Magatzem de columnes

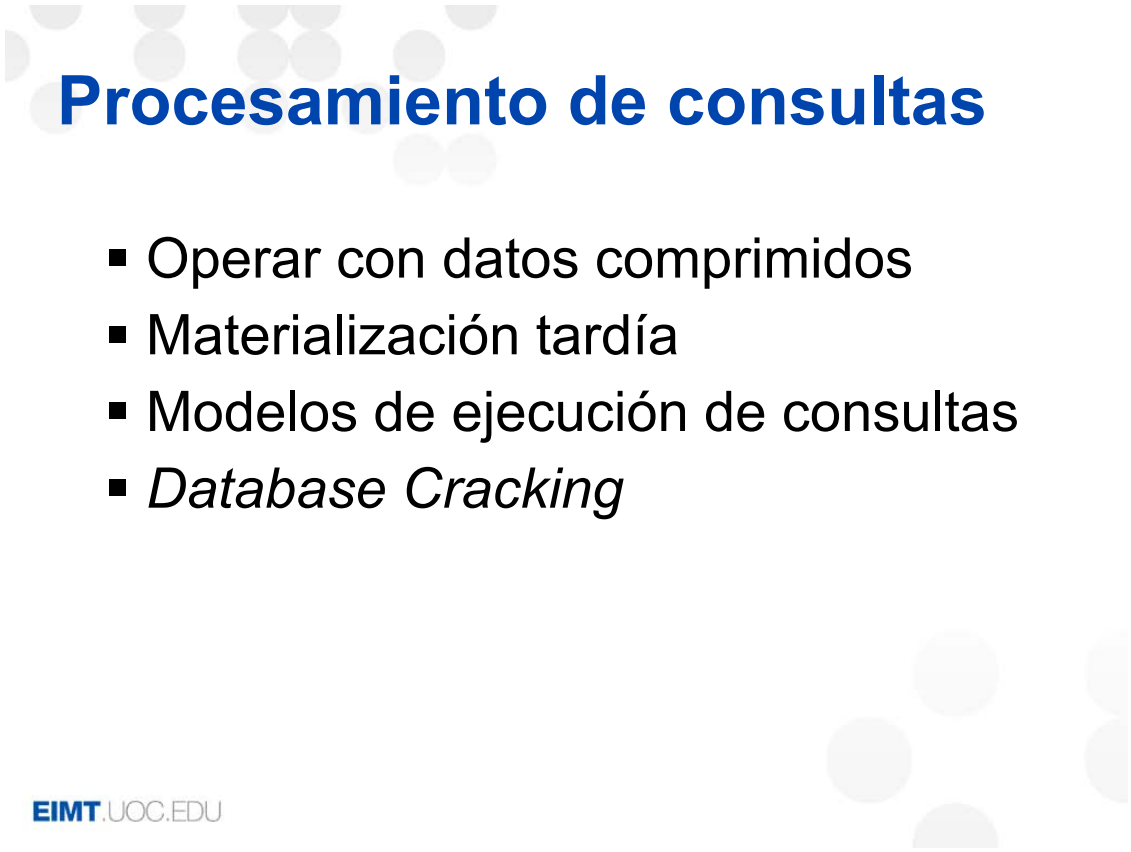
Processament de consultes

PID_00253058

Jordi Conesa i Caralt

M. Elena Rodríguez González

Universitat Oberta de Catalunya



Benvinguts a la quarta presentació en què explicarem com es gestionen les consultes en els magatzems de columnes.

Aquesta presentació està dividida en 4 parts:

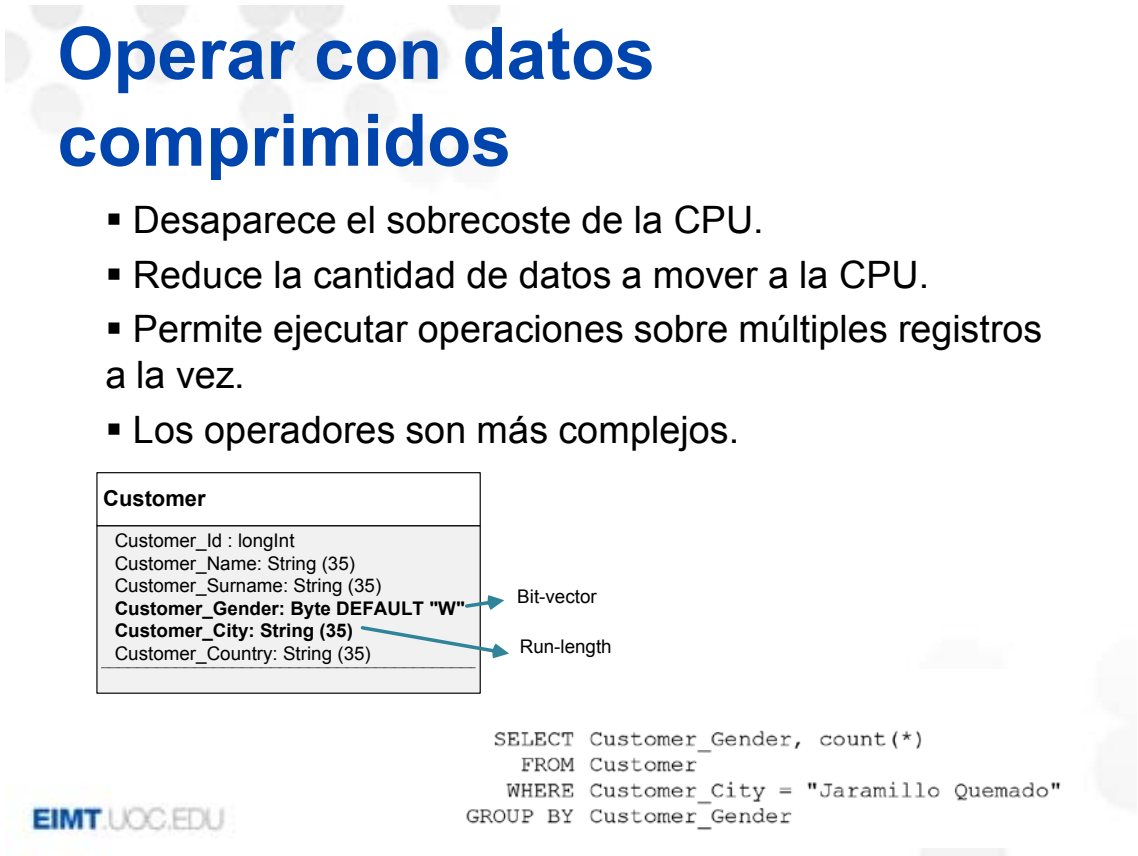
- Operar amb dades comprimides: en alguns casos els magatzems de columnes poden realitzar les operacions directament sobre les dades comprimides de la base de dades (BD). Veurem els avantatges que això té, així com alguns exemples en què s'opera directament sobre les dades comprimides.

- Materialització tardana: en aquest apartat veurem algunes estratègies utilitzades pels magatzems de columnes per a resoldre les consultes formulades sobre la BD. En particular, parlarem de la materialització primerenca, de la materialització tardana, dels avantatges d'una o altra opció en els magatzems de columnes i d'algunes tècniques d'optimització per quan s'utilitza la materialització tardana.

- Models d'execució de consultes: en aquesta secció presentarem els diferents models d'execució de consultes que un sistema gestor de bases de dades (SGBD) pot utilitzar per a resoldre les consultes i els contextualitzarem per al cas particular dels magatzems de columnes. Hi ha 3 models: els models que operen sobre els valors individuals (registre a registre), els que operen

directament sobre les columnes en bloc (materialització completa) i els que operen sobre fragments de columnes (vectorització).

- Finalment, parlarem del *database cracking* o índexs adaptatius, que són un mecanisme que ens permet indexar de forma dinàmica les dades de la BD en funció del tipus de consultes que es formulen.



Tal com ja hem comentat en altres presentacions, en alguns casos, els magatzems de columnes permeten que les operacions que necessita executar l'SGBD per a respondre les peticions formulades pels usuaris es puguin resoldre directament sobre les dades comprimides. Els avantatges d'operar directament sobre les dades comprimides són els següents:

1. Desapareix el sobrecost (o sobrecàrrega) de la CPU necessari per a descomprimir les dades abans d'operar sobre aquestes.

2. Es redueix la quantitat de dades que es desplacen (o es transfereixen) des de la memòria a la CPU per a operar amb aquestes.

3. Es pot executar l'operació amb menys cicles de CPU, ja que permet executar operacions sobre múltiples valors (registres) alhora.

No obstant això, s'ha de tenir en compte que processar directament sobre dades comprimides té un cost en la implementació dels operadors (o operacions) de la BD, que acabaran sent més complexos i menys escalables. Expliquem el perquè amb un parell d'exemples. Suposem que volem afegir un esquema de compressió a l'SGBD (un *Dictionary encoding*, per exemple). En aquest cas, els operadors unaris (per exemple, la projecció) hauran de ser capaços d'operar sobre columnes no comprimides i sobre columnes comprimides mitjançant el *Dictionary encoding*. Per tant, haurem de crear dues versions de cada operador unari, una versió de l'operador que treballi sobre

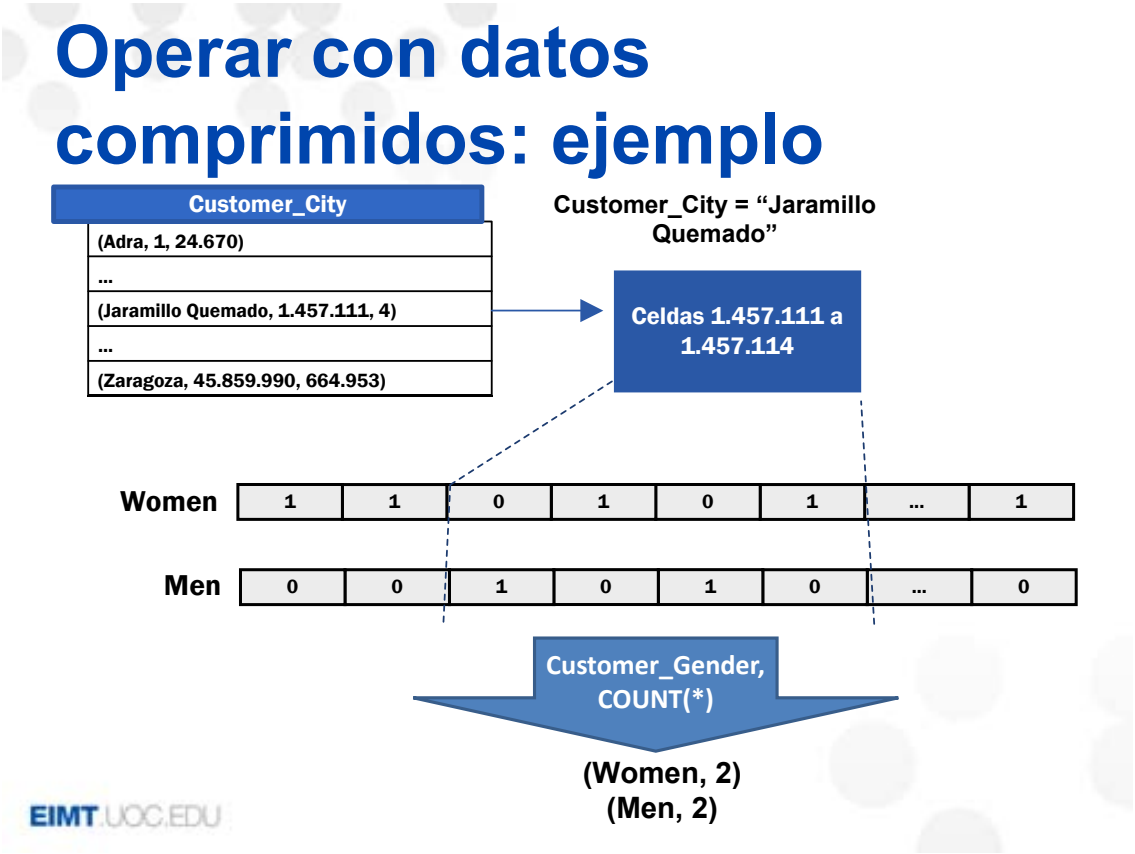
columnes no comprimides i una altra versió que operi sobre columnes comprimides mitjançant l'algorisme *Dictionary encoding*. Suposem ara que volem afegir un nou esquema de compressió (per exemple, un *Run-length encoding*). En aquest cas, els operadors unaris haurien de tenir una nova versió que permeti operar amb columnes comprimides mitjançant el *Run-length encoding*. Com a norma general, si *N* és el nombre d'esquemes de compressió suportats en un SGBD, cada operador unari haurà de tenir *N+1* implementacions, una per a cada esquema de compressió i una per a operar sobre les dades no comprimides.

