Evaluación y optimización de consultas

Clase 10

IIC 3413

Prof. Cristian Riveros

¿qué hemos aprendido hasta el momento?

1. Arquitectura de una DBMS.
2. Almacenamiento de los datos.
3. Organización de los datos.
4. Métodos de acceso a los datos (índices).

¿cuál es el siguiente paso?

Cómo responder consultas **eficientemente**!

- Cómo evaluar cada operador relacional.
- Cómo explotar el uso de índices (selección, joins).
- Cómo calcular el costo de cada operador.
- Cómo calcular el costo total de un plan de consulta.
- Cómo escoger el mejor plan de consultas.

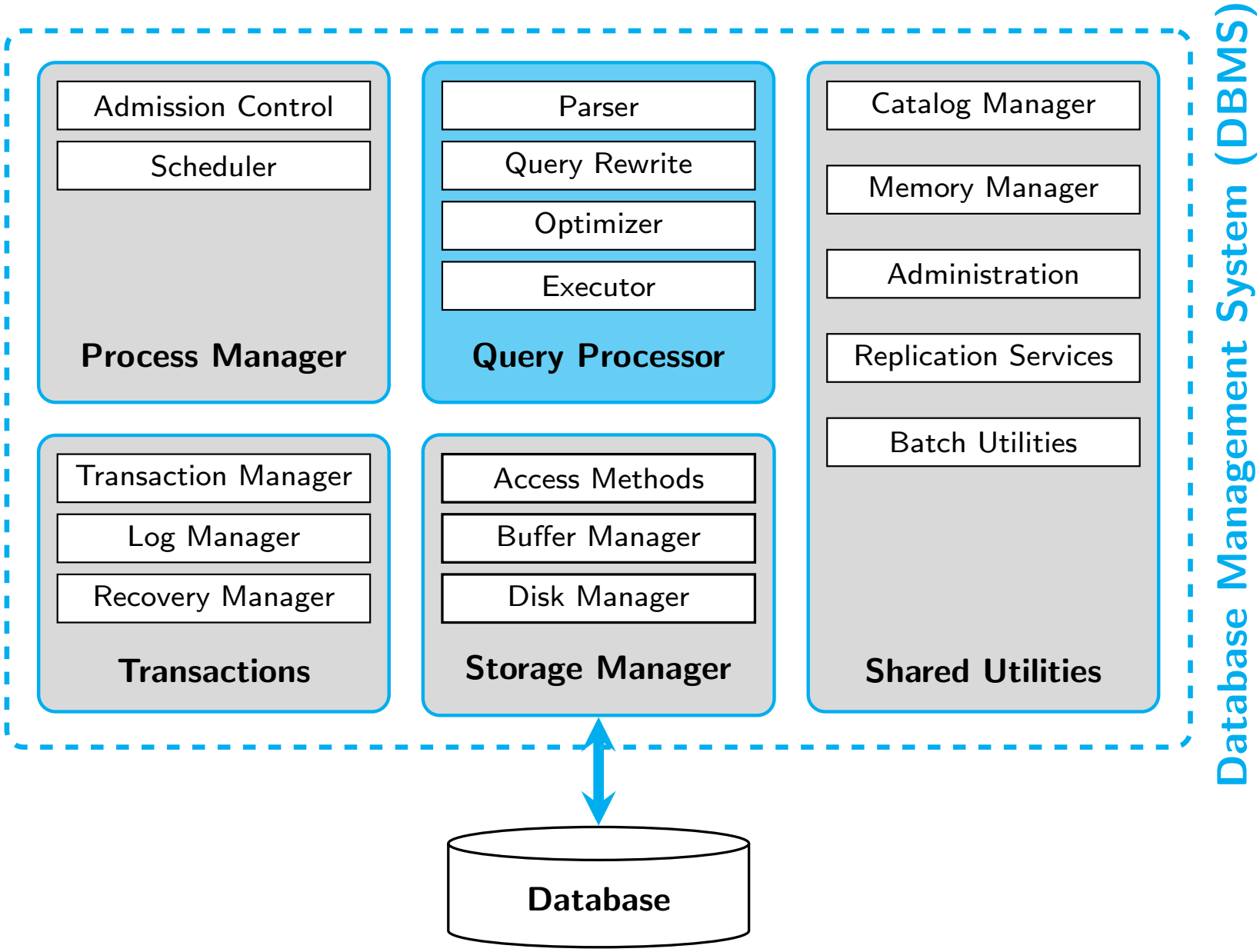
... esto lo veremos las próximas clases.

Para esta clase veremos...

