: uno para almacenar los clientes menores a 14 años y otro donde se almacenan los clientes con edades entre 14 y 17 años. El resto de *cracks* no se verán modificados. Fijaos que en este momento, en la columna de *cracking*, en los *cracks* 1 y 2 los valores ya están completamente ordenados.

Si en este momento se realizara una nueva consulta preguntando por los clientes de entre 18 y 64 años (condición 17<*Age*<65), el SGBD podría devolver la zona (o *crack*) 3 de la columna de *cracking*, y no debería reestructurar la columna, ya que todos los valores devueltos en la consulta ya están almacenados de forma consecutiva en un mismo grupo.

Consideraciones sobre *Database Cracking*

- Se ejecuta en operaciones de selección.
  - Combinaciones y agregaciones
- Concurrencia
- Actualizaciones
  - ¿Cómo afectan a los índices y las columnas de *cracking*?
  - ¿Cuándo actualizar las columnas de *cracking*?
    - Cuando los datos son consultados
- ¿Cuándo parar?
  - Generar demasiados *cracks* puede tener un efecto negativo en el rendimiento.

EIMT.UOC.EDU

La técnica de *database cracking* es relativamente nueva y se usa básicamente para optimizar las consultas que implican operaciones de selección (o sea, consultas por rango). No obstante, la tecnología muestra también potencial para optimizar las operaciones de agregación y combinación en los almacenes de columnas. Esto permitiría optimizar aún más las operaciones más costosas de los almacenes de columnas.

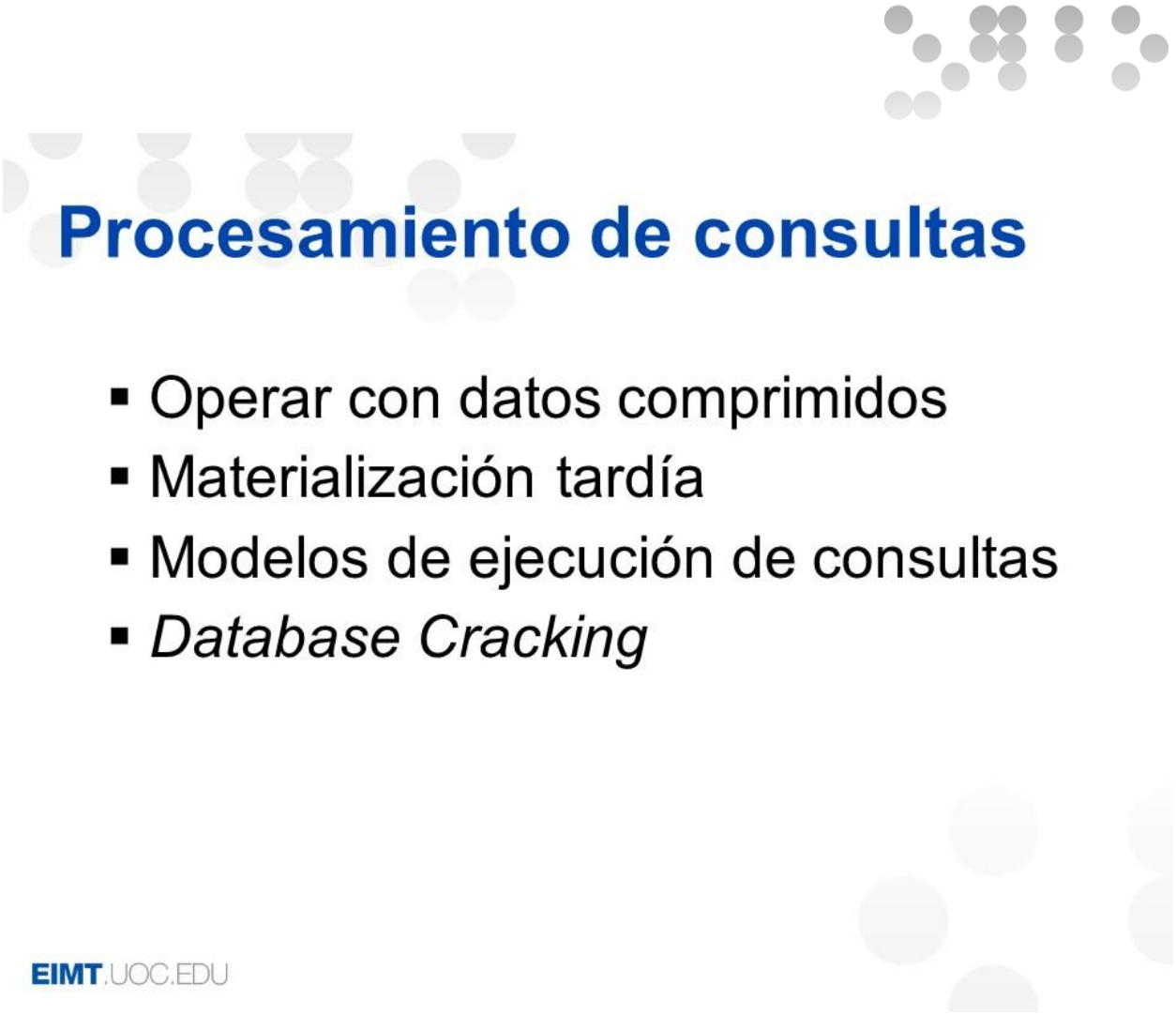
Otro aspecto a considerar es el acceso concurrente a los datos. Aquí nos podemos imaginar distintas situaciones donde se puede incurrir en problemas. Por ejemplo, una situación problemática sería la ejecución concurrente de dos consultas donde una de ellas está utilizando una zona (o *crack*) de la columna de *cracking* para obtener los valores relevantes (y por lo tanto, potencialmente, está reestructurando la zona) mientras otra consulta quiere obtener datos de la misma zona. Existen distintas alternativas, desde el típico bloqueo que no permite modificar una zona mientras otro usuario la esté consumiendo, hasta aproximaciones más optimistas que buscan paralelizar las consultas sobre partes disjuntas de los datos para permitir reestructurar la columna de *cracking* sin realizar bloqueos.

