Big Data Management.
Sesión 02. Fundamentos de bases de datos.

Autor: Ziani El Ali, Adil.

adilziani.wordpress.com

15 de abril de 2020

Índice

Sobre *Data File* e índices

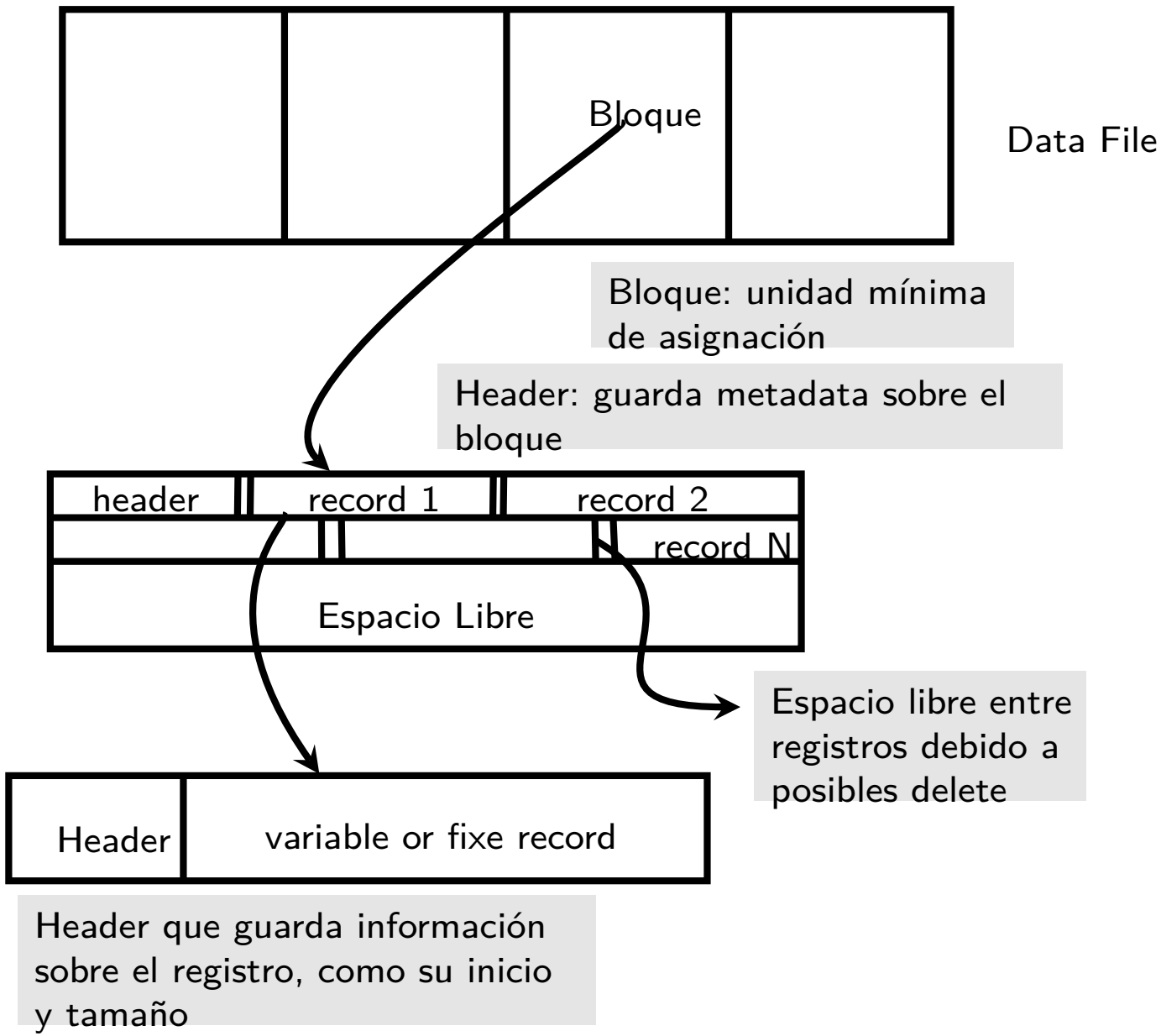
Vimos en la sesión precedente que los objetivos de una base de datos es guardar datos y acceder a ellos de la mejor manera posible. Entonces nos preguntamos:

Cómo una base de datos organiza internamente los datos?
Qué técnicas habilita para un acceso más óptimo a los datos?

Los datos en un sistema de almacenamiento acabarán en disco, aquí tenemos una estructura de ficheros, con lo cual una base de datos acabará almacenando los datos en ficheros (**Data Files**), pero de una determinada manera y con un determinado formato que permite al DBMS interpretar mejor los datos y "construir la tabla" optimizando ciertos accesos.

Las razones por las que se usa *Data Files*, en lugar de ficheros planos, son:

- **Eficiencia de almacenamiento**, los data files están organizados con la idea minimizar la sobrecarga de almacenamiento.
- **Eficiencia en accesos**, los registros se pueden ubicar en el menor tiempo posible.
- **Eficiencia en las actualizaciones**, si un registro cambia, que aquello suponga los menores cambios en disco.



Estructura *Heap File*

Las bases de datos también responden a la segunda pregunta creando **index files**, ficheros que guardan metadata sobre los registros, información que permite implementar índices sobre la tabla.

Un índice es una estructura que organiza registros de datos en el disco de una manera que facilita operaciones de recuperación eficientes de los datos (índices Cluster), o también tener un catálogo de cómo están organizados los registros y su localización en disco para un acceso más eficiente.

Veremos aquí los índices B+ Tree, B+ Tree Cluster e índice Hash.

Podemos completar información sobre Data Files y Data index en la primera parte del libro (Petrov, 2019)

B+ Tree

- Una estructura de índice implementada por diversas bases de datos.
- Responde satisfactoriamente a diversos casos de uso.

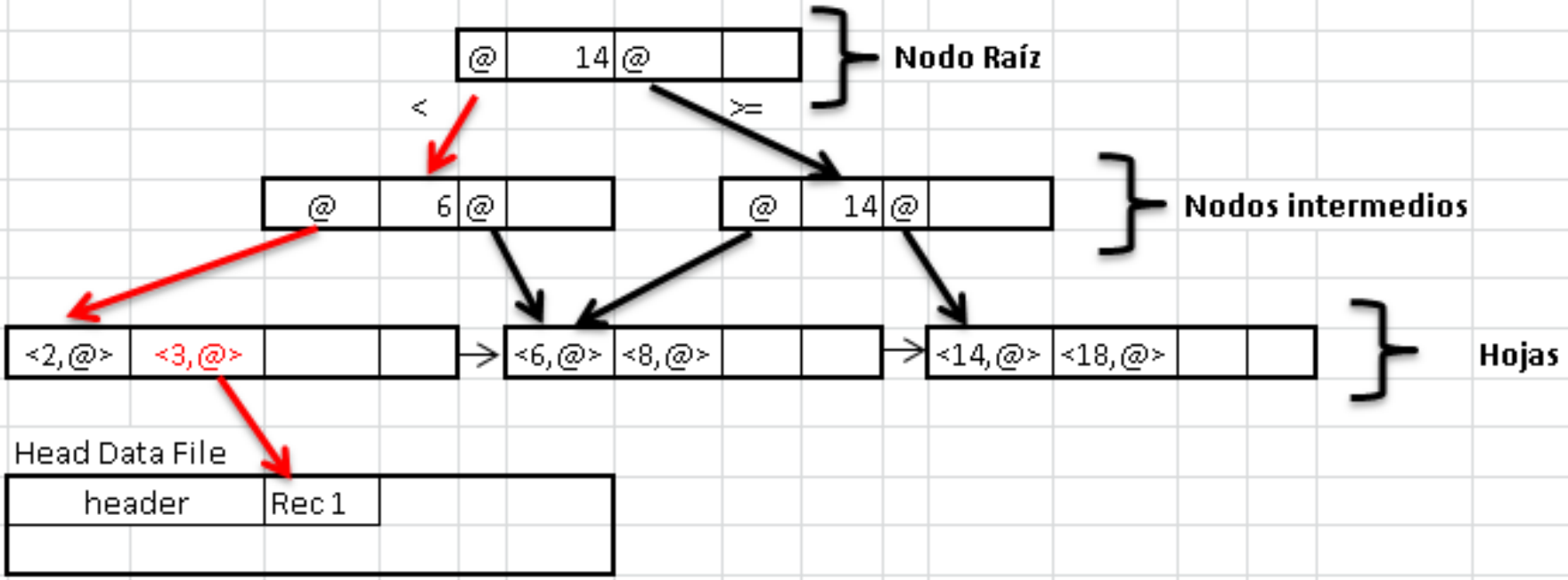
Un índice B-tree se organiza en árbol balanceado, es decir, todas las hojas están al mismo nivel.