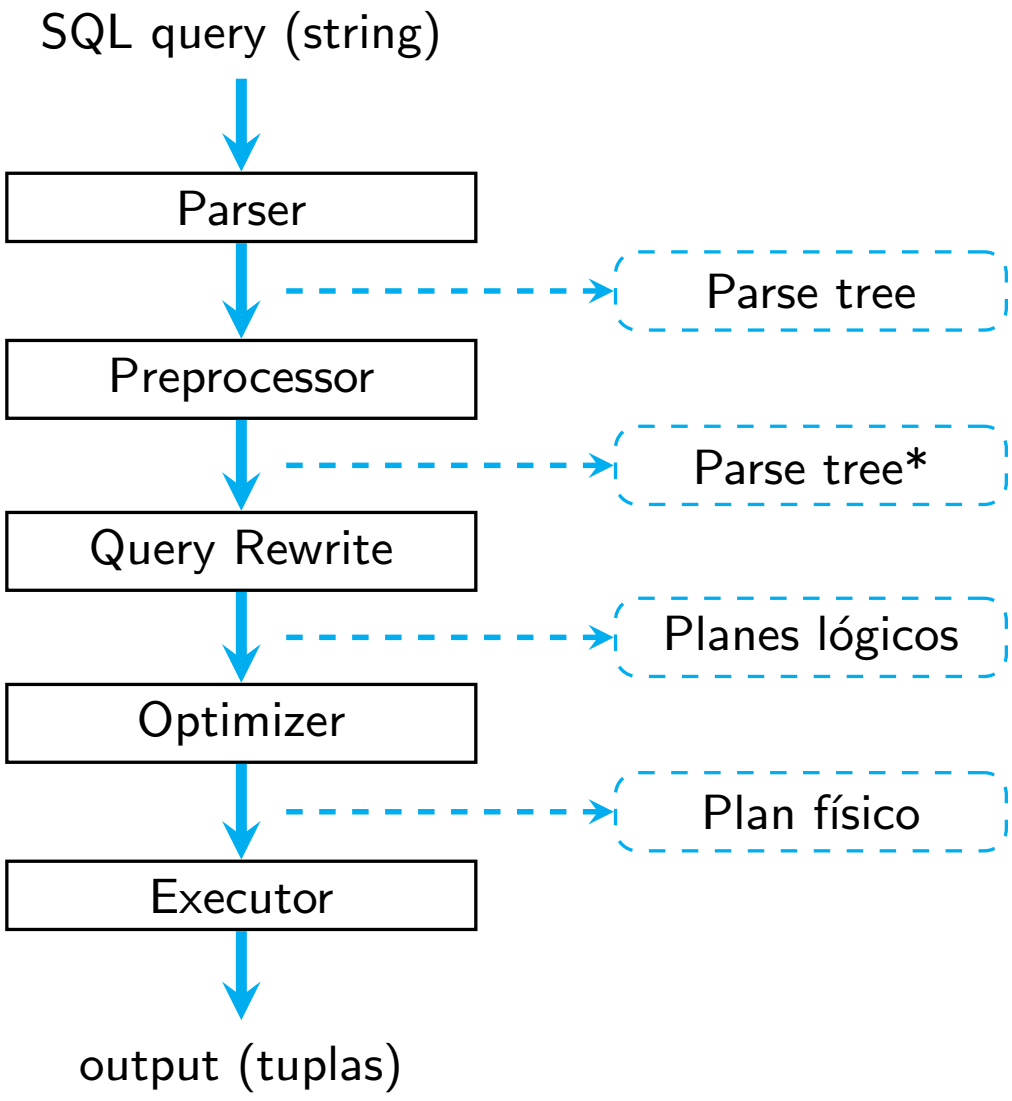
...un **overview** de todo el proceso incluyendo:

- Parser.
- Preprocesamiento y catálogo del sistema.
- Plan lógico.
- Detalles de implementación de operadores relacionales.
- Plan físico.
- Ejecución del plan de consultas.