En el cas dels operadors binaris (com seria el cas, entre d'altres, de la resolució d'operacions de combinació –o *join*–) el tema és molt més dramàtic, ja que el nombre d'implementacions de cada operador serà $(N+1)^2$, que són les possibles combinacions de tipus de codificació d'entrada per a cada operador. És clar, doncs, que els costos en implementació i el seu futur manteniment no són menyspreables. Cal dir que els números proporcionats apunten als casos pitjors. En alguns casos, és possible utilitzar una mateixa implementació per a esquemes de compressió diferents. També hi ha algunes tècniques, com per exemple, la dels *compression blocks* (vegeu el treball de la tesi d'Abadi referenciat en aquesta presentació) que permeten simplificar les implementacions dels operadors i reduir el nombre de versions a implementar. No obstant això, malgrat aquestes millores, les implementacions dels operadors continuen sent més complexes que les seves homòlogues que tracten amb dades no comprimides.

Una altra pregunta (relacionada amb el que acabem de discutir) que podem plantejar-nos és, quin tipus d'operacions són capaces de treballar amb dades comprimides? Segons la bibliografia (vegeu Graefe i Saphiro i la tesi d'Abadi), les operacions de *natural join* es poden realitzar sobre dades comprimides sempre que els participants en l'operació de *natural join* utilitzin el mateix esquema de compressió. Una possibilitat per a garantir aquesta condició en un SGBD podria ser requerir que tots els atributs (o columnes) d'un mateix domini (és a dir, que representin un mateix concepte del món real) utilitzin el mateix esquema de compressió. Altres operacions permeses són les cerques per valor (presents en les operacions de selecció, per exemple), que es podran executar sobre dades comprimides quan l'esquema de compressió utilitzat preservi l'ordre dels elements. Finalment, la projecció i l'eliminació de duplicats també es poden executar sobre dades comprimides. L'operador d'agrupació sembla que no sempre es pot executar sobre les dades comprimides. L'atribut (o columna) a agrupar sí que es pot generar a partir de les dades comprimides, però no hi ha garantia que l'atribut (o atributs) sobre el qual es realitzin les operacions d'agrupació (execució d'operacions `COUNT`, `SUM`, `AVG`, etc.) es pugui calcular mitjançant les dades comprimides.

Una vegada explicats els avantatges i desavantatges de treballar amb dades comprimides, vegem un exemple en què una sentència `SELECT` opera directament sobre les dades comprimides. L'exemple utilitzarà la taula de clients *Customer* que

hem usat al llarg d'aquest material didàctic dedicat als magatzems de columnes. L'operació a realitzar serà una consulta que ens retorni el nombre de clients, agrupats per gènere, de la població de Jaramillo Quemado (hem escollit aquesta població perquè només té 4 habitants). Per a l'exemple, suposarem que la columna *Customer_Gender* està comprimida mitjançant el *Bit-vector encoding* i la columna *Customer_City* mitjançant el *Run-length encoding*.



Tal com hem comentat, la columna *Customer_City* contindrà les dades de la ciutat on viuen els clients de l'empresa comprimides mitjançant un esquema de *Run-length encoding*. Recordem que l'empresa té 46 milions de clients (la població espanyola). Suposem que la columna està ordenada alfabèticament per població. Per tant, cada cel·la de la columna estarà composta per una tripleta amb el nom d'una ciutat, el nombre de la cel·la de la seva primera ocurrència en la columna original i el nombre de vegades que es repeteix. Com que està ordenada alfabèticament pel nom de la ciutat, només hi haurà una tripleta per a cada ciutat.

D'altra banda, tal com vam veure en la presentació de la compressió de dades, la columna *Customer_Gender* s'ha comprimit mitjançant dos vectors de bits amb tantes posicions com registres (o files) té la taula *Customer*. El primer vector, anomenat *Women*, conté un valor binari a cada cel·la que indica si el client associat al número de registre *i* de la taula és una dona (valor 1 per a la posició *i*-èssima) o no (valor 0 per a la posició *i*-èssima). El vector *Men* contindrà un valor binari que indiqui el contrari, si el client és un home.

En el cas que ens ocupa, l'SGBD podria executar la consulta directament sobre les dades comprimides. A la transparència es mostra de forma simplificada com es realitzaria. Primer, se seleccionaria la posició dels registres (o files) de la taula que satisfacin la condició indicada en el `WHERE` (clients que viuen a Jaramillo Quemado). Com que la columna *Customer_City* està ordenada, l'SGBD podria obtenir eficientment la cel·la associada a la ciutat de Jaramillo Quemado:

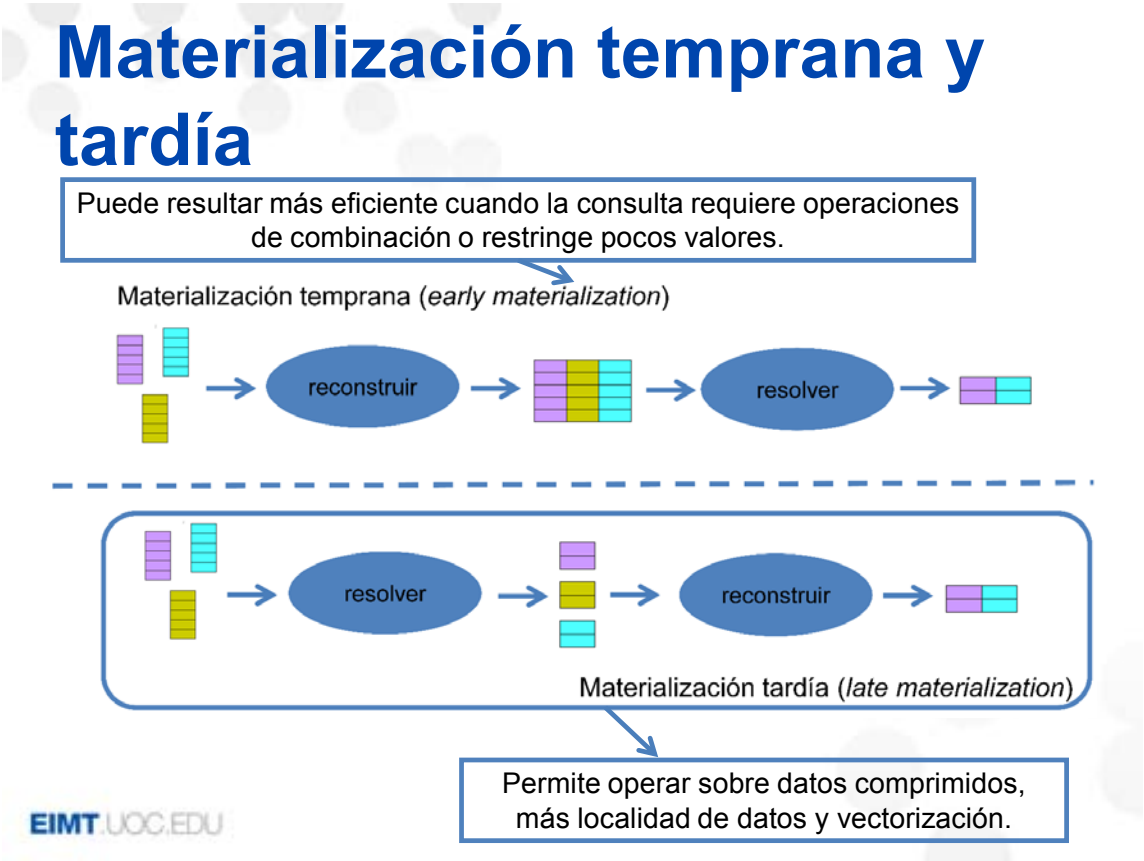
(Jaramillo Quemado, 1.457.111,4)

Suposem que el resultat és la cel·la anterior. Assumint que l'algorisme que implementa l'operació de selecció és capaç d'interpretar les dades comprimides mitjançant el *Run-length encoding*, l'SGBD interpretarà que els clients de Jaramillo Quemado són 4 i estan situats a les cel·les 1.457.111, 1.457.112, 1.457.113 i 1.457.114. Per a obtenir el resultat de la consulta, l'SGBD haurà de comptar quants homes i dones hi ha en aquestes posicions i retornar aquest valor. Per a fer aquest càlcul, assumint que és capaç d'operar amb dades comprimides mitjançant el *Bit-vector encoding*, simplement ha de comptar els uns (1) que hi ha a les posicions 1.457.111, 1.457.112, 1.457.113 i 1.457.114 dels vectors *Women* i *Men*. El resultat és *Men* 2 i *Women* 2. Aquest seria justament el resultat que retornaria la consulta.

Tal com es pot apreciar en aquest exemple, operar sobre dades comprimides resulta ser eficient, ja que evitarà descomprimir les columnes *Customer_City* i *Customer_Gender*. A més, permet reduir el nombre d'operacions a realitzar, ja que s'han pogut seleccionar 4 registres fent una sola comparació sobre la cel·la comprimida. També redueix la quantitat de dades que s'han de moure des del dispositiu d'emmagatzematge extern (disc) a la CPU.

Per a resoldre la consulta sobre les dades comprimides, l'SGBD hauria de tenir implementada una versió del l'operador de comparació (que resol l'operació de selecció plantejada en la consulta, *Customer_City="Jaramillo Quemado"*) que suporti un esquema de compressió de *Run-length encoding*. D'altra banda, l'operació de projecció (que projecta la columna *Customer_Gender*) i l'operació `COUNT` haurien de permetre operar sobre les dades comprimides mitjançant el *Bit-vector encoding*.

Bé, una vegada vist com operar amb dades comprimides, vegem com es poden processar les consultes formulades sobre la BD en els magatzems de columnes. Per a això, començarem veient els diferents tipus de materialització que existeixen i quin d'aquests és el més adequat en els magatzems de columnes. Recordeu que la materialització és el procés que reconstrueix les files d'interès d'una taula a partir de les seves columnes.



Tal com ja hem comentat en aquest material didàctic (en concret, en la presentació dedicada a les característiques dels magatzems de columnes), la materialització tardana és una de les característiques pròpies dels magatzems de columnes que pot millorar el rendiment de la BD, en cas d'utilitzar-se adequadament.

El fet és que, en un magatzem de columnes, les dades de cada taula s'emmagatzemen per columnes i en diverses ubicacions del disc (en definitiva, cada columna d'una taula, almenys, s'emmagatzema en un fitxer separat sense barrejar-se amb dades d'altres columnes). No obstant això, els usuaris, els programes i, fins i tot, els *drivers* (ADO.NET, JDBC, etc.) que accedeixen a la BD esperen que el resultat d'una consulta sigui un conjunt de files. Per tant, en el pla d'accés de la majoria de consultes, les dades de diverses columnes s'hauran de combinar en files d'informació. Aquest procés de reconstrucció de files es denomina materialització de files i és molt similar (des del punt de vista conceptual) a una operació de combinació.

Cal destacar que, en el cas dels magatzems de columnes, per a materialitzar les files s'han de descomprimir les dades quan els algorismes de compressió aplicats hagin estat els que redueixen el nombre de cel·les de la columna (aquest és el cas, per exemple, dels esquemes de compressió *Run-length encoding* i *Cluster encoding*).