Otro de los aspectos a tener en cuenta es cómo afecta la actualización de datos sobre los índices y columnas de *cracking*. Aquí lo importante es saber cuándo actualizar las columnas de *cracking*. Una aproximación es mantener las modificaciones separadas hasta que el número de modificaciones supere un cierto umbral. Esto puede llevar la BD a un estado en el que los usuarios puedan tener problemas para obtener los datos actualizados. También se pueden reestructurar los índices y columnas de *cracking* no cuando se añadan nuevos datos (o se modifiquen los existentes), sino cuando se consulten esos datos. Otra opción sería añadir los nuevos datos al final de las columnas de *cracking* (al recibir

inserciones o modificaciones) y “olvidar” los índices de *cracking*. Esto obligaría a crear de nuevo los índices y los distintos *cracks* a partir de cero en función de las nuevas consultas, volviendo a empezar el proceso de indexación. Algunos experimentos muestran que usando esta aproximación los índices de *cracking* consiguen un buen rendimiento rápidamente después de pocos cientos de consultas.

Por otro lado, otro aspecto relevante es saber cuándo hay que parar de hacer *cracking* sobre una columna. Parece razonable pensar que si continuamos particionando una columna indefinidamente, llegaremos a un punto donde el número de *cracks* será muy elevado y la gestión de los índices y de la columna de *craking* será demasiado costoso. La opción más adoptada para tratar esta situación consiste en definir el número de elementos mínimos que debe tener cada *crack*. Así, si potencialmente un *crack* tiene muy pocos valores (menos que el mínimo que se haya establecido), no se creará.

Si alguien quiere entrar más a fondo en los temas comentados en esta transparencia, aconsejamos consultar la tesis de Stratos Idreos, se trata de la última referencia que encontraréis al final de este documento.



Y hasta aquí estos materiales dedicados al procesamiento de consultas en almacenes de columnas, donde hemos visto las principales técnicas que permiten optimizar la ejecución de las consultas. En concreto, hemos explicado cómo operar con datos comprimidos, cuándo usar materialización tardía, los modelos de ejecución que un SGBD (y en particular un almacén de columnas) puede utilizar para resolver las consultas que se formulan sobre la BD, y de las técnicas de indexación más recientes que se han propuesto para los almacenes de consultas (*database cracking*).

Esperamos que hayáis encontrado los materiales amenos, útiles y el tema interesante. Algunos de los conceptos tratados son altamente especializados y de una cierta complejidad. Es más, parcialmente, se encuentran bajo investigación. El objetivo principal es que hayáis comprendido las ideas generales que subyacen en cada una de las técnicas que hemos explicado.

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, 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), pp. 197-280.

D.J. Abadi, D.S. Myers, D.J. DeWitt, S. Madden (2007). Materialization Strategies in Column-Oriented DBMS. *Proceedings of the International Conference in Data Engineering*, pp. 466-475.

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*, pp. 967–980.

G. Graefe, L. Shapiro (1991). Data compression and database performance. In *ACM/IEEE-CS Symposium on Applied Computing*, pp. 22-27.

S. Harizopoulos, D.J. Abadi, P. Boncz (2009). *Column-oriented Database Systems*. VLDB Tutorial.

S. Idreos, M.L. Kersten, S. Manegold (2007). Database Cracking, *3rd Biennial Conference on Innovative Data Systems Research* (CIDSR).

S. Idreos, M.L. Kersten, S. Manegold (2007). Updating a cracked database (2007). *Proceedings of the ACM SIGMOD International Conference on Management of data*.

S. Idreos (2010). *Database cracking: Towards auto-tuning database kernels*. PhD Dissertation. CWI University of Amsterdam.

EIMT.UOC.EDU

¡Qué tengáis un buen día!