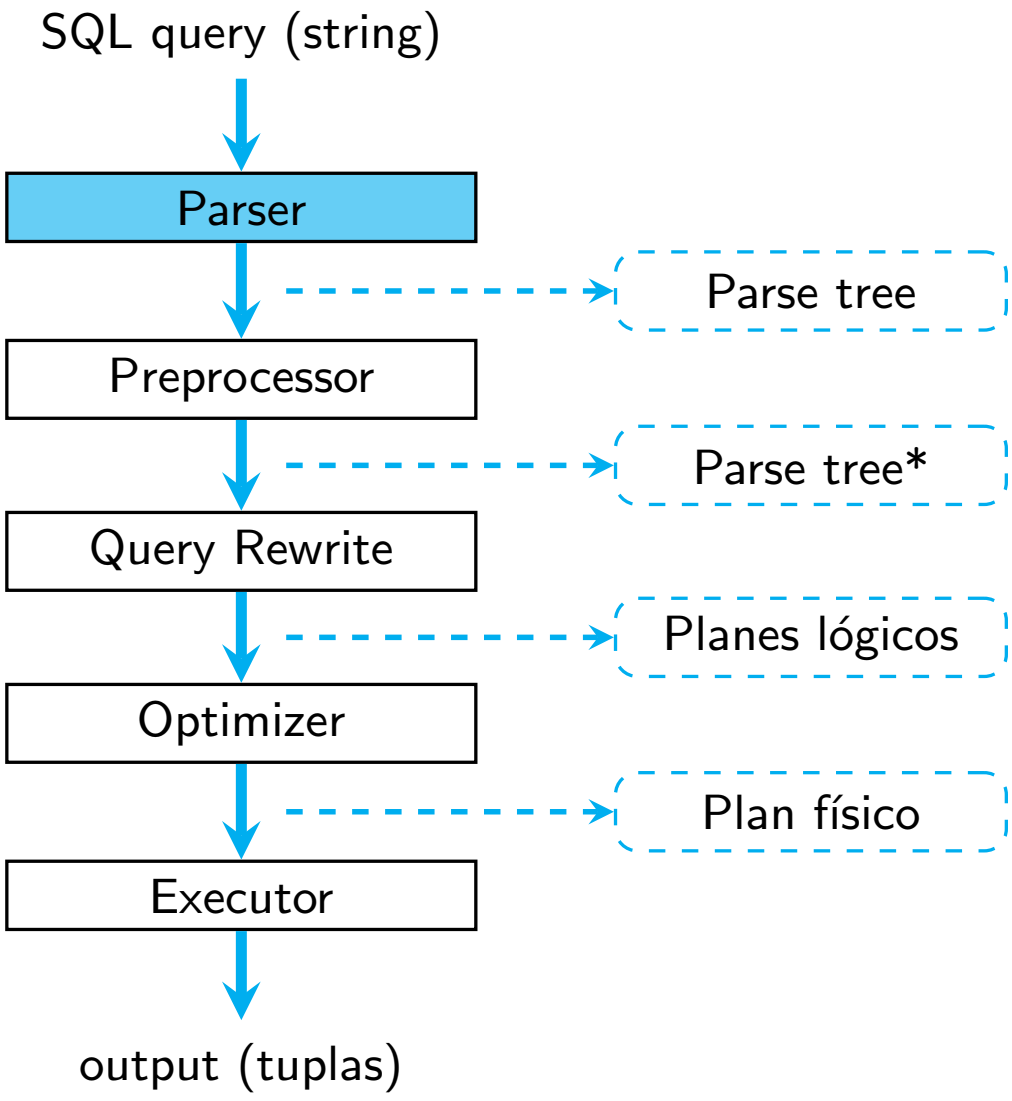
Evaluación y optimización de consultas



Zoom al procesador de consultas



Zoom al procesador de consultas



Sintaxis y Semántica de un lenguaje

Definición

1. La **sintaxis** de una lenguaje es un conjunto de reglas que definen los strings válidos que tienen significado.

¿estas consultas son válidas en SQL?

- `select from tables T1 AND T2 where b - 3`
- `select a from x, z where b = 3`

Sintaxis y Semántica de un lenguaje

Definición

1. La **sintaxis** de una lenguaje es un conjunto de reglas que definen los strings válidos que tienen significado.

2. La **semántica** de un lenguaje define el significado de un string correcto según la sintaxis.

¿cuál es la semántica de esta consulta?

```
select a from x, z where b = 3
```

Sintaxis y Semántica de un lenguaje

Definición

1. La **sintaxis** de una lenguaje es un conjunto de reglas que definen los strings válidos que tienen significado.

2. La **semántica** de un lenguaje define el significado de un string correcto según la sintaxis.

Los encargados de verificar (y ejecutar) la **sintaxis** y **semántica** de `SQL` en una DBMS son:

- Parser.
- Preprocessor.
- Executor (en parte también reescritura y optimizador).

Sintaxis

En este proceso se busca:

- verificar la sintaxis de un string.
- entregar la estructura de un string (árbol de parsing).

Consta de dos etapas:

1. Análisis léxico (**Lexer**).
2. Análisis sintáctico (**Parser**).

Análisis léxico (**Lexer**)

- El análisis léxico consta en dividir la consulta en una sec. de **tokens**.
- Un **token** es un substring (valido) dentro de una consulta.
- Un **token** esta compuesto por:
  - tipo (constante, tabla, ...).
  - valor (el valor mismo del substring).

Análisis léxico (**Lexer**)

Tipos usuales de **tokens** en `SQL`:

- *integer* (constante): 2, 345, 495, ...
- *string* (constante): 'pedro', 'casa', ...
- *keywords*: `SELECT`, `FROM`, ...
- *identificadores*: Jugadores, nombre, ...
- *delimitadores* (char): delimitadores como ',', '=', '+', ...

Análisis léxico (**Lexer**)

Ejemplo

```
select a from x, z where b = 3
```

Tipo	**Valor**
keyword	select
identificador	a
keyword	from
identificador	x
delimitador	,
identificador	z
keyword	where
identificador	b
delimitador	=
integer	3

Análisis léxico (**Lexer**)

- **Lexer** lee el string (consulta) de izquierda a derecha y responde con una secuencia de tokens.
- **Parser** procesa esta secuencia, uno a uno, para hacer el análisis sintáctico.

Análisis sintáctico (**Parser**)

Informalmente:

> *"Una* **gramática** *(parser) es una conjunto de reglas que describen como combinar una secuencia de tokens en nuestro lenguaje."*

Ejemplo

Gramática para definir predicados (`WHERE`):

$$
\begin{aligned}
\texttt{<Constant>} &:= \textit{string}^* \mid \textit{int}^* \\
\texttt{<Field>} &:= \textit{id}^* \\
\texttt{<Expression>} &:= \texttt{<Field>} \mid \texttt{<Constant>} \\
\texttt{<Term>} &:= \texttt{<Expression>} =^* \texttt{<Expression>} \\
\texttt{<Predicate>} &:= \texttt{<Term>} \mid \texttt{<Predicate>}\ \textit{AND}^*\ \texttt{<Predicate>}
\end{aligned}
$$

Los terminos con $(\cdot)^*$ son tokens.

Gramática

Definición

Una **gramática** G (o gramática libre de contexto) es un par:

$$G = (V, T, P, S)$$

- V es un conjunto de **variables** (ej. `<Constant>`),
- T es un conjunto de **tokens**,
- $S \in V$ es la variable inicial,
- P es un conjunto de **reglas** de la forma:

$$\texttt{<var>} := \alpha$$

donde `<var>` $\in V$ y α es un string en $V \cup T$.