La decisió sobre quan cal realitzar el procés de materialització de files en magatzems de columnes és similar a la decisió de quan executar l'execució d'operacions de

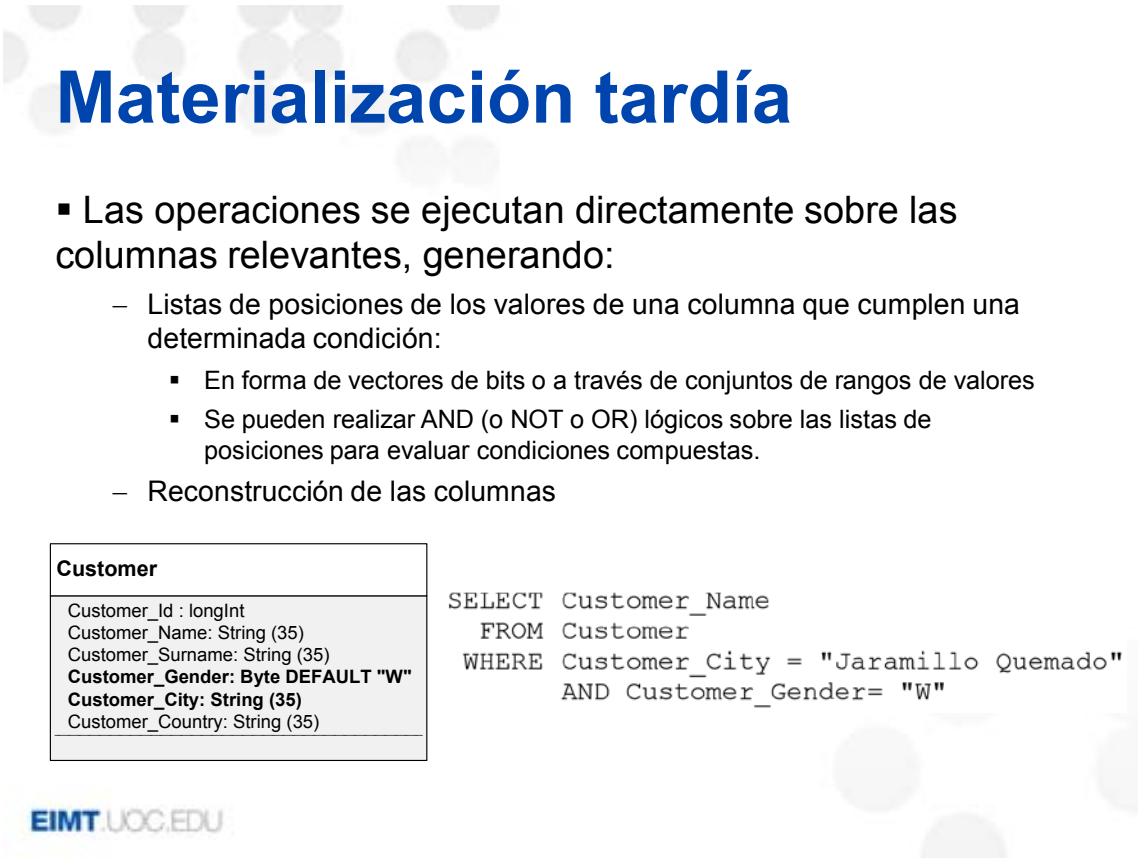
projecció en els magatzems de files, que també es pot realitzar de forma primerenca durant el procés d'execució d'una consulta o de forma més tardana.

La qüestió fonamental és a quin moment, durant el procés de resolució de la consulta, es realitza la reconstrucció o materialització de les files. En principi, hi ha dos enfocaments: la materialització primerenca (en anglès, *early materialization*) o la materialització tardana (*late materialization,* en anglès).

La materialització primerenca realitza la reconstrucció de les files a partir de les dades de les columnes rellevants al principi del pla d'accés de la consulta. En aquest cas, com que tenim les files creades abans de la resolució de la consulta, es poden aplicar els algorismes existents en els magatzems de files per a la resolució de les operacions indicades en la consulta (selecció, projecció, agregació, combinació, etc.). Si bé això permet estalviar costos de desenvolupament, desaprofita algunes de les possibilitats dels magatzems de columnes a l'efecte de rendiment. Entre d'altres, no permet l'execució d'operacions en bloc (tal com veurem en la secció següent), no permet l'execució d'operacions directament sobre les dades comprimides, no aprofita la localitat de les dades, requereix més traspàs de dades entre el disc, la memòria principal i la CPU, i, en molts casos, materialitza registres (o files) innecessaris, ja que les consultes poden requerir processar menys files (en les operacions de selecció, els predicats que incorporen poden reduir el nombre de files de sortida i les agregacions combinen diferents files en una de sola). Per aquests motius, malgrat sigui més complexa, la materialització tardana se sol utilitzar de forma intensiva en els magatzems de columnes.

D'entrada, podem pensar que la materialització tardana sempre genera millors resultats, o sigui, que és més eficient. Això no és del tot cert. En alguns casos, la materialització primerenca és més convenient. El problema principal de la materialització tardana és que, alguns cops, les columnes necessiten ser accedides múltiples vegades en el pla de la consulta. Per exemple, suposem que s'accedeix a una columna una vegada per a obtenir les posicions que satisfan un predicat i una segona vegada (més endavant en el pla) per a recuperar els seus valors. El sobrecost es pot incrementar encara més en el cas que estiguem fent una combinació i s'hagi d'ordenar la columna segons la clau de la combinació. En la materialització primerenca, es genera un conjunt de files la primera vegada que es consulta la columna, eliminant la necessitat d'accessos futurs a la columna.

Per tant, i malgrat que la materialització tardana permet algunes optimitzacions, si el cost d'accedir repetidament a unes mateixes columnes és elevat, pot ser més convenient utilitzar la materialització primerenca. D'altra banda, també pot ser més convenient utilitzar la materialització primerenca quan els predicats que incorpora una operació de selecció d'una consulta no són molt restrictius (en altres paraules, generen un nombre elevat de files).



Quan s'utilitza la materialització tardana, les operacions implicades en el pla de la consulta s'executen directament sobre les columnes rellevants. Per norma general, cada operació s'executa sobre una columna (operacions unàries) o dues columnes (operacions binàries).

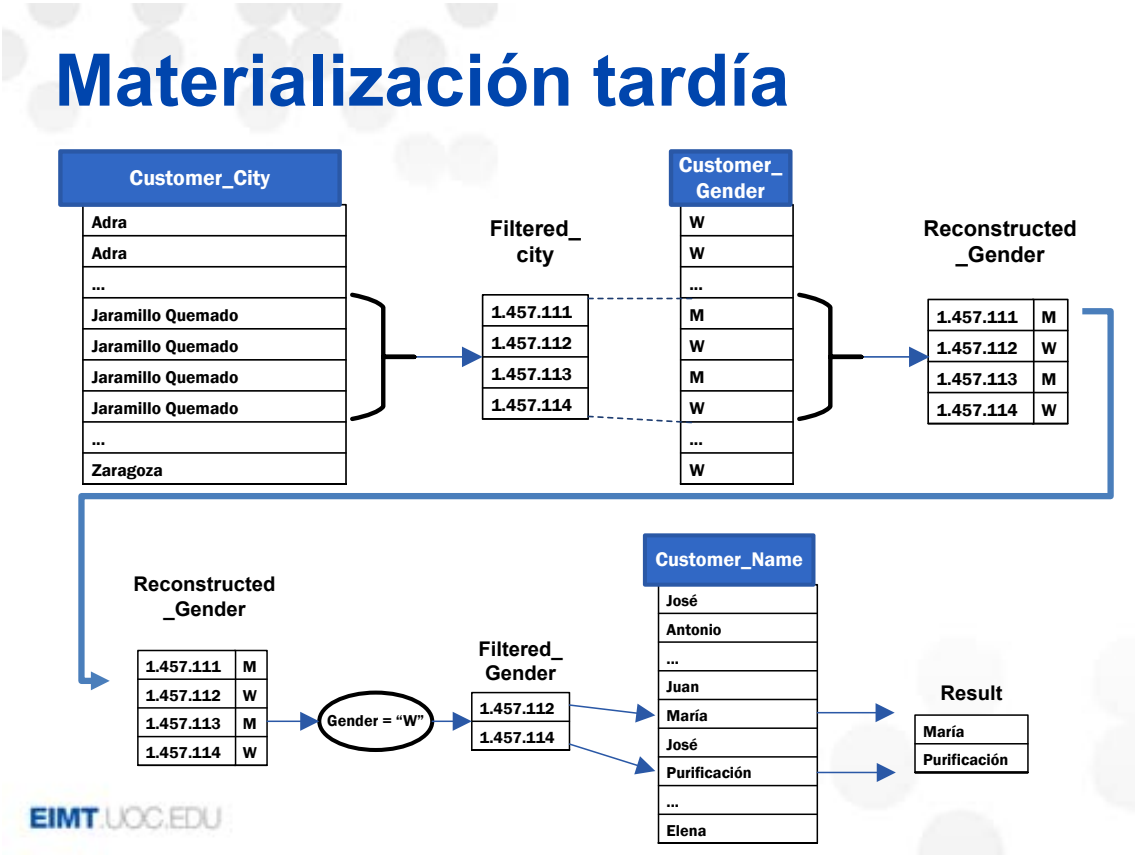
Quan s'apliquen operacions de selecció en una consulta (aquestes operacions apareixeran en clàusules `WHERE`) se solen generar llistes de posicions que indiquen el resultat de les operacions de selecció sobre les columnes. Cada llista de posicions indica les posicions dels valors de la columna que satisfan la condició (o predicat) avaluada. Aquestes llistes poden tenir forma de vectors de bits o de conjunts de rangs de posicions (*rowid* o identificadors de files), en funció de les operacions a considerar. En el cas que tinguem condicions sobre més d'una columna (o sigui, en el cas que s'incloguin diferents operacions de selecció en la consulta), es poden utilitzar operacions *AND* i *OR* lògiques sobre les llistes de posicions (quan aquestes prenen la forma de vectors de bits) per a calcular ràpidament el conjunt de files que compleixen les diferents condicions.

Per exemple, suposem una consulta que realitza un filtre (o sigui, imposa condicions) sobre dos atributs (o columnes) i projecta un tercer atribut de la taula que compleixi les dues condicions indicades. En aquest cas, es podria generar un vector de bits per a cada columna filtrada, indicant les cel·les de cada columna que satisfan la condició (amb un 1). Posteriorment, es realitzaria un *AND* lògic de tots dos vectors de bits per a obtenir un nou vector de bits que representa les cel·les de la tercera columna que han

de ser seleccionades en la consulta. Al final de la consulta, i abans de retornar el resultat, es combina el vector de bits que resulta de l'operació lògica d'*AND* amb la columna (o atribut) d'interès en la consulta per a generar el resultat de la consulta (un conjunt de files que contenen un únic valor). Com que es retarda la creació de files tan tard com sigui possible, s'aconsegueix una major localitat de dades i augmenta la possibilitat d'operar sobre les dades comprimides, d'operar amb dades en bloc i d'evitar generar files innecessàries.

Cada vegada que s'han de consultar els valors d'una columna d'acord amb una llista de posicions diem que es realitza una reconstrucció de la columna. En la reconstrucció es crea una nova columna que constitueix un subconjunt de la columna original, ja que només conté els valors que compleixen les condicions representades per la llista de posicions. Les reconstruccions de columnes se solen executar múltiples vegades dins del pla de cada consulta.

A continuació, vegem un exemple de materialització tardana utilitzant la nostra BD d'exemple. En aquest exemple suposarem que es realitza una consulta que pretén obtenir els noms dels clients que viuen a la ciutat Jaramillo Quemado. Per a simplificar l'exemple, suposarem que les dades no estan comprimides, encara que aquesta consulta es podria realitzar igualment sobre dades comprimides.



En el cas d'utilitzar la materialització primerenca, el primer que faríem quan executem la consulta seria la materialització. Per tant, es crearien més de 46 milions de files de clients amb els valors de les columnes *Customer_City, Customer_Gender* i *Customer_Name*, que són les tres columnes implicades en la consulta que es pretén resoldre. A partir d'aquest moment, s'executarien les diferents operacions de la consulta (en el nostre exemple es tracta d'operacions de selecció i projecció).