Definición: se dice que que el árbol tiene orden D si en cada hoja caben hasta 2D punteros.

Generalmente, una hoja corresponde a un bloque en disco, por lo que ofrece gran espacio para almacenar valores y punteros a registros.

Perso	Nombr	Edad
P1	Andrea	14
P2	María	8
P3	Lucia	3
P4	Raúl	18
P5	José	2
P6	Lucas	6

```
SELECT * FROM Tabla
  WHERE Edad = 3
```



Indice de orden 2, sobre el atributo Edad. Carga del 75 %

Algoritmo B+ Tree

1. Crear un fichero con las claves y punteros
2. Ordenar el fichero por las claves
3. Construir las hojas <clave,@puntero> (respetando el orden y la carga del árbol)
4. Construir los nodos intermedios (respetando la carga del árbol). Recuerde que los nodos intermedios tienen diferente estructura que los nodos hoja, no contienen punteros a disco si no a nodos hoja (|@,valor,@,valor,...|)
5. Guardar el Data File Index en disco.

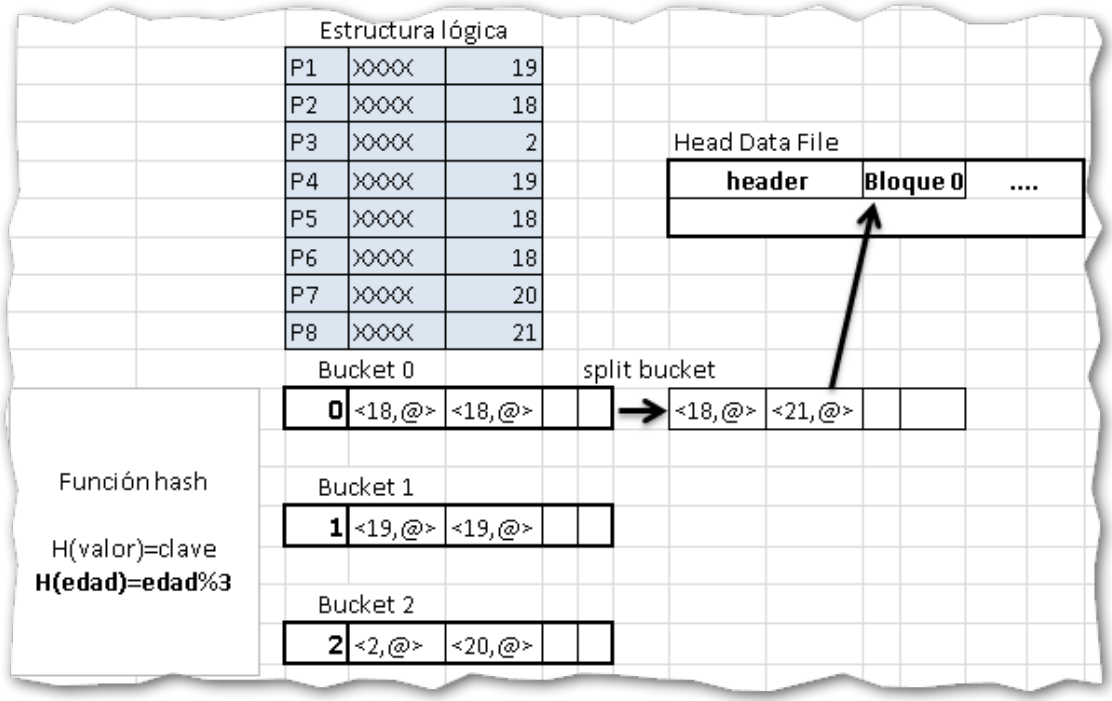
Consideraciones:

- La carga del 75 % permite dejar espacio para futuros aumentos de la tabla, y por tanto indexar más registros.
- Ser balanceado, todas las hojas al mismo nivel, permite acceder a los datos de manera uniforme.
- Entre las hojas existen punteros, lo que permite ir de hoja en hoja sin pasar por el nodo padre.
- En caso de Updates, Delete e Insert, el índice puede que necesite reorganizarse, bien suprimiendo hojas o haces split de nodos intermedios para añadir más hojas.
- El índice **B+Tree Cluster** fuerza un orden por las claves del índice en el guardado de la tabla. Eficaz para búsquedas por rangos, pero puede penalizar inserciones masivas.
- En la práctica, cuando se trata de inserciones masivas o por lo general modificaciones masivas en tabla, se desactiva el índice y posteriormente se regenera de nuevo.

Indice Hash

Los índices hash, son un tipo de índice que organiza el *data file index* en formato *Hash File*, relacionando así las claves de búsqueda con sus punteros asociados (punteros a registros).

Los *hash file* almacenan datos con la misma clave en bloques juntos, aplicando dicha estructura a índices hash, tenemos que para cada clave de búsqueda, se almacenan punteros en los mismos bloques (**buckets**), y mediante una función (**función hash**)aplicada sobre el valor a buscar, obtenemos su clave y accedemos al bucket/s correspondientes, con una lectura secuencial de dicho bucket/s obtenemos el puntero/s respuesta a nuestra consulta y entonces se accede a disco para recuperar la información.



Consideraciones:

- La función hash ideal, que permite el mapeo ente valores y claves, sería una función de distribución uniforme, es decir, a cada bucket asigna mismo número de valores dentro de todo el conjunto de valores posibles.
- El índice hash es un índice secundario.
- Normalmente las funciones hash suelen ser del tipo $H(x) = h(x)\,\%N$ donde N es el número de buckets.
- Los índices hash son más eficientes en búsquedas con igualdad.
- Como en el ejemplo anterior, se deja espacio libre en los buckets para futuras inserciones en la tabla.