Gramática

Ejemplo

$$
\begin{aligned}
\texttt{<Constant>} &:= \textit{string}^* \mid \textit{int}^* \\
\texttt{<Field>} &:= \textit{id}^* \\
\texttt{<Expression>} &:= \texttt{<Field>} \mid \texttt{<Constant>} \\
\texttt{<Term>} &:= \texttt{<Expression>} =^* \texttt{<Expression>} \\
\texttt{<Predicate>} &:= \texttt{<Term>} \mid \texttt{<Predicate>}\ \textit{AND}^*\ \texttt{<Predicate>}
\end{aligned}
$$

- $V = \{\texttt{<Constant>}, \texttt{<Field>}, \texttt{<Expression>}, \ldots\}$
- $T = \{\textit{string}, \textit{int}, =, \textit{AND}\}$
- $S = \texttt{<Predicate>}$
- $P =$ conjunto de reglas

Árbol de derivación

Definición

Un **árbol de derivación** de $G = (V, T, P, S)$ es un árbol ordenado y rotulado $\mathcal{T}$ con etiquetas en $V \cup T$ tal que:

- $\text{label}(h) \in T$ para todo nodo hoja $h \in \text{nodes}(\mathcal{T})$,
- $\text{label}(n) \in V$ para todo nodo interno $n \in \text{nodes}(\mathcal{T})$,
- $\text{label}(\text{root}(\mathcal{T})) = S$,
- si $n_1, \ldots, n_k$ son los hijos de $n \in \text{nodes}(\mathcal{T})$, entonces:

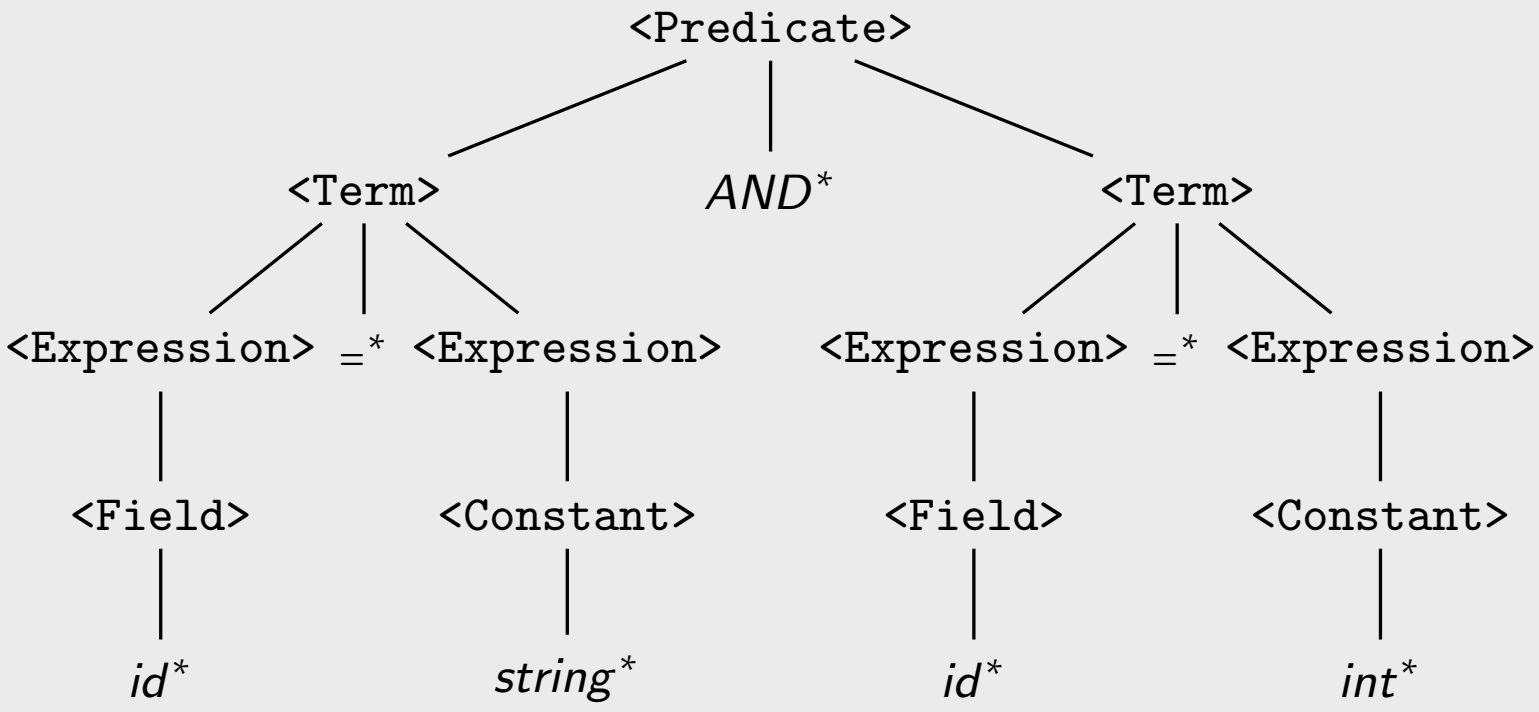
$$\text{label}(n) := \text{label}(n_1)\ \text{label}(n_2) \cdots \text{label}(n_k)$$

es una regla en P.

Árbol de derivación

Ejemplo

Un **árbol de derivación** para la gramática anterior:



Árbol de parsing

Sea Q una consulta y $\bar{Q}$ su secuencia de tokens.