No obstant això, si utilitzéssim una estratègia de materialització tardana, primer executaríem les operacions de la consulta sobre les columnes rellevants i, quan tinguéssim el resultat esperat, podríem materialitzar les files que conformen el resultat final de la consulta (sobre el nostre exemple, i atès que solament projectem les dades de la columna *Customer_Name*, cada fila contindria un únic valor). Partint del supòsit que usem la materialització tardana, podríem descompondre la consulta d'exemple en les operacions següents (cal tenir en compte que hi podrien haver altres estratègies de resolució, amb propostes diferents d'operacions, igualment vàlides per a resoldre la consulta):

1. Identificar les files que compleixen la condició *Customer_City = "Jaramillo Quemado"*.

2. Reconstruir la columna *Customer_Gender* d'acord amb les files seleccionades en l'anterior operació.

3. Identificar les files de la columna creades en el pas anterior que contenen el valor *W* (gènere femení).

4. Reconstruir la columna *Customer_Name* a partir de la llista de posicions obtinguda en el pas anterior.

Després del pas 4 ja tindríem el resultat esperat. En el cas que la consulta requerís més columnes, es reconstruirien les columnes afectades després del pas 3 i es materialitzarien posteriorment.

Per a l'exemple proposat, assumirem que les llistes de posicions es generen mitjançant una llista que conté les posicions (és a dir, els *rowid* o identificadors de files) dels valors que satisfan la condició.

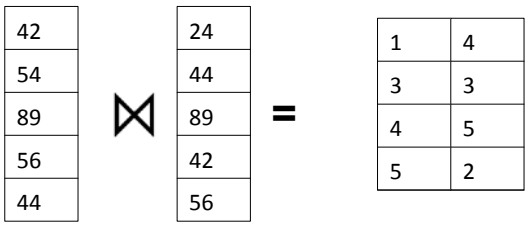
En l'exemple podem veure que la primera operació identificaria les posicions dels valors de la columna *Customer_City* que contenen el valor Jaramillo Quemado. Com a resultat, es generarà una llista de 4 posicions, amb els valors 1.457.111, 1.457.112, 1.457.113 i 1.457.114 que identifiquen els valors d'interès. Òbviament, l'execució d'aquesta operació seria més o menys eficient en funció de si la columna està ordenada, de si els seus valors estan comprimits i d'altres factors que veurem en l'apartat següent.

Posteriorment es reconstruirà la columna *Customer_Gender* (afectada en el predicat següent de la consulta) amb els valors de la llista de posicions creada. Com a resultat, es generarà una nova columna amb els valors del gènere de les persones que resideixen a Jaramillo Quemado. Com que només desitgem obtenir els noms de les dones, realitzarem una selecció dels valors de la nova columna, seleccionant els que tenen un valor igual a *W*. El resultat d'aquesta selecció generarà una nova llista de posicions. La nova llista contindrà les posicions 1.457.112 i 1.457.114, que són les posicions que corresponen a les dones que resideixen a Jaramillo Quemado. Aquestes posicions indiquen les files que volem obtenir. Per tant, l'última operació serà la de reconstruir la columna *Customer_Name* per als valors seleccionats. Una vegada fet això tindríem el resultat final de la consulta.

Com es pot observar, el fet de realitzar una materialització tardana, permet optimitzar la consulta en aquest cas d'exemple. El motiu és que en la primera operació es poden descartar més del 99 % dels valors de la taula. En el cas d'haver utilitzat la materialització primerenca, aquest descart no s'hagués pogut fer fins després de materialitzar els 46 milions de files amb els valors de les tres columnes d'interès per a la consulta. Per tant, s'hauria generat un gran nombre de files que són irrellevants per a la resolució 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

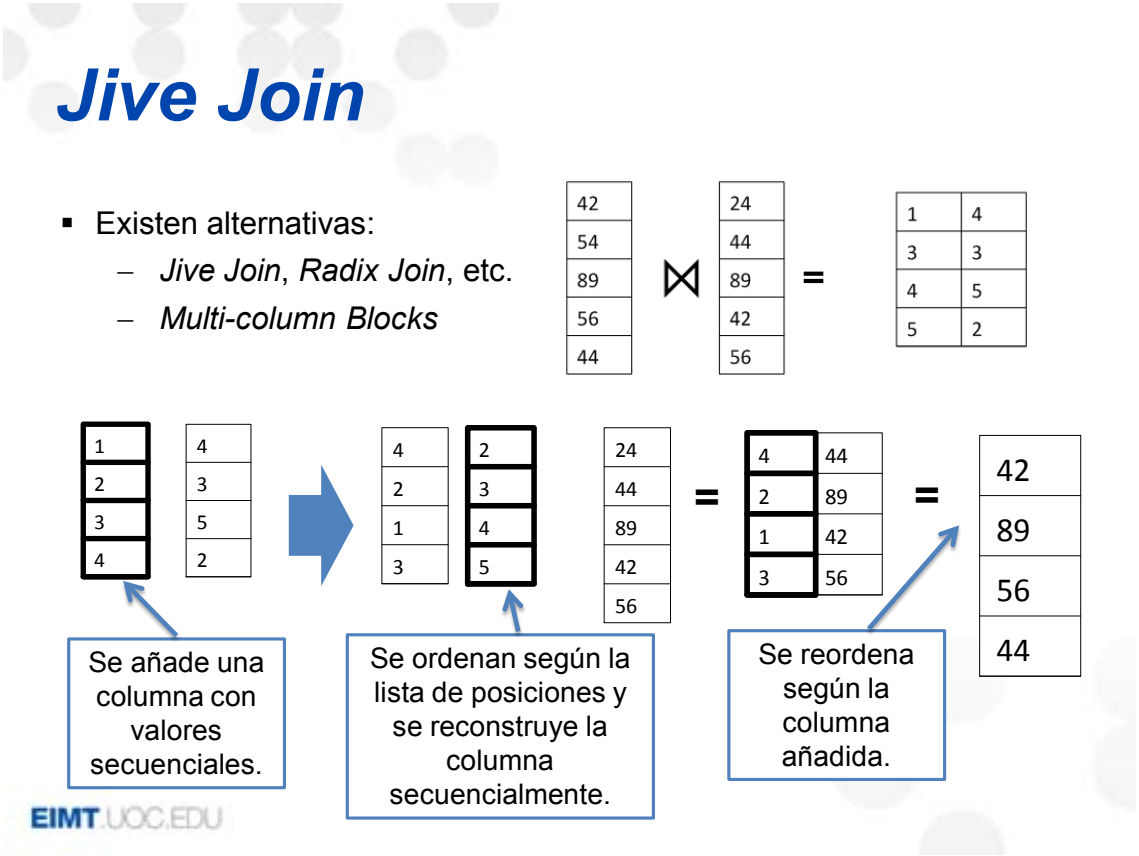
Quan s'usa la materialització primerenca, els registres (o files) es creen abans de realitzar les operacions de combinació i, per això, es poden utilitzar els operadors de combinació dissenyats i optimitzats per als magatzems de files.

Però, què succeeix quan es desitja utilitzar la materialització tardana? La manera més senzilla de realitzar una combinació en un magatzem de columnes és generar un parell de llistes de posicions, una llista per a cada columna a combinar. Les llistes de posicions indicaran les posicions (els *rowid*) dels valors equivalents en ambdues columnes, de forma semblant al que es fa en una operació de *hash join*. A la transparència podem veure un exemple en què fem la combinació de dues columnes (aquestes columnes i els seus valors, en l'exemple, es mostren a l'esquerra). El resultat de la combinació serien 2 llistes de posicions (a la part dreta de l'exemple) de 4 files cadascuna, una per a la posició del valor 42, una altra per a la posició del valor 89, una altra per a la posició del valor 56 i una altra per a la posició del valor 44.

Per tant, les llistes de posicions indicaran, per a cada valor coincident (o sigui, per als valors que formen part del resultat de l'operació de la combinació), la seva posició en ambdues columnes. En conseqüència, la primera posició de les llistes contindrà un 1 per a la primera columna i un 4 per a la segona, ja que el valor 42 es troba en la primera posició de la primera columna i en la quarta posició de la segona. La segona posició de les llistes contindrà un 3 per a la primera columna i un 3 per a la segona, i així successivament. Per a la majoria dels algorismes de combinació que puguem utilitzar, es generaria la primera llista de posicions ordenada (la de l'esquerra) mentre que la

segona llista de posicions no ho estaria (la de la dreta). Això és així perquè sobre la columna de l'esquerra s'itera de forma seqüencial, mentre que per a la columna de la dreta es requereix buscar els valors de la columna que satisfan el valor de la columna de l'esquerra.

Tenir la llista de posicions de la segona columna desordenada és problemàtic, atès que implica realitzar un conjunt d'accessos directes per posició per a reconstruir la porció de la columna que ens interessa. Els accessos directes per posició, en altres àrees de la informàtica, com seria la dels sistemes operatius, es denominen accessos o lectures aleatòries, tal com s'indica a la transparència, i poden tenir un cost prohibitiu depenent del nombre d'elements inclosos en la llista de posicions. Per tant, és millor accedir a aquests elements realitzant un recorregut seqüencial per posició. El problema és que no es podrà reconstruir la porció de la columna que ens interessa a partir d'aquesta llista fent un únic recorregut seqüencial sobre el fitxer de dades en què s'emmagatzema la columna en qüestió. En el pitjor dels cassos, serà necessari realitzar tants recorreguts seqüencials sobre el fitxer com valors desitgem recuperar (sobre el nostre exemple, en concret, podria ser necessari haver de realitzar fins a 4 recorreguts seqüencials).



Hi ha algorismes que aborden aquest problema i proporcionen solucions satisfactòries. Un parell d'aquests són el *Jive join* i el *Radix join*. Altres alternatives que utilitzen els SGBD són la materialització de les columnes rellevants abans de realitzar la combinació o l'ús de blocs multicolumna.

Vegem de forma sintètica un parell d'exemples d'aquestes estratègies. Podeu trobar més informació en les referències indicades al final de la presentació.

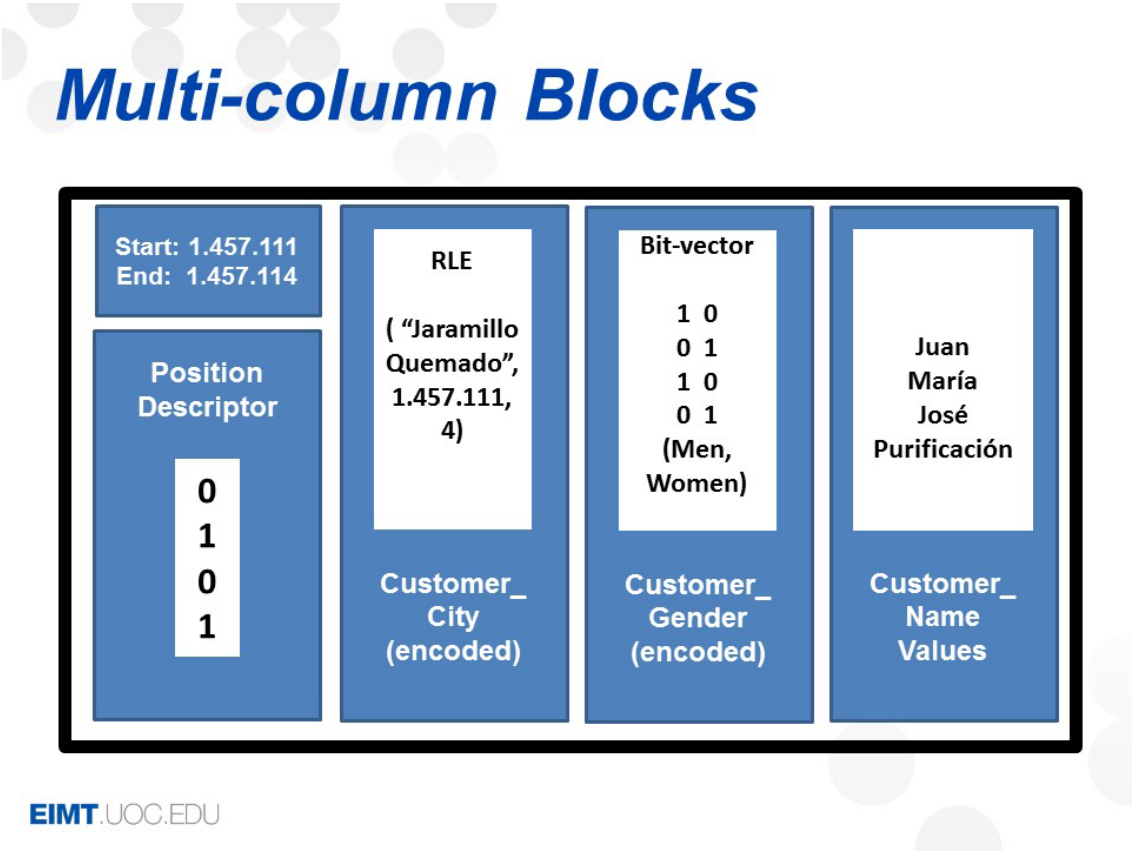
Comencem veient un exemple de l'ús de l'algorisme *Jive join* per a obtenir una reconstrucció de la segona columna de forma ordenada.