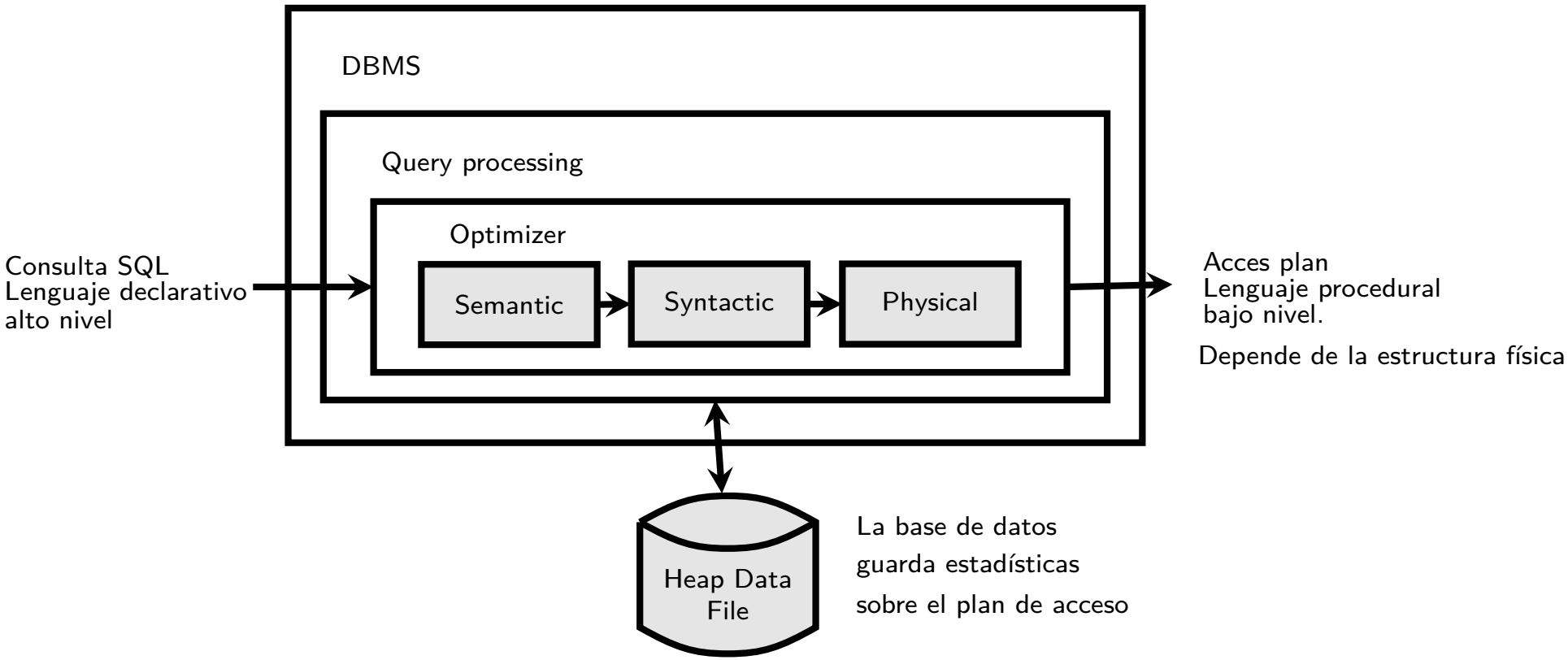
Consideraciones generales.

- **Considerar definir un índice:**
  - **B-tree**
    - Hay una condición muy selectiva en la consulta.
    - El predicado es un rango.
    - El orden es relevante. GROUP BY, ORDER BY.
  - Hash
    - Hay una condición muy selectiva en la consulta, con igualdad.
    - La table es muy grande.
    - La tabla no es muy volátil, puede provocar muchos splits en los buckets.
    - Es necesario usar una función hash uniforme
  - Cluster
    - Mismos beneficios que B-tree, condiciones selectivas, rangos.
    - Para lecturas de rangos, aprovecha beneficios lectura secuencial y cache.
    - La tabla no es muy volátil, muchos cambios obligan reordenaciones en disco.
- **No considerar índices**
  - La tabla es pequeña, con pocos bloques en disco
  - La clave del índice aparece con frecuencia en la consulta, en funciones de agregación
  - Las consultas más relevantes sobre la tabla no son muy selectivas, consideran recuperar gran parte de los bloques.
  - Por heurística, en caso de que las consultas requieren retornar más del 10 % de los registros (caso SQL), 3 % caso NoSQL, es preferible un acceso secuencial frente a acceso por índice.

Query optimiser

Las bases de datos relacionales llevan implementado un potente optimizador, con diferentes niveles para transformar nuestra consulta en un plan de acceso óptimo. Veremos técnicas usuales de optimización que nos sirvan para nuestros desarrollos en Big Data, pues como adelanto, las bases de datos NoSQL por lo general no disponen de un fuerte optimizador como los que encontramos bases SQL.

Nuestros desarrollos,
por tanto, han de realizarse pensando en optimizar cada transformación de los datos.



Semantic Optimizer

Una fase de optimización que consiste en transformar la consulta SQL en otra consulta equivalente de menor coste considerando:

- Integridad de las restricciones.
- Lógica

```
CREATE TABLE students
id CHAR(8) PRIMARY KEY,
mark FLOAT CHECK ( mark > 3);


SELECT * FROM students WHERE mark < 2;
-- No cumple la integridad, return null

SELECT * FROM students WHERE mark <6 AND mark < 7;
-- Equivale a SELECT * FROM students WHERE mark < 7;
```

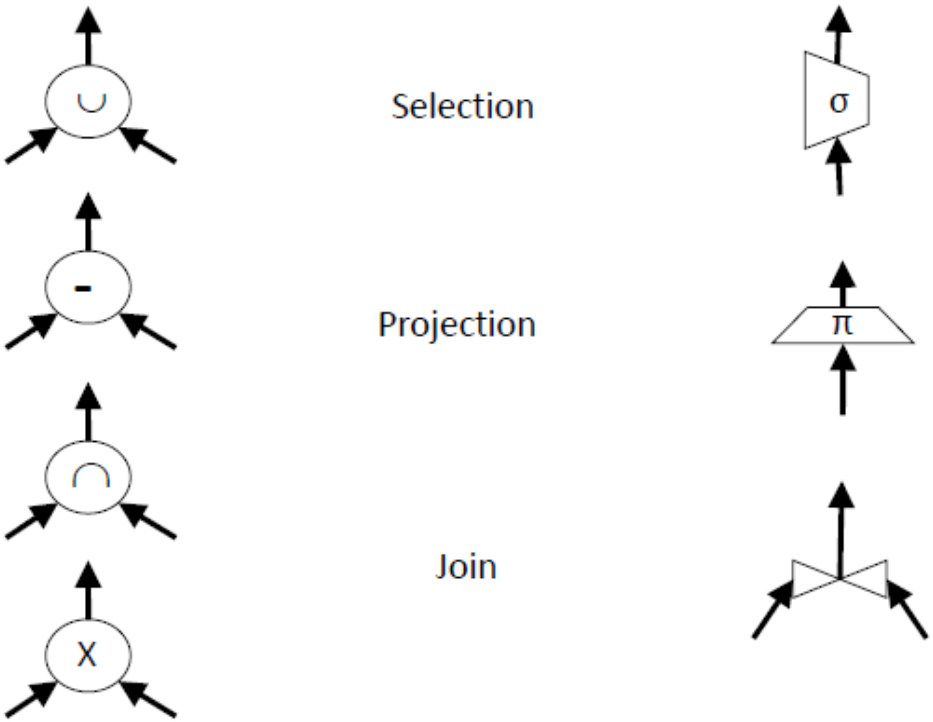
Syntactic Optimiser

Definición: llamamos **árbol sintáctico** a un árbol lógico que describe el acces plan de una consulta