Definición

Un **árbol de parsing** de $G = (V, T, P, S)$ para Q es un árbol de derivación $\mathcal{T}$ de G tal que si $h_1, \ldots, h_n$ son las hojas de $\mathcal{T}$ (en pre- o post-order) entonces:

$$\bar{Q} = \text{label}(h_1) \cdot \text{label}(h_2) \cdot \ldots \cdot \text{label}(h_n)$$

Árbol de parsing

Ejemplo

$$nombre = \text{'Cristian'} \quad AND \quad edad = 31$$

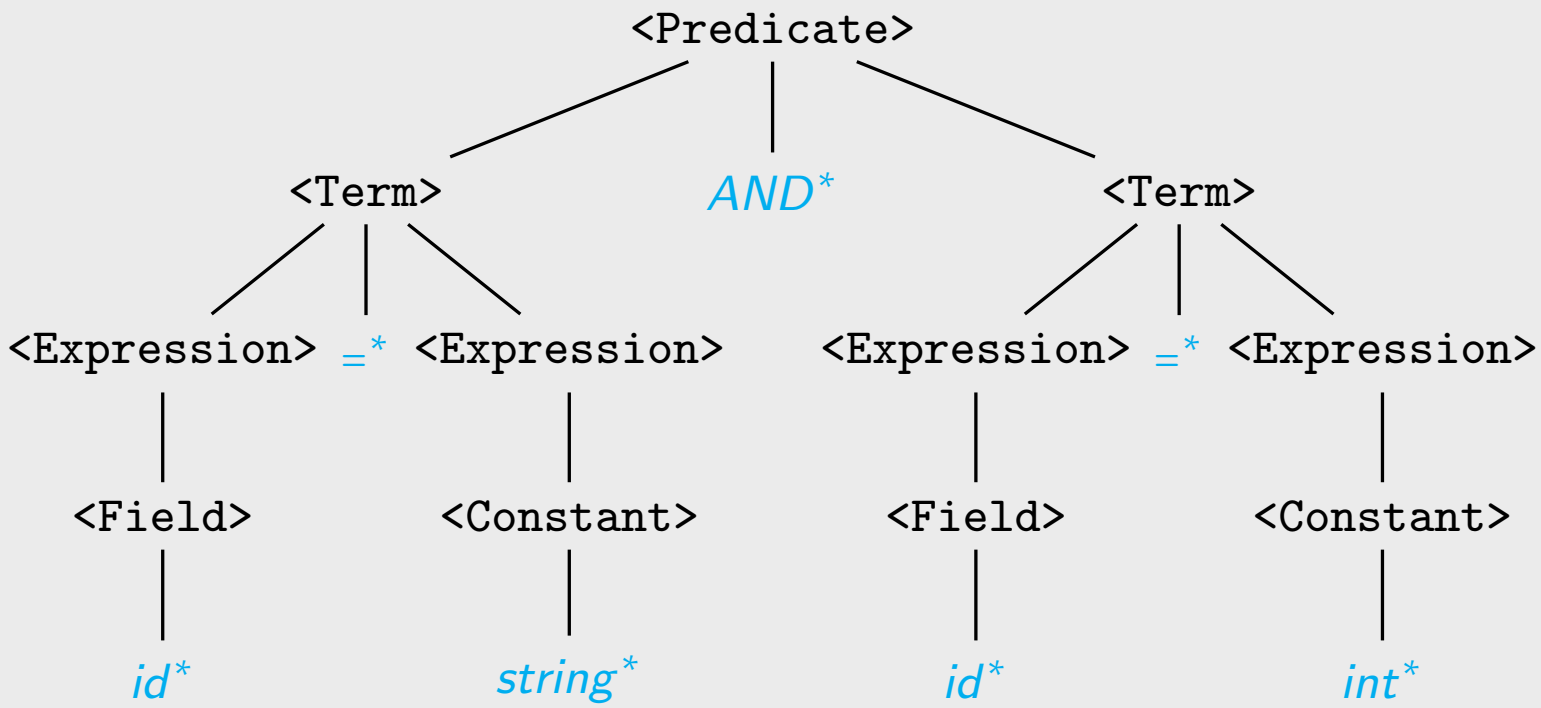
$$\begin{aligned}
\texttt{<Constant>} &:= string^* \mid int^* \\
\texttt{<Field>} &:= id^* \\
\texttt{<Expression>} &:= \texttt{<Field>} \mid \texttt{<Constant>} \\
\texttt{<Term>} &:= \texttt{<Expression>} =^* \texttt{<Expression>} \\
\texttt{<Predicate>} &:= \texttt{<Term>} \mid \texttt{<Predicate>}\ AND^*\ \texttt{<Predicate>}
\end{aligned}$$

Árbol de parsing

Ejemplo

nombre = 'Cristian' *AND* *edad* = 31

Un **árbol de parsing** para esta expresión es:



El **árbol de parsing** le da "estructura" a una oración.

Resumen del proceso del **Parser**

INPUT: Consulta Q y gramática $G_{\texttt{SQL}}$ ($\texttt{SQL}$).

OUTPUT: Árbol de parsing $\mathcal{T}$ de Q.

1. **Lexer** (o análisis léxico): lee y convierte Q en una sequencia de tokens:

$$\bar{Q} \;=\; t_1 \ldots t_n$$

2. **Parser** (o análisis sintáctico): convierte $\bar{Q}$ en un árbol de parsing $\mathcal{T}$ de $G_{\texttt{SQL}}$ para Q.

Árbol de parsing es nuestro punto de partida para la optimización.

Detalles de implementación del **Lexer** y **Parser**

- El algoritmo de Lexer para generar
  la secuencia de tokens va mas allá de este curso.

  (lenguajes regulares y automatás finitos)

- El algoritmo de Parser para generar
  el árbol de parsing va mas allá de este curso.