Assumim que es realitza la combinació que es mostra a la part superior dreta de la transparència i que com a resultat s'obtenen les dues llistes de posicions mostrades. La segona llista de posicions conté les posicions dels valors d'interès de forma desordenada.

Suposem que apliquem l'algorisme de *Jive join* per a materialitzar la columna a partir de la seva llista de posicions. Per a això, serà necessari seguir una sèrie de passos. El primer pas consisteix en afegir una nova columna a la llista de posicions, que contingui la seqüència *1, 2, ... N*, on *N* és el nombre total de posicions de la llista de posicions, tal com es mostra a la figura de més a l'esquerra de la part inferior de la transparència. En el cas que vulguem obtenir els valors de la columna (és a dir, si volem reconstruir la columna) es podria (com a segon pas) ordenar la llista de

posicions d'acord amb les posicions dels elements rellevants (vegeu la part central de la figura de la part inferior de la transparència). Una vegada fet això, tindríem les posicions d'interès dels valors de la columna a reconstruir de forma seqüencial i podríem reconstruir la columna fàcilment fent una sola lectura seqüencial del fitxer en què estan les dades de la columna, tal com es mostra a la transparència.

Fixeu-vos que quan s'ordena per posició, la seqüència que hem afegit en el pas anterior ha quedat desordenada. Si més endavant hem de reconstruir l'ordenació original, per exemple, per a realitzar una materialització de columnes de taules diferents, únicament hauríem de reordenar la columna resultant en funció de la seqüència introduïda.



Una altra opció per a minimitzar les problemàtiques associades a la materialització tardana és continuar treballant amb columnes, però emmagatzemar els resultats intermedis de les operacions incloses en les consultes de forma agrupada en funció de les columnes i les files d'interès. D'aquesta forma s'emmagatzemen junts fragments de columnes que potencialment es consultaran junts més endavant. La filosofia és mantenir les dades agrupades per columnes i comprimides però en una estructura que sembli (o simuli) que s'ha realitzat una materialització primerenca, per així minimitzar els problemes de posteriors accessos a les columnes. Aquestes estructures es denominen blocs multicolumna (*multi-column blocs,* en anglès).

A la transparència podem veure el bloc multicolumna rellevant per a la consulta de l'exemple anterior (obtenir el nom dels clientes de Jaramillo Quemado). Els blocs multicolumna contenen:

1. Informació sobre el rang de valors continguts en el bloc, per exemple a la transparència s'indica que el bloc conté les columnes que van de la fila 1.457.111 a la fila 1.457.114, que corresponen amb les files dels clients que resideixen a Jaramillo Quemado. El baix nombre de valors respon a l'exemple concret que estem utilitzant, tal com ja hem discutit amb anterioritat. En situacions normals, el nombre de valors continguts, potencialment, serà més elevat.

2. Un *array* de «minicolumnes», que conté els fragments de columnes diferents que corresponen al rang indicat en el punt anterior. Per tant, en l'exemple tindríem fragments de les columnes implicades en la consulta (*Customer_City*, *Customer_Gender* i *Customer_Name*). Els fragments de les columnes es poden guardar comprimits. A la figura d'exemple, es guarda un fragment de la columna *Customer_City* comprimida mitjançant el *Run-length encoding*, un fragment de la columna *Customer_Gender* comprimida mitjançant el *Bit-vector encoding* i un fragment de la columna *Customer_Name* sense comprimir.

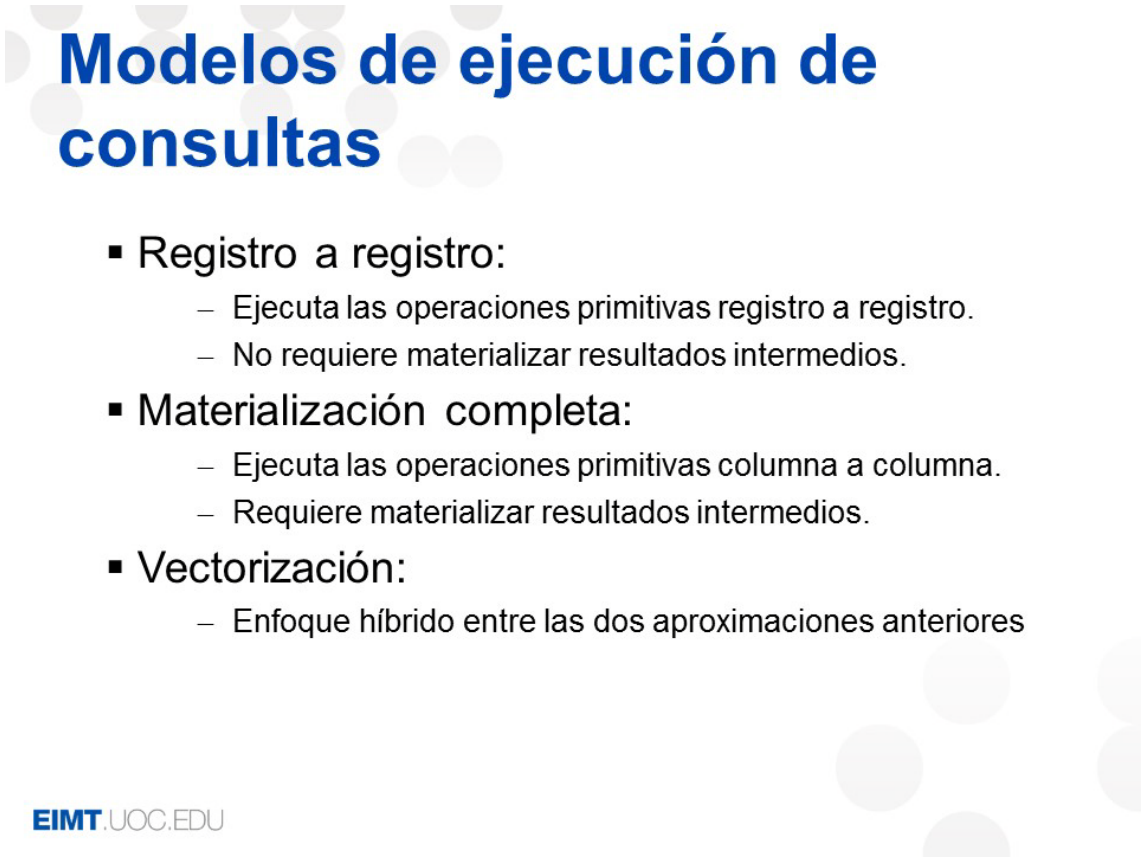
3. Un descriptor de posicions que indica les posicions del bloc que són vàlides. A l'exemple, una vegada aplicat el filtre *(Customer_Gender="W")* el descriptor de posicions indicaria que les posicions vàlides són la segona i la quarta.

Procesamiento de consultas

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

EIMT.UOC.EDU

A continuació, veurem els diferents models d'execució de consultes i com alguns d'aquests poden utilitzar l'execució en bloc per a optimitzar les consultes en magatzems de columnes.



La vectorització és una tècnica complexa. Conèixer el seu funcionament de forma detallada queda fora de l'abast d'aquest material didàctic. Malgrat això, a continuació farem una petita introducció (des del punt de vista conceptual i a grans trets) de la vectorització i una enumeració dels avantatges que pot oferir.

L'execució d'una consulta en un SGBD es pot fer de diferents formes: 1) registre a registre (o si ho preferiu, fila a fila o tupla a tupla), 2) mitjançant la materialització completa i 3) via vectorització.

L'execució fila a fila, també coneguda com *Volcano-style* o *Tuple-at-time pipelining*, és la més clàssica des d'un punt de vista històric i també la més elegant des d'un punt de vista d'enginyeria del programari. En aquest model d'execució, les diferents operacions involucrades en el pla d'una consulta s'executen de forma sincronitzada i seguint l'estructura (normalment d'arbre) definida en el pla de la consulta. En aquest cas, cada operador (o operació) relacional implicat en el pla de la consulta produeix una nova tupla (o fila o registre) cada vegada, a partir de les dades rebudes de l'execució de les seves operacions filles en l'arbre, i genera una nova tupla que envia a la seva operació pare. Per tant, tenim un procés en forma d'arbre (cada node de l'arbre representa una operació relacional que és necessari executar per a resoldre la consulta) que gestiona els registres (o files o tuples) un a un, i en què el control del flux d'execució (la sincronització) es realitza des dels nivells superiors de l'arbre (en altres paraules, des de les operacions pare). Com que es processen els registres un a un, les

operacions de l'SGBD treballen amb valors individuals i també es minimitza la necessitat d'emmagatzemar resultats intermedis.

El segon model d'execució de consultes, la materialització completa, executa les operacions presents en el pla d'execució de la consulta de forma seqüencial. En concret, cada operació present en el pla de la consulta s'executa consumint totalment les dades d'entrada (el conjunt de files a processar, en definitiva), generant una sortida (o resultat intermedi) que, al seu torn, s'emmagatzema (o sigui, es materialitza). L'execució de les operacions s'encadena seqüencialment una després de l'altra per a generar la solució final (en altres paraules, per a generar el conjunt de files que constitueixen el resultat de la consulta).

Aquest model d'execució, per exemple, és l'utilitzat per MonetDB. Es tracta d'un model que aconsegueix execucions més eficients en termes de CPU. No obstant això, pot requerir un ús de recursos excessiu perquè genera resultats intermedis que caldrà emmagatzemar, especialment en consultes que continguin operacions de selecció que incloguin predicats poc restrictius. Per exemple, suposeu una consulta que calculi la mitjana de totes les vendes amb un valor de més d'1 €. En cas que la gran majoria de les nostres vendes (imaginem que al voltant del 99 %) fossin superiors a 1 €, l'operació que filtra les files amb vendes superiors a 1 € generaria una materialització (o resultat intermedi) que contindria al voltant del 99 % de les files de la taula original. Això podria ser massa costós en entorns que gestionen una gran quantitat de dades. Com que es processa tota l'entrada alhora, les operacions de l'SGBD treballen amb conjunts de valors.

El model d'execució vectorial (o vectorització) és una solució de compromís entre les dues aproximacions anteriors, en què l'execució es continua realitzant de forma iterativa, com en l'execució fila a fila, però els operadors s'executen sobre un conjunt de valors en comptes de sobre un valor individual. Això permet executar les operacions en bloc i reduir el nombre d'iteracions necessàries. Per tant, cada vegada que s'iteri, es processarà un vector d'N tuples que conté les dades sobre les quals s'operarà. Respecte a les operacions, en aquest cas, processarien vectors de longitud N. La grandària del vector normalment es defineix de manera que càpiga confortablement en la *caché*. VectorWise, per exemple, utilitza la vectorització usant vectors de 1.000 posicions per defecte.

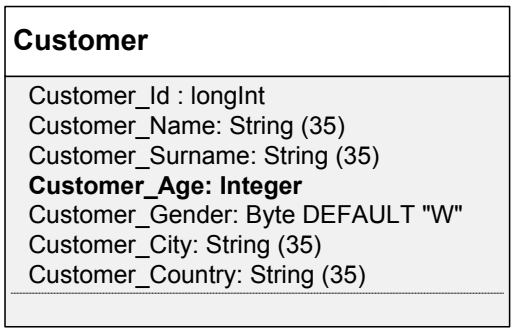
Actualment, les tècniques de materialització completa i vectorització poden representar millores importants en l'eficiència, a causa de la capacitat d'executar operacions amb múltiples dades (SIMD) dels processadors actuals.