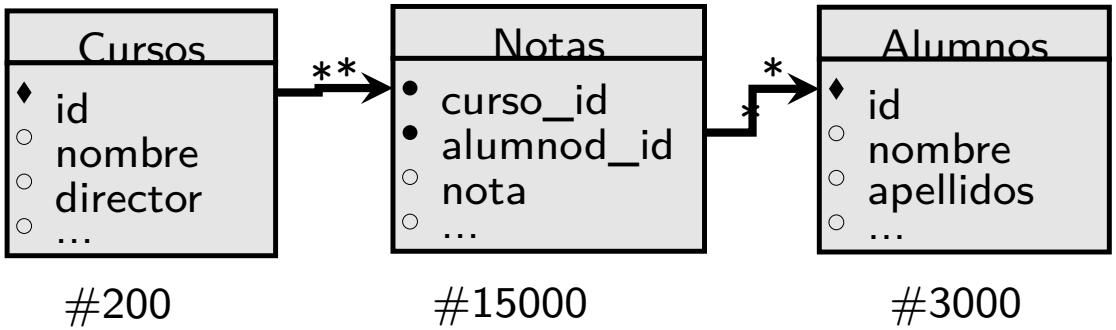
- Nodos
  - Nodos internos describen las operaciones lógicas
  - Nodos hoja representan las tablas involucradas en la consulta
  - Nodo raíz representa el resultado de la consulta
- Aristas
  - Representan acceso directo de datos entre nodos, también se usan para describir cardinalidad en cada trasformación.

Entonces, la pase **Syntactic Optimiser** consiste en transformar la consulta SQL en una secuencia de operaciones algebraica representadas en forma de árbol sintáctico, y buscar aquel de menor coste teniendo en cuenta heucarísticas y estadísticas que la base de datos guarda.

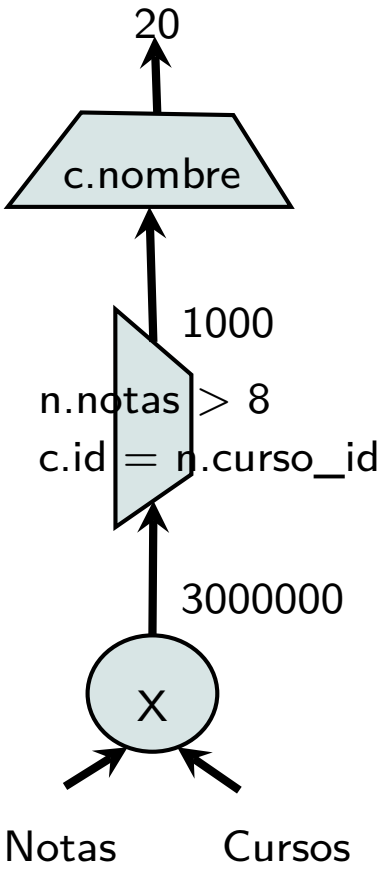
Relational Operations

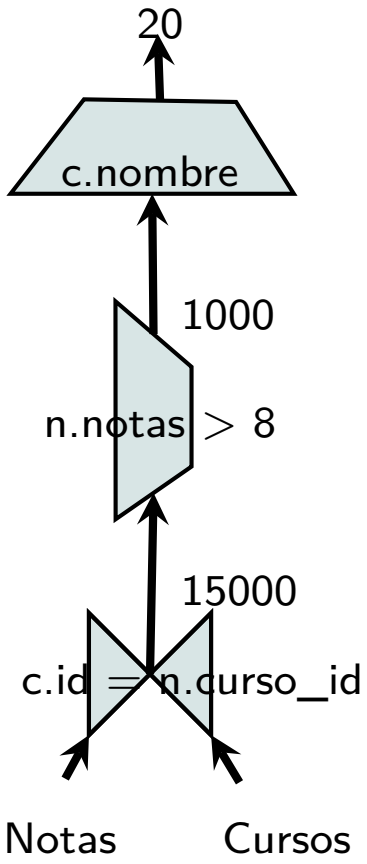


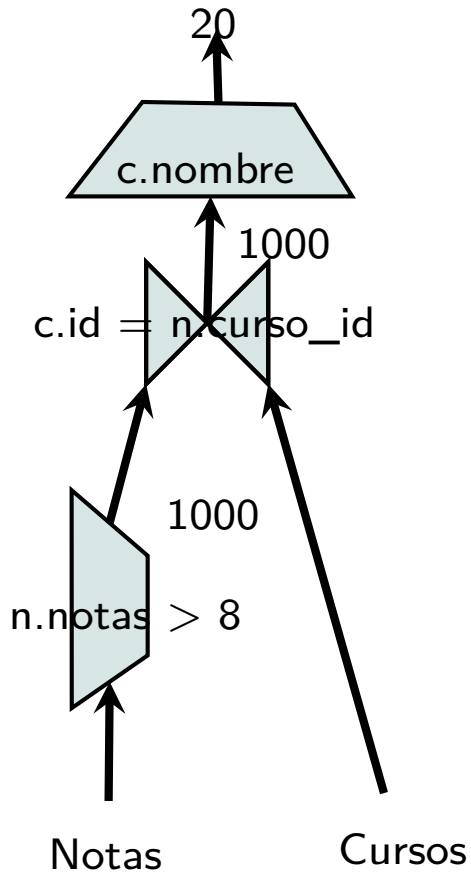
By Oscar Romero




```
SELECT DISTINCT c.nombre
FROM Cursos c, Notas n
WHERE c.id = n.curso_id
AND n.nota > 8
```







Transformando el árbol sintáctico

Para construir un árbol sintáctico óptimo, el objetivo es reducir en lo posible el cardinal de los nodos intermedios.

Pasos:

1. Tener un primer árbol que responda a la consulta
2. Dividir las selecciones en selecciones simples
3. Bajar las selecciones tanto como sea posible
4. Agrupas selecciones consecutivas (sobre la misma tabla)
5. Bajas las proyecciones tanto como sea posible
6. Agrupas proyecciones consecutivas (sobre la misma tabla)

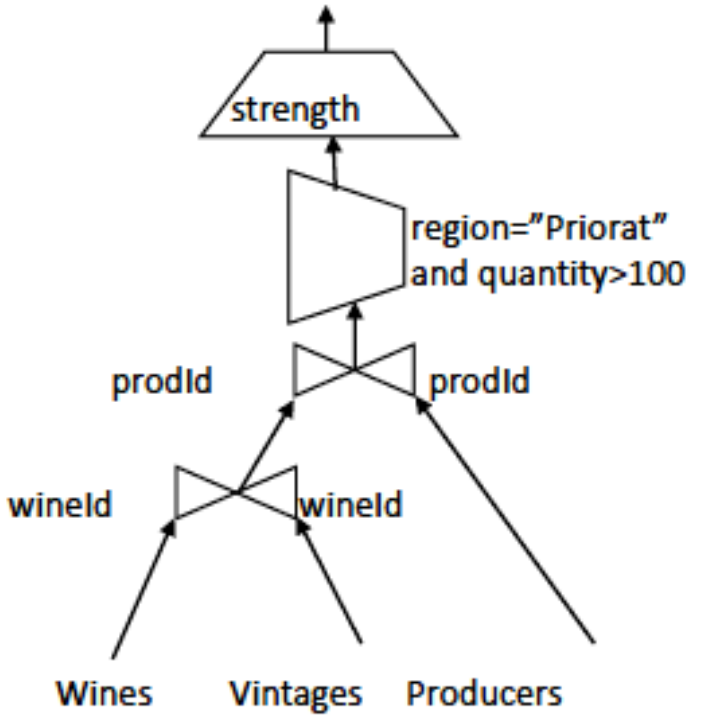
En definitiva, sobre cada tabla aplicar los filtros y proyecciones antes de posibles joins y/o agregaciones, así los datos que se cruzan entre tablas y que se procesan, son menores. Se obtiene así nodos intermedios de menor cardinalidad y por ende operaciones menos costosas.