  (lenguajes libres de contexto y autómatas con stack)

En este curso daremos el lexer y parser como ya implementados.

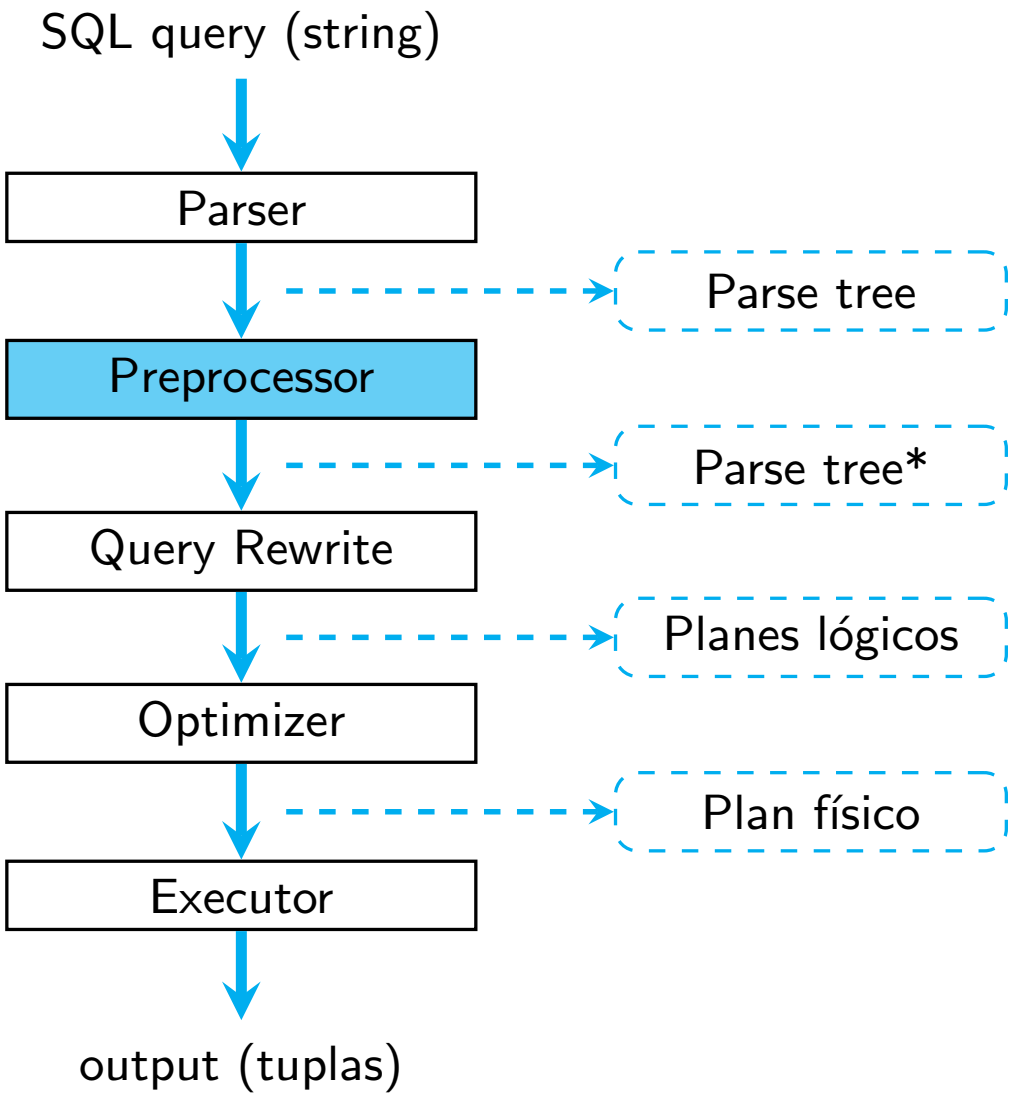
Sobre gramáticas

La gramática de un lenguaje puede ser muy compleja.

Por ejemplo, ver la gramática de `SQL-2003`:

- `https://ronsavage.github.io/SQL/sql-2003-2.bnf.html`

Zoom al procesador de consultas



Preprocesamiento de consultas

Antes de la reescritura de consultas es necesario:

- Chequear que el uso de las tablas es correcto.
- Chequear y resolver el uso de atributos.
- Chequear que los tipos son correctos.
- Calcular expresiones simples.
- Preprocesar el uso de vistas.

Para chequear esta información usamos
el catalogo del sistema.

Metadatos y catálogo del sistema

Definición

- Los **metadatos** es la información de la DB, fuera de su contenido.
- El **catálogo del sistema** es el encargado de almacenar los metadatos en tablas (relaciones) del mismo sistema.

Los metadatos se pueden dividir en 4 categorías:

1. Información sobre las **relaciones**.
2. **Estadísticas** del contenido.
3. Información sobre los **índices**.
4. Información sobre las **vistas**.

1. Información sobre las relaciones

Por cada relación se almacenan:

- Nombre de la relación.
- Dirección de almacenamiento.
- Restricciones de integridad.
- Para cada campo/atributo:
  - Nombre.
  - Tipo de dato.
  - Tamaño.
  - Detalles de almacenamiento.
- ...

2. Estadísticas del contenido

Por cada relación se almacena:

- Número de páginas.
- Número de tuplas/records.
- Tamaño (promedio) de una tupla.
- Para cada campo/atributo:
  - Cantidad de valores distintos.
  - Tamaño (promedio).
  - Distribución de los datos (histogramas).
  - Valores máximos y mínimos.
- ...