Els processadors actuals suporten execucions de tipus SISD (*single instruction single data,* en anglès) i SIMD (*single instruction multiple data,* en anglès). Bàsicament aquestes sigles indiquen com s'executen les operacions a nivell de processador. Si utilitzem SISD, les operacions del processador s'executaran sobre una dada única. Les tècniques SIMD permeten executar una operació sobre un conjunt de dades. Per

exemple, si volem incrementar en una unitat tots els valors d'una columna i utilitzem tècniques SISD, haurem d'executar l'operació d'increment sobre cada valor de la columna. No obstant això, si utilitzem tècniques SIMD, podrem executar l'operació sobre tota la columna (o part de la mateixa) de forma atòmica. Utilitzar operacions SIMD per a executar operacions de la BD permet millorar el rendiment de les consultes.

L'arquitectura SIMD es pot aplicar fàcilment en els magatzems de columnes per a executar una sola operació sobre una columna (o un subconjunt d'aquesta) en bloc. Aquesta tècnica rep el nom d'execució en bloc (en anglès, *block iteration*). Com que els valors de les columnes estan emmagatzemats de forma consecutiva i les operacions de la BD s'executen sobre els valors de les columnes, és fàcil moure les dades juntes d'una columna a la CPU i aplicar operacions de tipus SIMD que executin una mateixa operació per a tots els valors de la columna.

Ejemplo de materialización completa

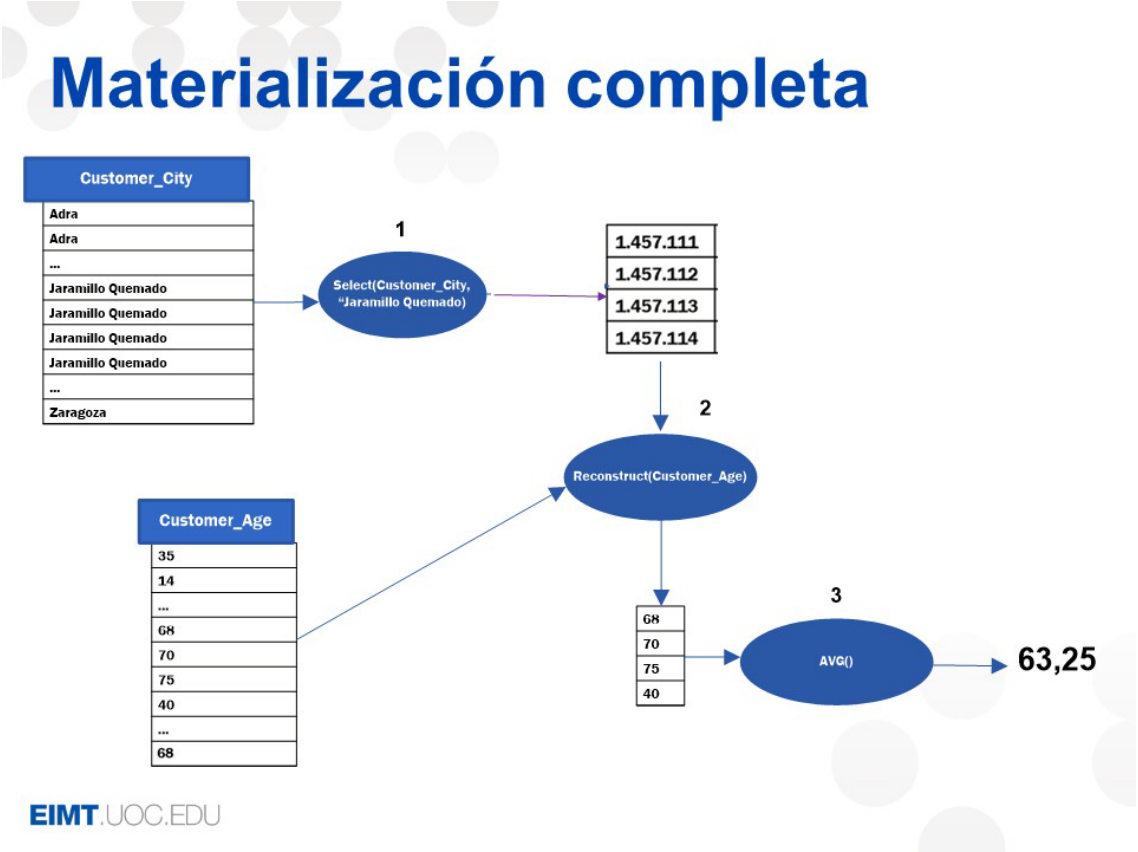



```
SELECT AVG(Customer_Age)
  FROM Customer
 WHERE Customer_City = "Jaramillo Quemado"
```

EIMT.UOC.EDU

A continuació, vegem un exemple de com funciona la materialització completa. No parlarem detalladament de la vectorització però, conceptualment, podem entendre-la com un cas particular de materialització completa que requereix menys recursos per a la materialització de resultats intermedis i que, alhora, pot treure profit de l'ús de les tècniques SIMD.

En l'exemple següent observarem com s'executaria, utilitzant la materialització completa, la consulta que s'indica a la transparència. En aquest cas particular, la consulta demana la mitjana d'edat dels habitants de la població de Jaramillo Quemado. Per a això, hem afegit un atribut (o columna) edat (*Customer_Age*) a la taula *Customer*.

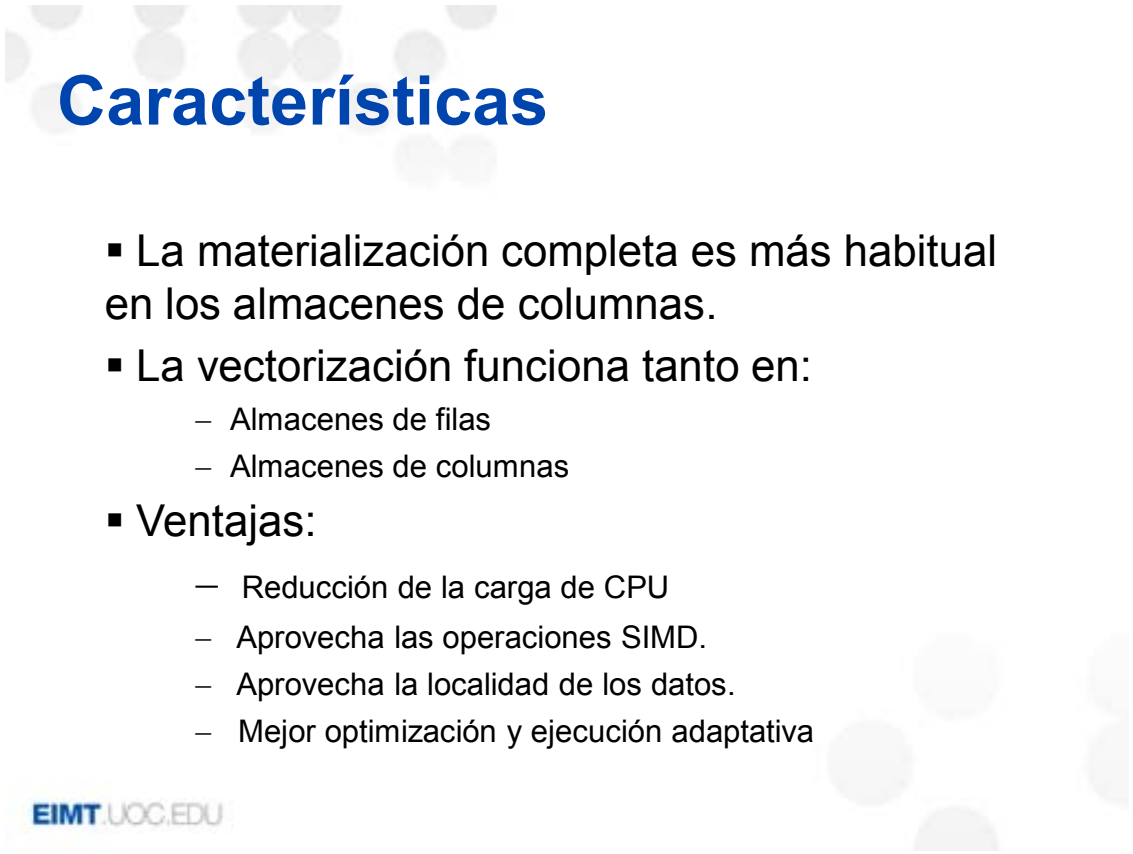


Suposarem que la consulta de l'exemple s'executa utilitzant estratègies de materialització tardana. En aquest cas, un possible pla d'execució per a la consulta podria ser el següent:

1. Selecció dels clients que resideixen a Jaramillo Quemado.
2. Reconstrucció de la columna *Customer_Age* dels clients que resideixen a Jaramillo Quemado (a partir de la llista de posicions obtinguda en l'operació anterior).
3. Calcular l'edat mitjana (mitjançant una operació `AVG`) dels clients seleccionats

A la transparència podem veure com s'executarien les tres operacions. Per a executar la primera operació assumirem que la BD oferiria una operació que, donada una columna i un valor concret, obté les posicions d'una columna que contenen aquest valor. Internament, l'operació de l'SGBD es podria implementar mitjançant operacions SIMD en el processador, amb una operació d'igualtat en bloc per exemple. Això podria accelerar enormement el temps de resposta. Com a resultat d'aquesta operació obtindríem la llista de les 4 posicions que correspon als clients que resideixen a Jaramillo Quemado. Aquesta llista de posicions, juntament amb la columna *Customer_Age*, seran les entrades de l'operació següent que, donada una llista de posicions i una columna, reconstrueix la columna amb els valors de les posicions indicades. Aquesta operació retornaria una columna amb 4 cel·les que

contindrien les edats dels clients d'interès: 68, 70, 75 i 40. Finalment, l'operació de la mitjana (AVG) calcularia la mitjana dels valors de la columna, retornant el valor final: 63,25 anys.



La vectorització pot suportar tant representacions de dades per columnes com per files. Segons la bibliografia, el rendiment de les operacions que requereixen recorreguts seqüencials (com seria el cas de la projecció i la selecció, per exemple) és millor quan s'utilitza sobre representacions columnars (ja que aprofiten les característiques SIMD dels processadors actuals), mentre que les operacions que es basen en un accés aleatori a les dades (*hash-join* o agregació) tenen un rendiment millor perquè usen representacions per files.

En els magatzems de columnes és convenient retardar la materialització el més tard possible quan s'utilitza la vectorització per a permetre execucions en bloc. La vectorització suporta tant dades comprimides com dades no comprimides, però convé que les dades tinguin una grandària fixa per a poder facilitar la seva execució vectorial (en altres paraules, és convenient que els tipus de dades associades a les columnes siguin de longitud fixa i, en alguns casos , que el nombre de cel·les per a cada columna també sigui fix ja que sinó pot dificultar la materialització).