Wines(wineId, wineName, strength)

Vintages(wineId, prodId, quantity)

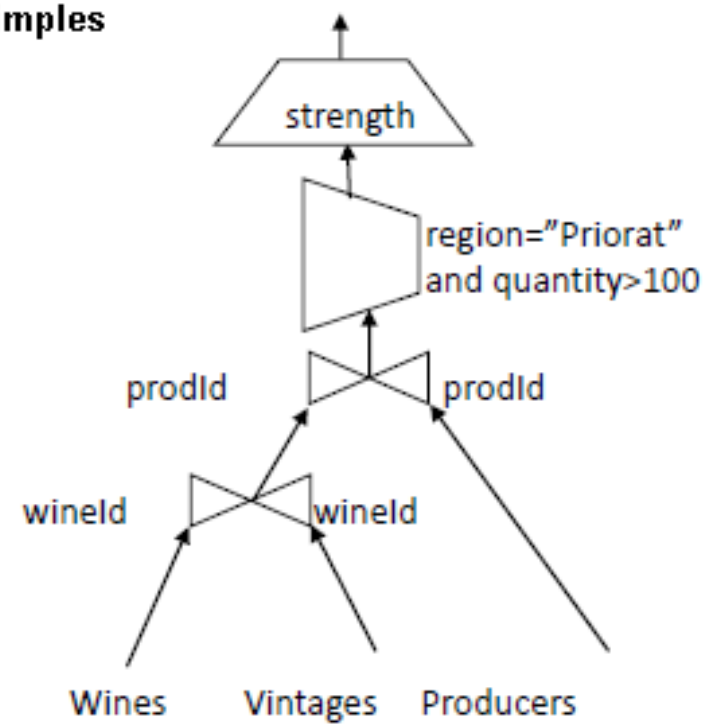
Producers(prodId, prodName, region)

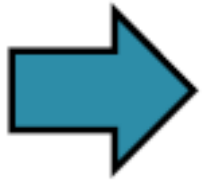
```
SELECT DISTINCT w.strength
FROM wines w, producers p, vintages v
WHERE v.wineId=w.wineId
        AND p.prodId=v.prodId
        AND p.region="Priorat"
        AND v.quantity>100;
```

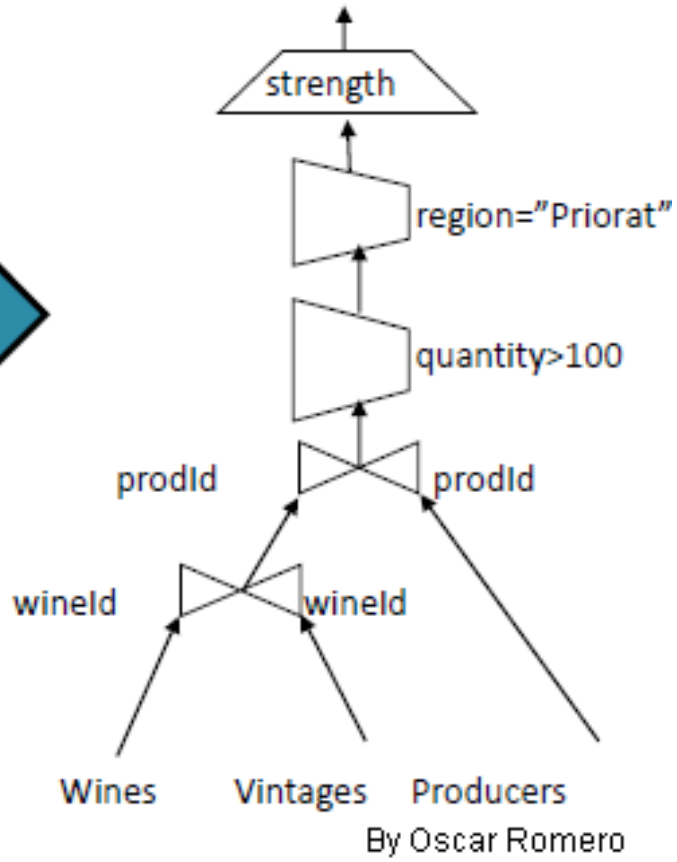


By Oscar Romero

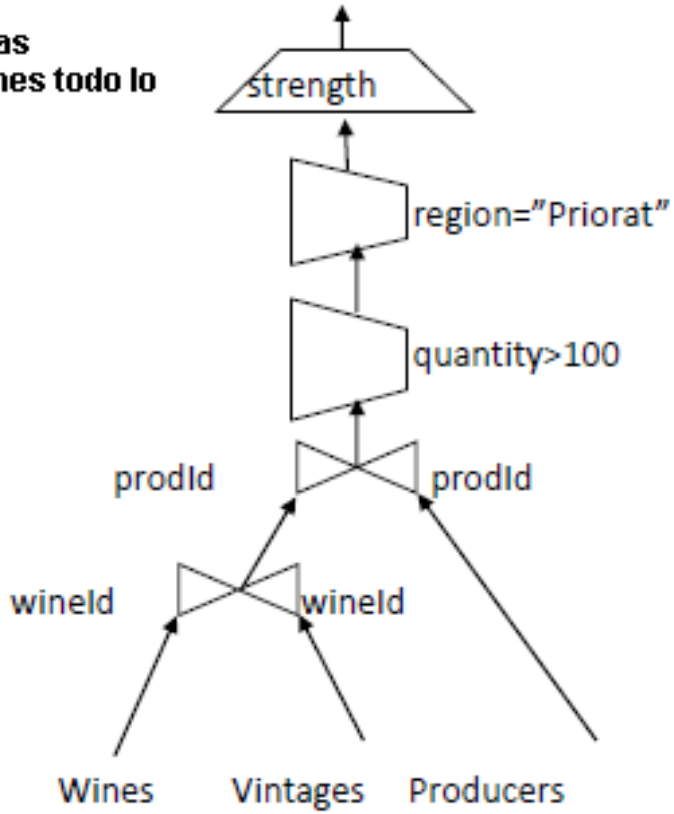
2. Dividir las selecciones en selecciones simples