Estos datos son muy importantes para la optimización (?)

(... los veremos en detalle mas adelante)

3. Información sobre los índices

Por cada índice de una relación se almacena:

- Nombre del índice.
- Campo indexado.
- Tipo de índice.
- Tamaño del índice.
- Altura (en caso árbol).
- Número de overflow pages.
- Costo promedio del índice (en bloques).
- ...

Estos datos son también importantes para la optimización!

4. Información sobre las vistas

Por cada vista se almacena:

- Nombre.
- Definición.
- Materialización (si existe).
- ...

Almacenamiento y actualización del catálogo

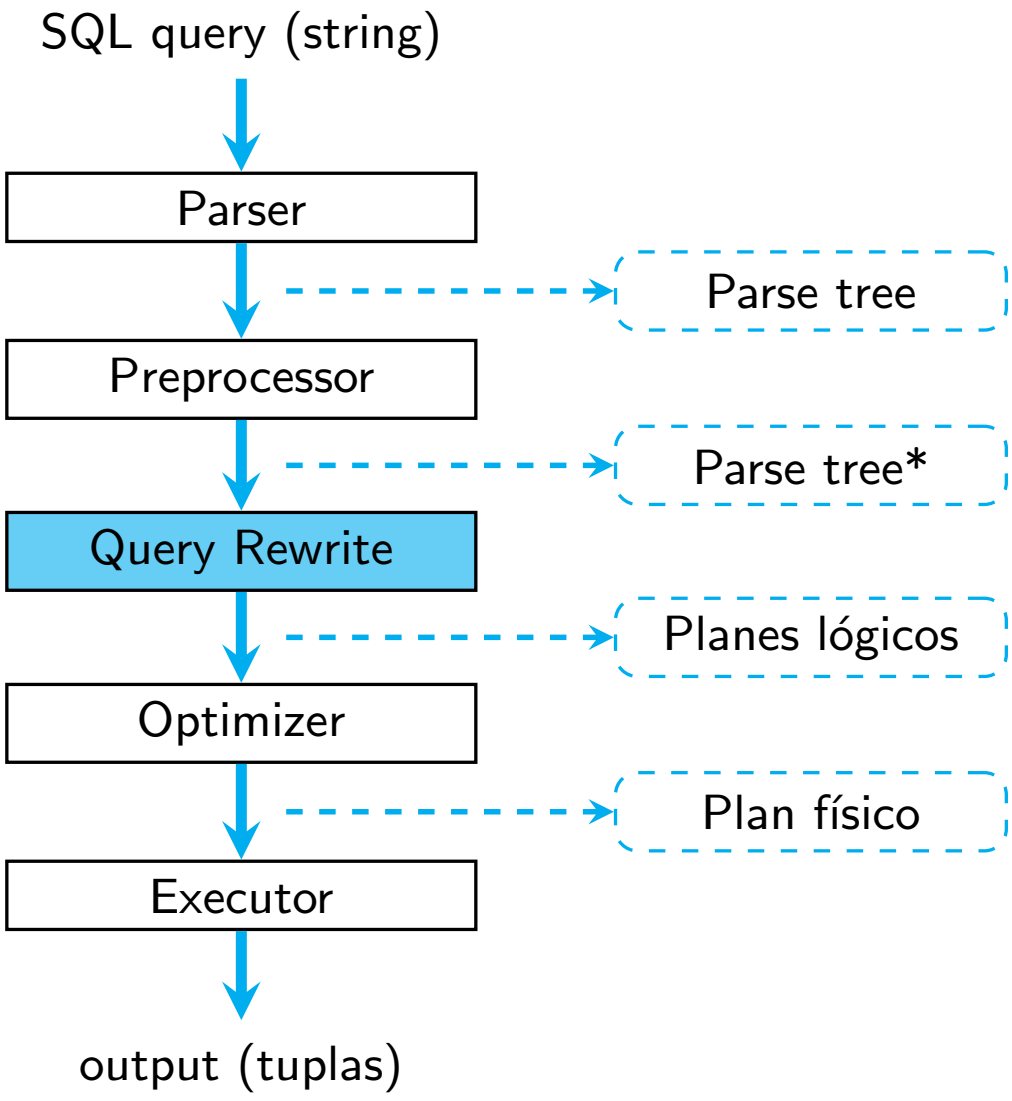
- Metadatos son almacenados en varias relaciones (catálogo).
- Metadatos del catálogo son traidos a memoria al iniciar sistema.

¿cuándo actualizamos la información del catálogo?

Varias alternativas:

1. Consulta a consulta.
2. Periódicamente.
3. Recalcular metadatos (siempre que se estime necesario).

Zoom al procesador de consultas



¿qué es un plan lógico?

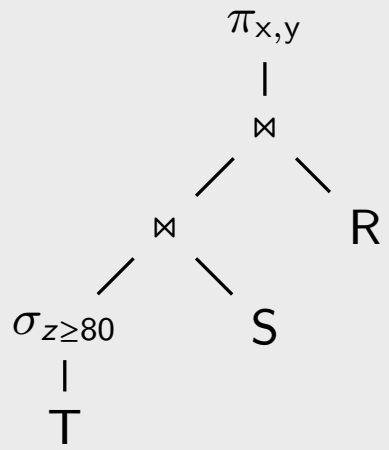
Definición

Un **plan lógico** es un árbol ordenado y etiquetado $\mathcal{T}$ tal que:

- label(h) es una relación,
- label(n) es un operador de algebra relacional y
- $|\text{children}(n)| = \text{arity}(\text{label}(n))$

para todo nodo hoja $h \in \text{nodes}(\mathcal{T})$ y nodo interno $n \in \text{nodes}(\mathcal{T})$.