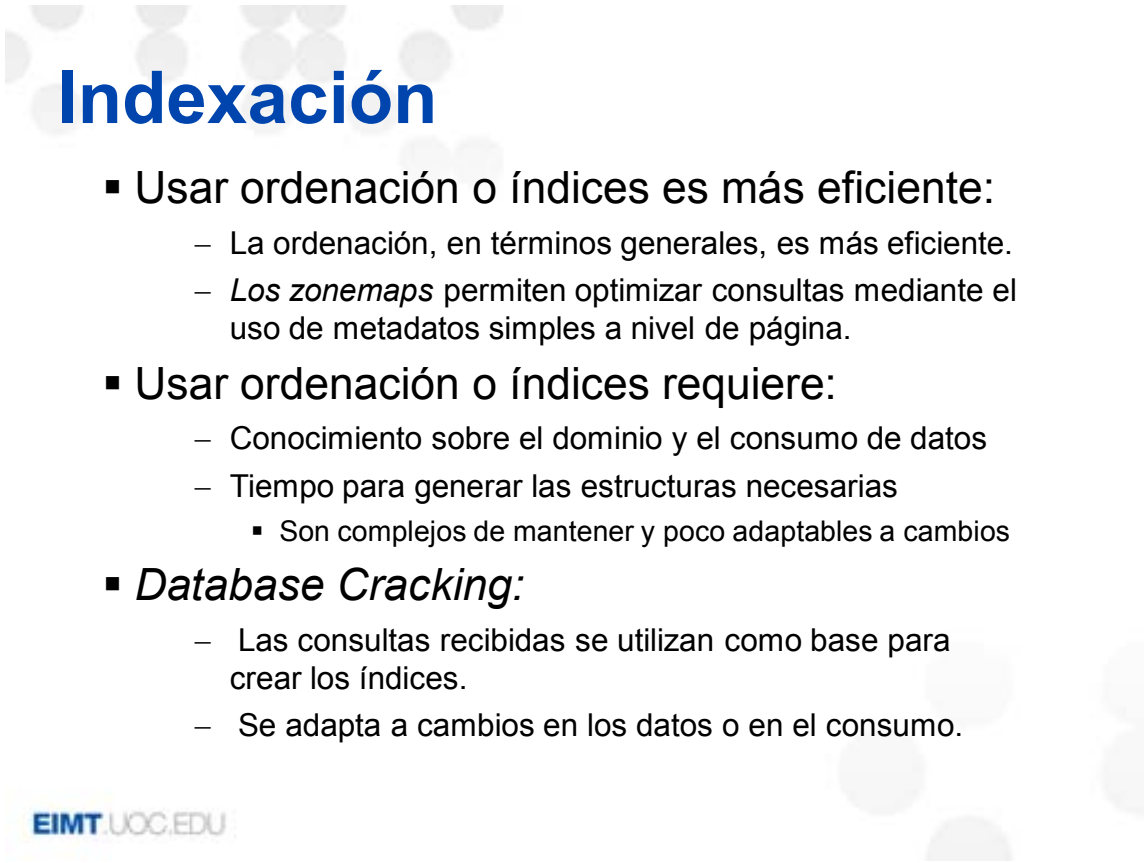
Alguns dels avantatges de la vectorització són:

- Es redueix la càrrega de la CPU: el nombre de crides realitzades sobre l'intèrpret d'una consulta es redueix en funció de la grandària del vector a utilitzar. Per exemple, si tenim vectors de 1.000 valors, podrem reduir en un factor de 1.000 el nombre d'operacions executades.

- Oportunitats d'optimització: algunes operacions de la BD es podran traduir directament en operacions SIMD del processador o combinacions de les mateixes.

- Execució adaptativa: la implementació vectorial de les operacions permet tenir informació més detallada del rendiment de la consulta i del cost de la CPU per a cada operació. Aquesta informació pot ser aprofitada per l'SGBD per a canviar el pla de la consulta de forma dinàmica, en funció del cost esperat d'una operació. Per exemple, suposem que hem de multiplicar certs valors d'una columna per 5. Suposem que el planificador, en temps d'execució (després d'aplicar una operació de selecció), comprova que el nombre de valors d'interès és molt baix (10 valors per exemple) mentre que el nombre de valors de la columna és molt elevat (hi ha molts valors irrellevants). En aquest cas, el planificador pot optar per executar l'operació de forma individual per a cada valor d'interès (cosa que semblaria convenient en aquest cas) en comptes d'executar la multiplicació de la columna en bloc (que probablement tindria un cost d'execució major).

Una vegada arribats a aquest punt, parlarem d'una de les últimes tècniques afegides als magatzems de columnes. El *database cracking* o indexació adaptativa.



El recorregut de dades en els magatzems de columnes és, per defecte, més eficient que els recorreguts en els magatzems de files. No obstant això, hi ha maneres de millorar encara més la seva eficiència explotant l'ordenació/indexació de forma adequada.

Hi ha treballs que demostren que la utilització d'índexs en els magatzems de columnes aconsegueixen millores en l'eficiència dels recorreguts d'una o dues ordres de magnitud. No obstant això, altres estudis demostren que, en termes generals, és més eficient l'accés sobre les columnes quan aquestes estan físicament ordenades, fent innecessari l'ús d'índexs com els utilitzats en els magatzems de files.

Per tant, una opció per a millorar el rendiment, és emmagatzemar les columnes de forma ordenada, en funció del valor d'una columna o conjunt de columnes (que reben el nom de clau d'ordenació). Addicionalment, una mateixa columna o grup de columnes (cada grup rep el nom de projecció) es poden emmagatzemar diverses vegades, d'acord amb diferents criteris d'ordenació (o sigui, en funció de diferents claus d'ordenació). Si recordeu, aquestes qüestions van ser tractades en profunditat en la presentació dedicada a les característiques dels magatzems de columnes.

Una altra forma de millorar el rendiment dels magatzems de columnes (que pot ser usada conjuntament amb l'ordenació física de les dades) és a través de *zonemaps*. Aquestes estructures auxiliars (que no deixen de ser índexs no densos –en anglès, *sparse indexes*–) emmagatzemen metadades bàsiques a nivell de pàgina, com per exemple (entre d'altres) el valor mínim i màxim emmagatzemat a la pàgina. Aquestes

metadades son útils per a poder descartar les pàgines que siguin irrellevants per a una consulta donada, sense haver d'accedir al seu contingut. Aquestes estructures també van ser discutides en la presentació dedicada a les característiques dels magatzems de columnes.

Un desavantatge de las estratègies basades en l'ordenació de les dades és que requereix un coneixement elevat del domini, perquè siguin efectives. Per exemple, és necessari conèixer la càrrega d'execució que haurà de suportar la BD, les dades que contindrà i com es distribuiran, quina serà la freqüència d'actualització i consulta de les dades, quins tipus de consultes s'esperen, etc. A més, una vegada es té aquesta informació, cal considerar els costos de creació i manteniment de les projeccions, i de la resta d'estructures auxiliars que s'hagin pogut crear (els *zonemaps* o índexs no densos).

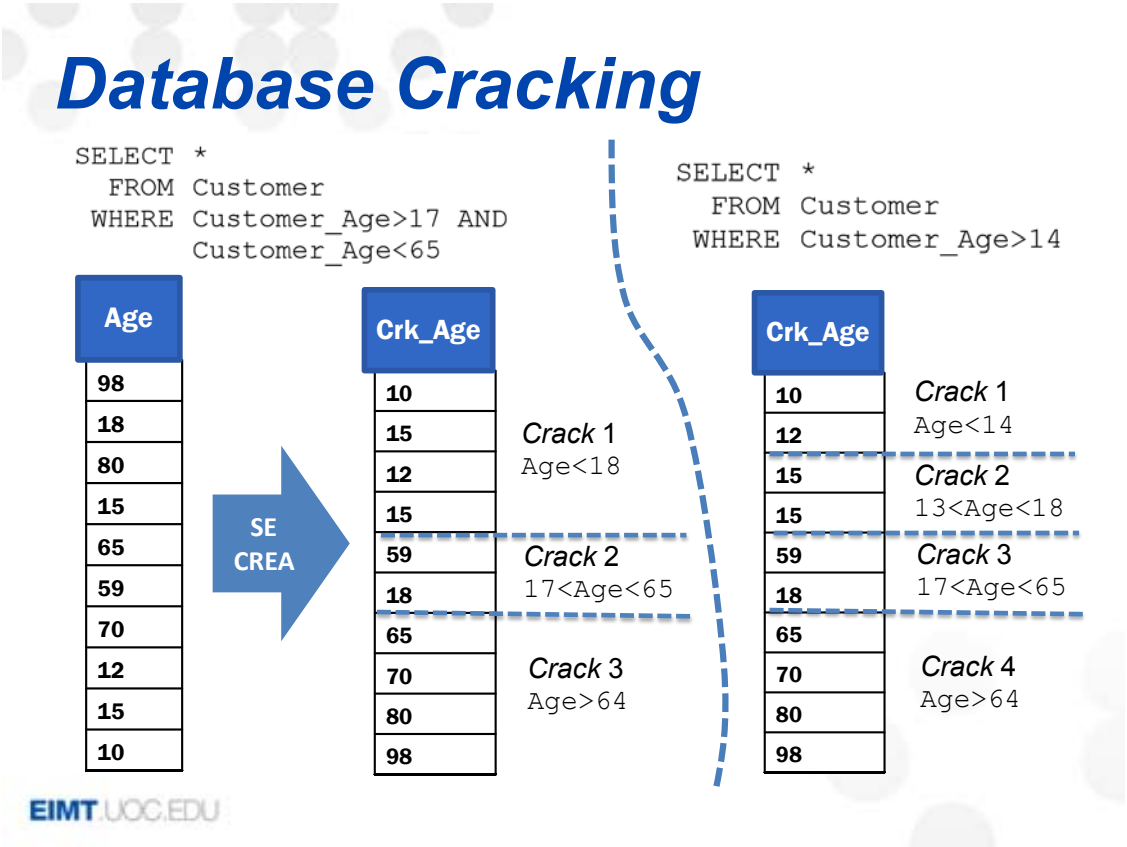
És important destacar que tot el que acabem d'explicar no està alineat amb certes necessitats actuals, en què el repte és proveir SGBD capaços de processar quantitats massives de dades de forma ràpida (encara que no sigui òptima), però suficientment flexibles per a adaptar-se a canvis en les dades o en els seus hàbits de consum. Per a donar solució a aquestes noves necessitats sorgeixen els índexs adaptatius, normalment anomenats *database cracking*.

El *database cracking* és una aproximació en què la definició i l'evolució dels índexs està dirigit pel tipus de consultes que es realitzen sobre les dades. Aquesta tècnica utilitza normalment les consultes per a inferir les necessitats de les consultes de les dades. Quan s'utilitza el *database cracking*, cada consulta rebuda s'interpreta com una pista del tipus de consultes que es rebran en el futur. Per a adaptar-se a futures consultes, agrupa físicament les dades per a oferir més eficiència a consultes semblants que es puguin efectuar en el futur. Per tant, l'índex es crea dinàmicament, evolucionant després de cada consulta (i mai després de les operacions d'inserció i d'actualització) i adaptant la manera en què s'agrupen les dades físicament en el dispositiu d'emmagatzematge extern.

El *database cracking* és una tècnica recent i molt popular en l'actualitat (almenys en entorns de recerca), però és més convenient que la indexació/ordenació? Bé, depèn. Si tenim un entorn en què sabem per endavant quines dades són interessants per als usuaris, quines dades s'utilitzaran en les consultes i podem assumir els costos d'indexar/ordenar les dades abans de rebre les consultes o entre una actualització i una consulta, l'ordenació/indexació és una estratègia millor. El *database cracking* és molt útil en entorns en què no hi ha coneixement sobre quines dades són interessants o no hi ha temps suficient per a mantenir l'ordre físic de les dades davant els canvis en les dades.

Alguns exemples de magatzems de columnes que es basen en l'ordenació de les dades (i en l'ús de *zonemaps*) són Vertica (i C-Store, el prototipus en què es basa), i Amazon

Redshift. Per la seva banda, MonetDB és, possiblement, el millor exemple de magatzem de columnes que usa el *database cracking*.



En aquesta transparència veurem com funciona el *cracking* en un magatzem de columnes. Per a veure-ho, utilitzarem la BD d'exemple utilitzada fins ara. En particular, utilitzarem el *cracking* per a indexar la columna *Age*.

La primera vegada que s'usa una consulta q per rang sobre un atribut (o columna) A, es crea una còpia de la columna A. Aquesta còpia es dirà la columna de *cracking* d'A i complirà la condició que els valors d'A que satisfan cada condició de q s'emmagatzemaran de forma consecutiva (és a dir, s'emmagatzemaran de forma conjunta). En consultes subsegüents sobre el mateix atribut, la seva columna de *cracking* s'anirà actualitzant d'acord amb la nova condició. Per tant, la columna s'anirà dividint en particions (o subdividint) en més i més *cracks* (o zones) a mesura que arribin més consultes.