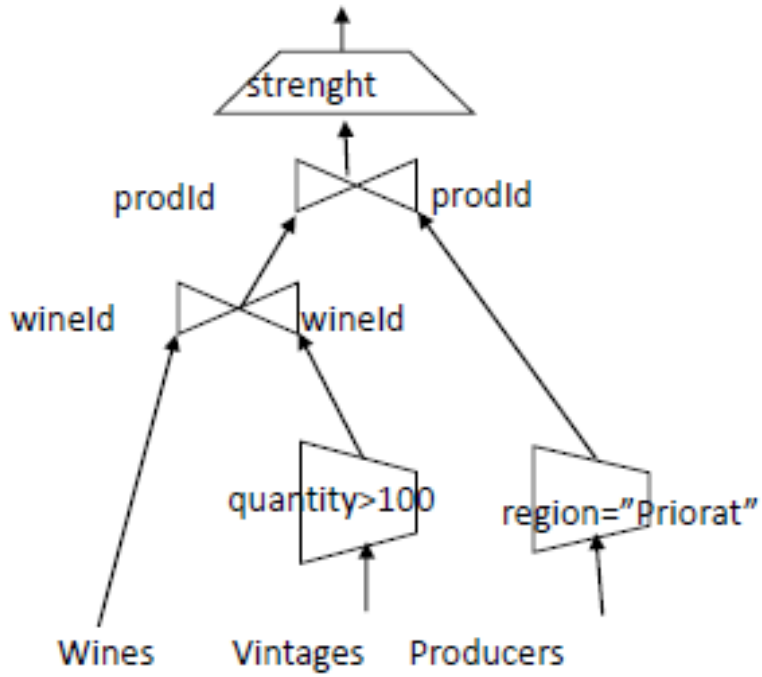






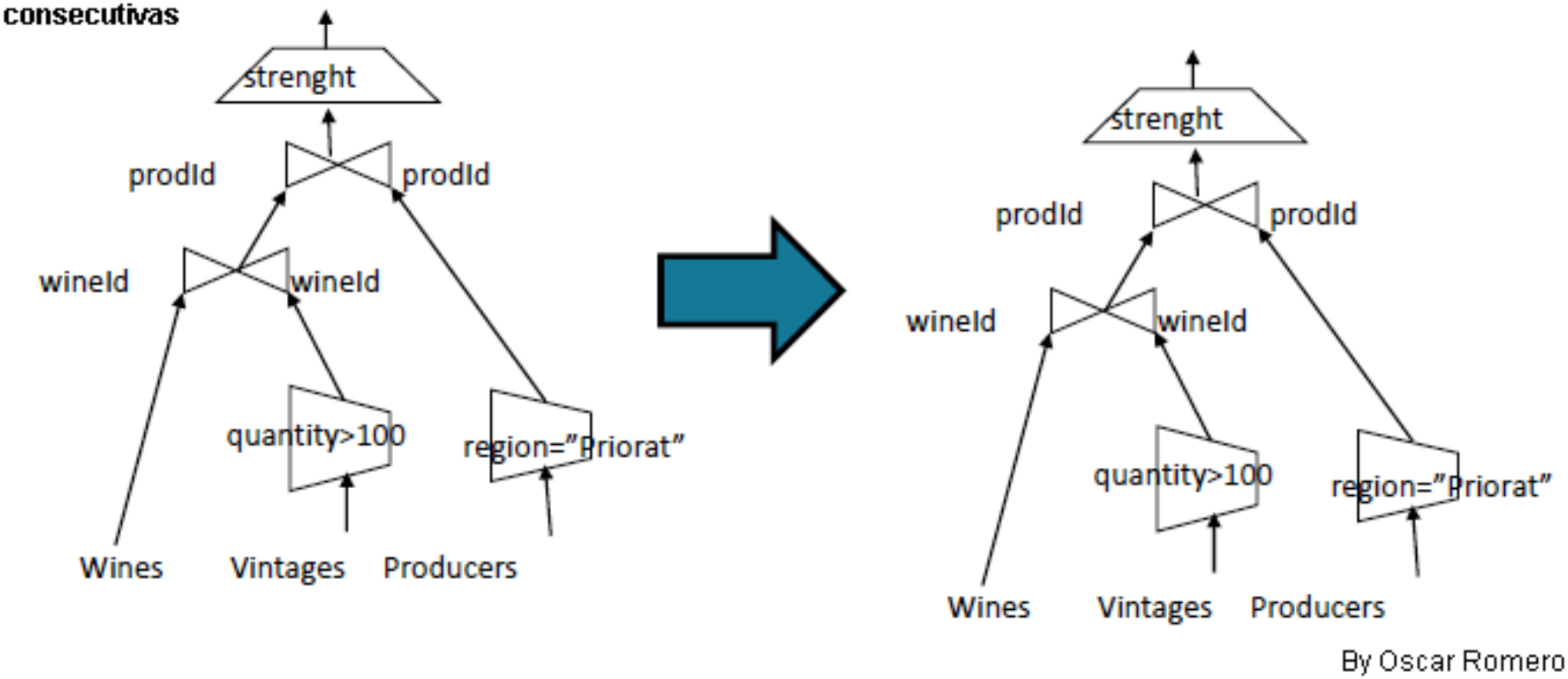
3. Bajar las selecciones todo lo posible



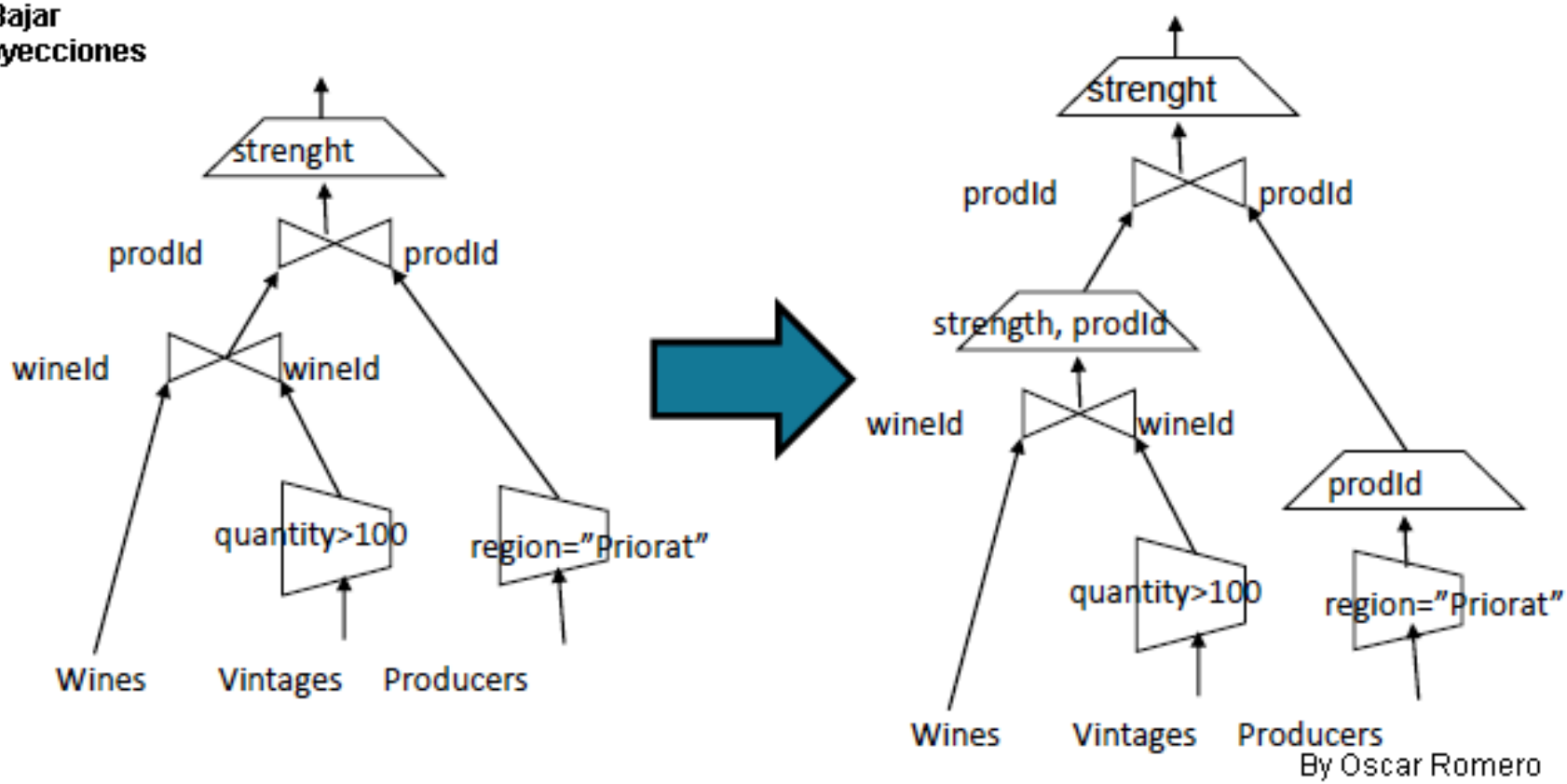


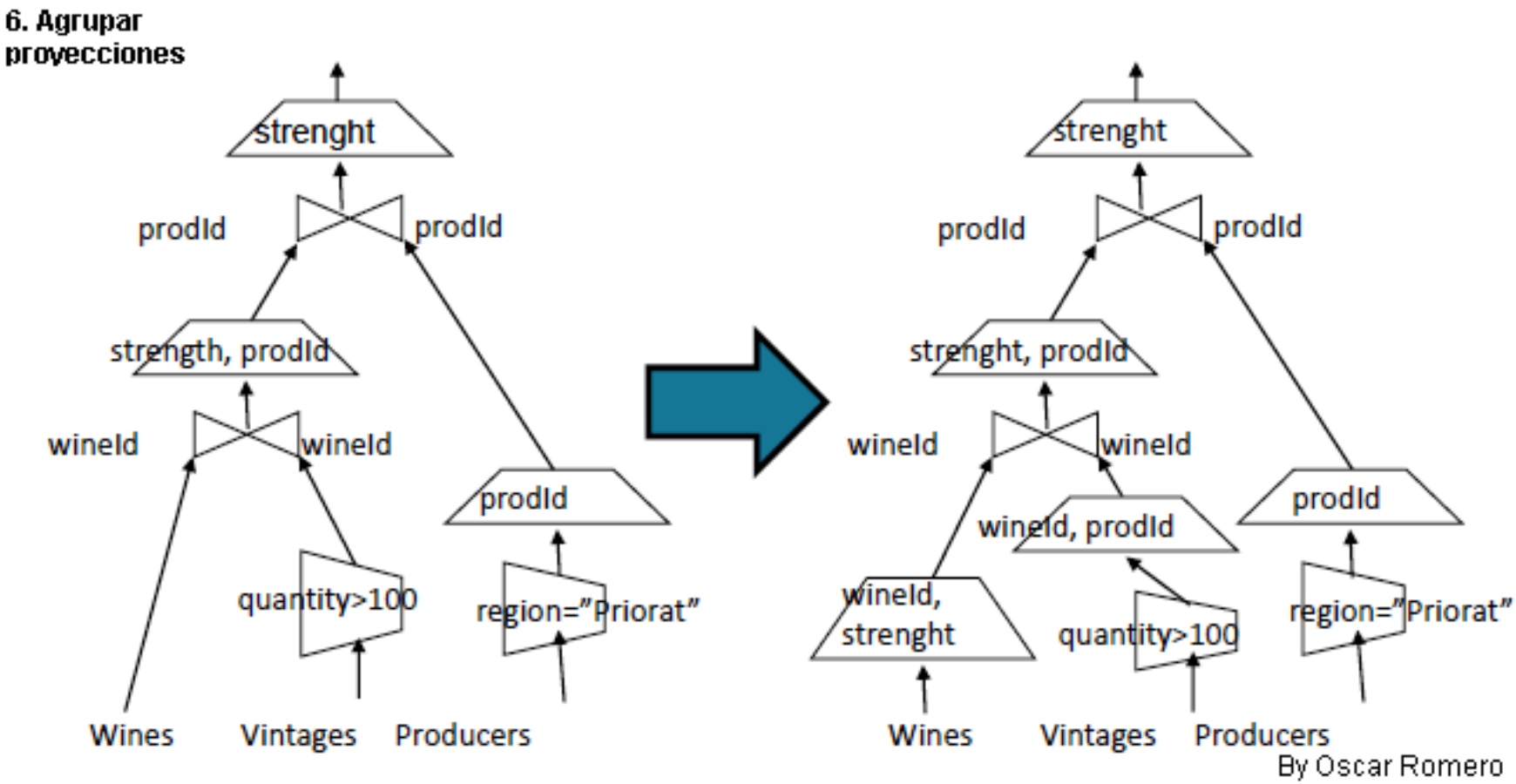
By Oscar Romero

4.Agrupar selecciones consecutivas



5. Bajar proyecciones





Bajar y agrupar selecciones y proyecciones tanto como sea posible

Physical Optimisation

Consiste en generar un plan de ejecución para el mejor árbol sintáctico de la fase anterior. Para ello se tiene en consideración:

- Estructura física de los datos, cómo están guardados en disco.
- Alternativas en las operaciones algebraicas.
- Acces Path, cómo recuperar los registros. Indices, vistas...
- Algoritmos, existen diferentes algoritmos para las operaciones algebraicas

En esta fase se generan diferentes planes de acceso para el árbol sintáctico, dichas opciones se representan de forma de árboles lógicos (**process tree**):

- Nodos
  - Raiz, representa el resultado final
  - Nodos intermedios, representan tablas intermedias, vistas no materializadas
  - hojas, representan tablas, vistas materializadas, índices
- Aristas
  - Sirven para representar el flujo y la dirección del acces plan.

Para escoger un **process tree** dentro de las opciones:

1. Generar process tree
2. Estimar cardinalidad de nodos intermedios.
3. Estimar coste de cada algoritmo para operar las operaciones algebraicas.(Las bases de datos suelen usar estadísticas que almacenan costes a modo de heurísticas para futuras queries)
4. Considerar el mejor process tree y crear el acces plan final.

Las bases de datos NoSQL, como veremos en las próximas sesiones, no disponen de un fuerte plan de optimización general como las bases relacionales, pero sí que son más especializadas para determinadas consultas por lo que a la tarea de escoger una base u otro requiere conocer bien nuestro negocio y necesidades.