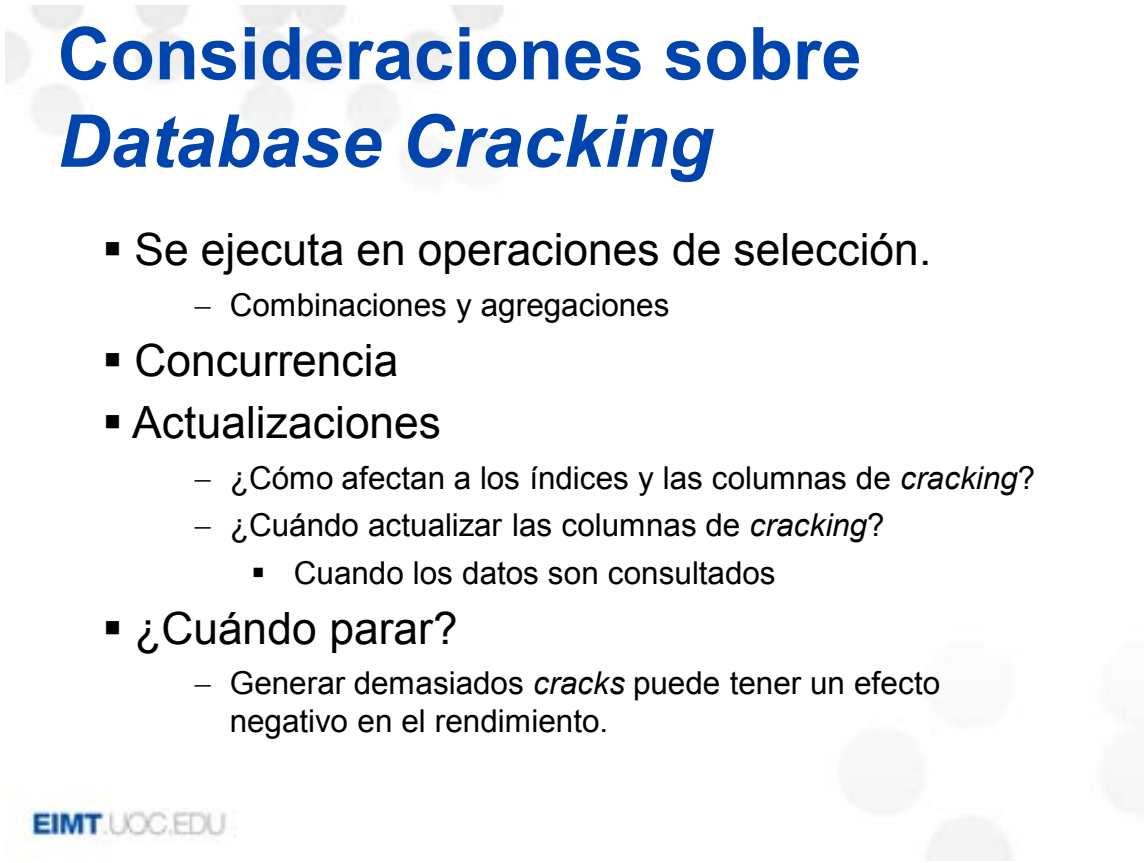
Per a tenir controlats els diferents *cracks*, els valors que contenen i com es distribueixen, s'utilitzen índexs. Aquests índexs poden ser de diferents tipus, però la idea és que permeten accedir a les diferents zones o *cracks* que s'hagin creat.

Normalment, el *cracking* es realitza en la memòria principal, permetent evitar la còpia de dades per a cada *crack*. En condicions normals, els *cracks* contenen apuntadors (els *rowid*) als valors associats en la columna original (que estarà emmagatzemada en el dispositiu d'emmagatzematge extern), permetent una major eficiència en el procés.

En l'exemple de la transparència, suposem que la primera consulta realitza un filtratge pels clients majors de 17 anys i menors de 65. En aquest cas, es construiria una columna de *cracking* que estaria dividida en tres *cracks* o zones. Una per als valors inferiors a 18 (primera condició de la consulta), una altra amb els valors superiors a 64 (segona condició de la consulta) i l'altra amb la resta de valors, justament els que són d'interès per a la consulta (majors de 17 anys i menors de 65 anys, o sigui clients entre 18 i 64 anys). A partir d'ara, qualsevol consulta que busqui valors amb rangs similars als utilitzats podria accedir directament a les zones rellevants. Fixeu-vos que els *cracks* no estan totalment ordenats ja que l'algorisme de *cracker* mou solament els valors necessaris per a complir la condició indicada (els valors que satisfan cada condició han d'estar situats consecutivament en un mateix grup, però no necessàriament ordenats). Òbviament, si s'executessin moltes consultes, podríem acabar tenint la columna de *cracking* totalment ordenada.

Suposem ara que l'SGBD rep una nova consulta que sol·licita els clients de més de 14 anys. En aquest cas l'SGBD, d'acord amb la nova consulta (i les condicions que aquesta imposa), partiria el primer *crack* de la columna de *cracking* en dos: un per a emmagatzemar els clients menors a 14 anys i l'altre on s'emmagatzemarien els clients amb edats entre 14 i 17 anys. La resta de *cracks* no es veuran modificats. Fixeu-vos que en aquest moment, en la columna de *cracking*, en *els cracks* 1 i 2 els valors ja estan completament ordenats.

Si en aquest moment es realitzés una nova consulta preguntant pels clients d'entre 18 i 64 anys (condició 17<*Age*<65), l'SGBD podria retornar la zona (o *crack*) 3 de la columna de *cracking*, i no hauria de reestructurar la columna, ja que tots els valors retornats en la consulta ja estan emmagatzemats de forma consecutiva en un mateix grup.



La tècnica de *database cracking* és relativament nova i s'usa bàsicament per a optimitzar les consultes que impliquen operacions de selecció (o sigui, consultes per rang). No obstant això, aquesta tecnologia també mostra potencial per a optimitzar les operacions d'agregació i combinació en els magatzems de columnes. Això permetria optimitzar encara més les operacions més costoses dels magatzems de columnes.

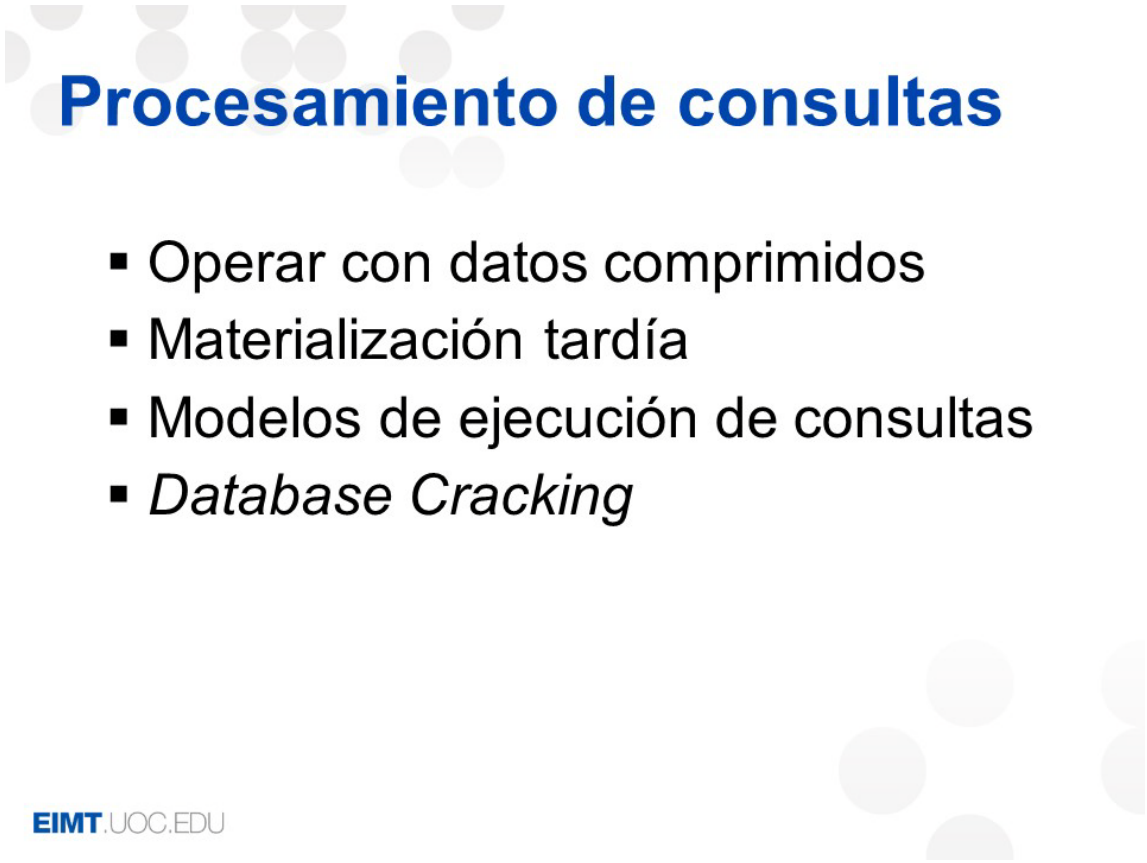
Un altre aspecte a considerar és l'accés concurrent a les dades. Aquí ens podem imaginar diferents situacions en què es pot incórrer en problemes. Per exemple, una situació problemàtica seria l'execució concurrent de dues consultes en què una d'aquestes estigués utilitzant una zona (o *crack*) de la columna de *cracking* per a obtenir els valors rellevants (i, per tant, estigués potencialment reestructurant la zona) mentre una altra consulta volgués obtenir dades de la mateixa zona. Hi ha diferents alternatives, des del típic bloqueig que no permet modificar una zona mentre un altre usuari l'estigui consumint, fins a aproximacions més optimistes que busquen paral·lelitzar les consultes sobre parts disjuntes de les dades per a permetre reestructurar la columna de *cracking* sense realitzar bloquejos.

Un altre dels aspectes a tenir en compte és com afecta l'actualització de les dades sobre els índexs i columnes de *cracking*. Aquí l'important és saber quan actualitzar les columnes de *cracking*. Una aproximació és mantenir les modificacions separades fins que el nombre de modificacions superi un cert llindar. Això pot portar la BD a un estat

en què els usuaris puguin tenir problemes per a obtenir les dades actualitzades. També es poden reestructurar els índexs i columnes de *cracking* no quan s'afegeixin noves dades (o es modifiquin les existents), sinó quan es consultin aquestes dades. Una altra opció seria afegir les noves dades al final de les columnes de *cracking* (quan reben insercions o modificacions) i «oblidar» els índexs de *cracking*. Això obligaria a crear de nou els índexs i els diferents *cracks* a partir de zero en funció de les noves consultes, tornant a començar el procés d'indexació. Alguns experiments mostren que usant aquesta aproximació els índexs de *cracking* aconsegueixen ràpidament un bon rendiment després de pocs centenars de consultes.

D'altra banda, un altre aspecte rellevant és saber quan cal parar de *fer cracking* sobre una columna. Sembla raonable pensar que si continuem dividint en particions una columna indefinidament, arribarem a un punt en què el nombre *de cracks* serà molt elevat i la gestió dels índexs i de la columna de *craking* serà massa costós. L'opció més adoptada per a tractar aquesta situació consisteix en definir el nombre d'elements mínims que ha de tenir cada *crack*. Així, si potencialment un *crack* té molt pocs valors (menys que el mínim que s'hagi establert), no es crearà.

Si algú vol estudiar més a fons els temes comentats en aquesta transparència, aconsellem consultar la tesi d'Stratos Idreos, es tracta de l'última referència que trobareu al final d'aquest document.



I fins aquí aquests materials dedicats al processament de consultes en magatzems de columnes, en què hem vist les tècniques principals que permeten optimitzar l'execució de les consultes. En concret, hem explicat com operar amb dades comprimides, quan usar la materialització tardana, els models d'execució que un SGBD (i en particular un magatzem de columnes) pot utilitzar per a resoldre les consultes que es formulen sobre la BD i de les tècniques d'indexació més recents que s'han proposat per als magatzems de consultes (*database cracking*).

Esperem que hàgiu trobat els materials amens, útils i el tema interessant. Alguns dels conceptes tractats són altament especialitzats i d'una certa complexitat. És més, parcialment, estan en recerca. L'objectiu principal és que hàgiu comprès les idees generals subjacents de cadascuna de les tècniques que hem explicat.

A continuació, trobareu un conjunt de referències rellevants sobre el tema, per si us interessa aprofundir més en aquestes qüestions.

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.

Que tingueu un bon dia!