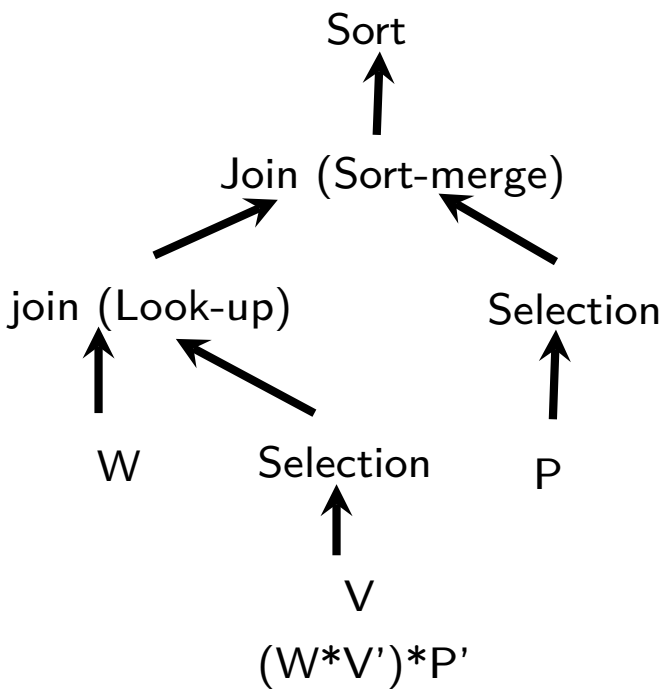
1. Generar *process tree* alternativos

A tener en consideración:

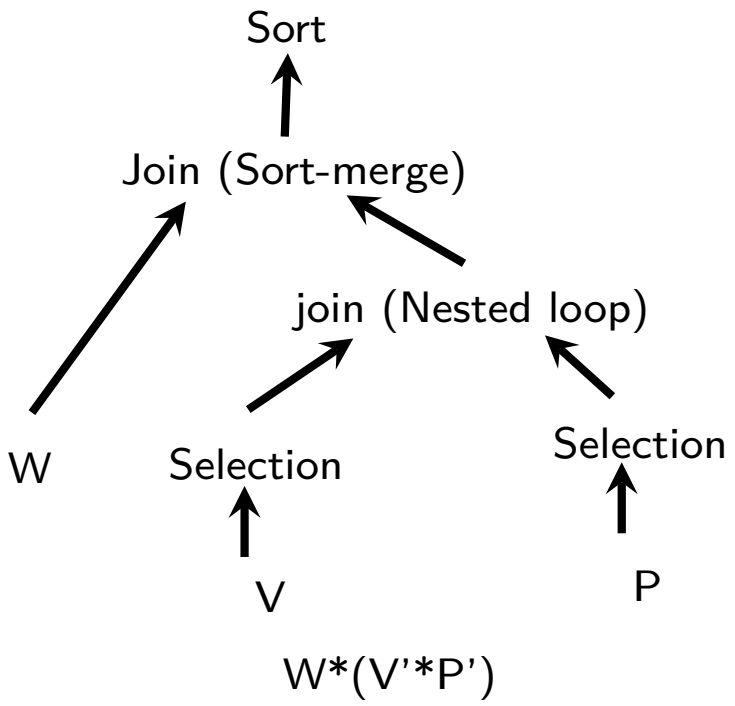
1. Orden de los joins
2. Considerar los acces path disponibles
3. Considerar los diferentes algoritmos disponibles para cada operación algebraica[1]

Con esto se tienen diferentes *process tree*, ejemplo:

PT1



PT2



[1] En el libro (Dwyer, 2016) capítulo 6, podéis ampliar información sobre diversos algoritmos.

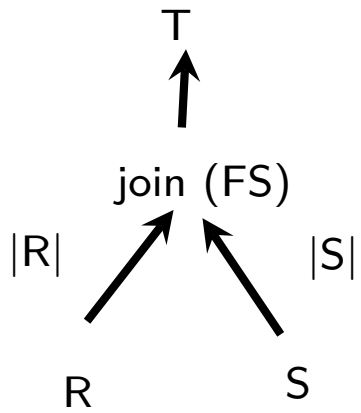
1. Generar *process tree* alternativos: acces path

	No índice	**B+**	**Hash**	**Clustered**
Todos los registros	Scan table			
Un registro		Considerar índice Scan bloque	Considerar índice, aplicar función, recuperar bucket, scan bloque	Considerar índice scan bloque
Unos cuantos registros		Considerar índice Scan bloque/s		Considerar índice scan bloques consecutivos
Join	Nested Loop Hash join	Row Nested Loops	Row Nested Loops	Row Nested Loops Sort-Merge

2. Estimar cardinalidad de nodos intermedios

Definición. Llamaremos factor de selección (**FS**) a la razón entre número total de registros de una tabla(materializada o no) y los devueltos en una consulta sobre dicha tabla. Es decir $FS = \frac{|\text{consulta}|}{|\text{total}|}$, Ejemplo: SELECT * FROM T tendría un FS = 1

- Estimar cardinalidad de una selección
  $|section(T)| = FS * |T|$
- Estimar cardinalidad join
  $|join(R,S)| = FS * |R| * |S|$
- Estimar cardianlidad union
  - Con repetición
    $|union(R,S)| = |R| + |S|$
  - Sin repetición
    $|union(R,S)| = |R| + |S| - |join(R,S)|$
- Estimar diferencia (anti-join)
  $|dif(R,S)| = |R| - |join(R,S)|$
- Se estiman las cardinalidades desde las hojas hacia la raiz
- Se usan estadísticas para estimar cardinalidades: Número de bloques, media tamaño registros, cantidad de atributos, valores máximos y mínimos,etc.



3. Estimar coste de algoritmos

Definición: llamaremos coste del modelo de un process tree a la suma de costes de cada operación algebraica.

El coste de cada operación es a su vez la suma del coste de resolución de cada operación y el coste escribir el resultado.
Los factores principales que intervienen en el coste son:

- Uso de la CPU
- Tiempo acceso a la memoria (a diferencia de bases relacionales, las bases NoSQL intentan tirar de memoria lo máximo posible en sus operaciones)
- Tiempo acceso a disco

Ejemplo: Table scan:

- Sin índice
  $B * D$
- B+
  $\lceil |T|/u \rceil * D + |T| * D$
- Clustered
  $\lceil 1.5B \rceil * D$
- Hash
  $\lceil 1.25(|T|/2d) \rceil * D + |T| * D$

Donde:

- B: número total de bloques en disco donde se almacena la tabla
- D: Tiempo medio de lectura o escritura de un bloque
- $|T|$: número de registros en la tabla
- d: orden del árbol
- u = load * 2d
- Se usan aproximaciones en los coeficientes, no se consideran los beneficios de un sequential acces ni aprovecho de cache.

En resumen. Tuning Tasks

- Un índice "not clustered" nunca empeora una consulta, en todo caso es ignorado, pero cuidado con updates e inserts masivos, el índice los puede ralentizar.
- Para tablas pequeñas, usar índices no aporta mejoras. Las bases de datos lo ignorarán en sus consultas, optarían por full acces.
- Para tablas grandes, un índice mejoraría ciertas consultas, las que filtran por los campos indexados
- Condiciones de filtrado con igualdad sugieren índice Hash, descartar B+
- Condiciones de filtrado con rangos sugieren B+, Clustered, descartar Hash
- Tablas con consultas de rango sugieren índice cluster. Solo se puede un único índice Cluster

En la próxima sesión

Sesion 03: **Hadoop Distributed File System (HDFS).**

- Qué es HDFS
- Arquitectura HDFS
- File Formats
- ...

referencias

Dwyer, Barry (2016). *Systems Analysis and Synthesis*. Morgan Kaufmann is an imprint of Elsevier 225 Wyman Street, Waltham, MA 02451 USA: Elsevier Inc. ISBN: 978-0-12-805304-1.

Petrov, Alex (2019). *Database Internals*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: OReilly Media, Inc. ISBN: 978-1-492-04034-7.