Ejemplo



¿cómo convertimos un árbol de parsing en un plan lógico?

```
SELECT   <SelList>
FROM     <FromList>
WHERE    <Condition>
```

1. Combinamos las relaciones en el `<FromList>` con **productos cruz**:

$$R_1 \times \ldots \times R_n$$

2. Aplicamos **selección** con la condición C dada en el `<Condition>`:

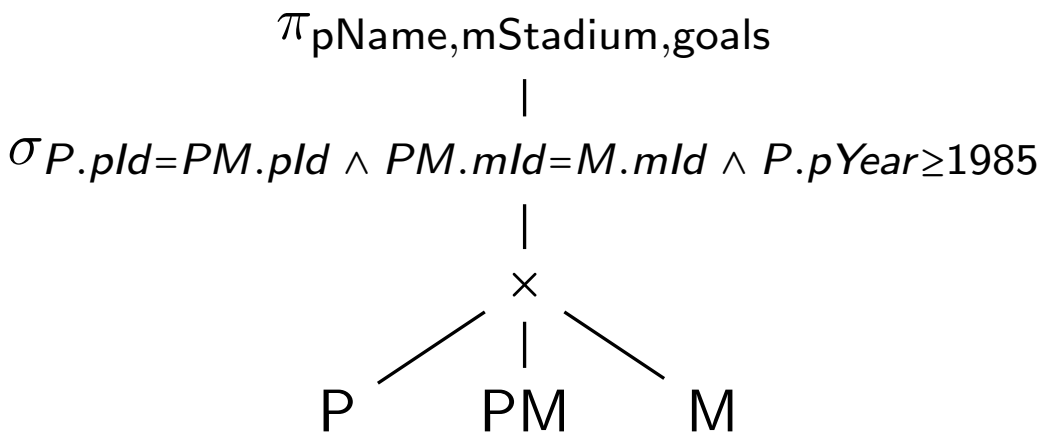
$$\sigma_{\texttt{<Condition>}}(R_1 \times \ldots \times R_n)$$

3. Aplicamos **proyección** con las atributos dados en el `<SelList>`:

$$\pi_{\texttt{<SelList>}} (\sigma_{\texttt{<Condition>}}(R_1 \times \ldots \times R_n))$$

Ejemplo de árbol de parsing a plan lógico

```
SELECT   pName, mStadium, goals
FROM     Players AS P, Matches AS M, Players_Matches AS PM
WHERE    P.pId = PM.pId AND PM.mId = M.mId AND
         P.pYear ≥ 1985
```



Desde un árbol de parsing a un plan lógico

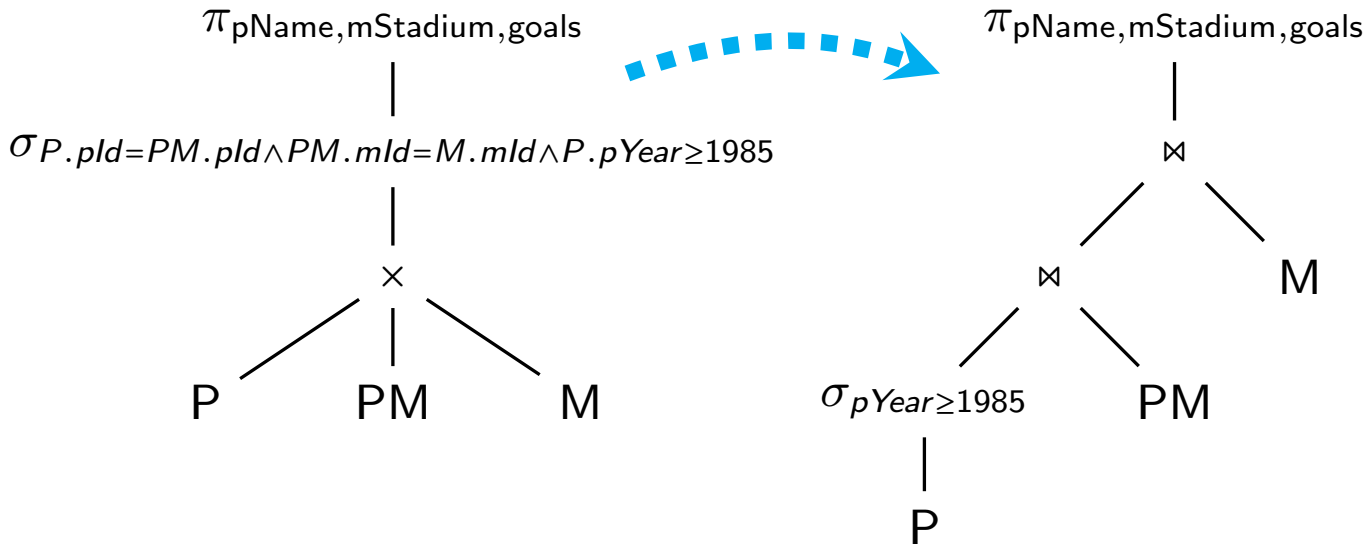
La transformación desde un árbol a un plan lógico es un poco más compleja:

- Considerar **otros operadores** (union, intersección, etc).
- Considerar consultas **anidadas** y **correlacionadas**.
- Considerar **vistas**.

(...estos subpasos los veremos más adelante.)

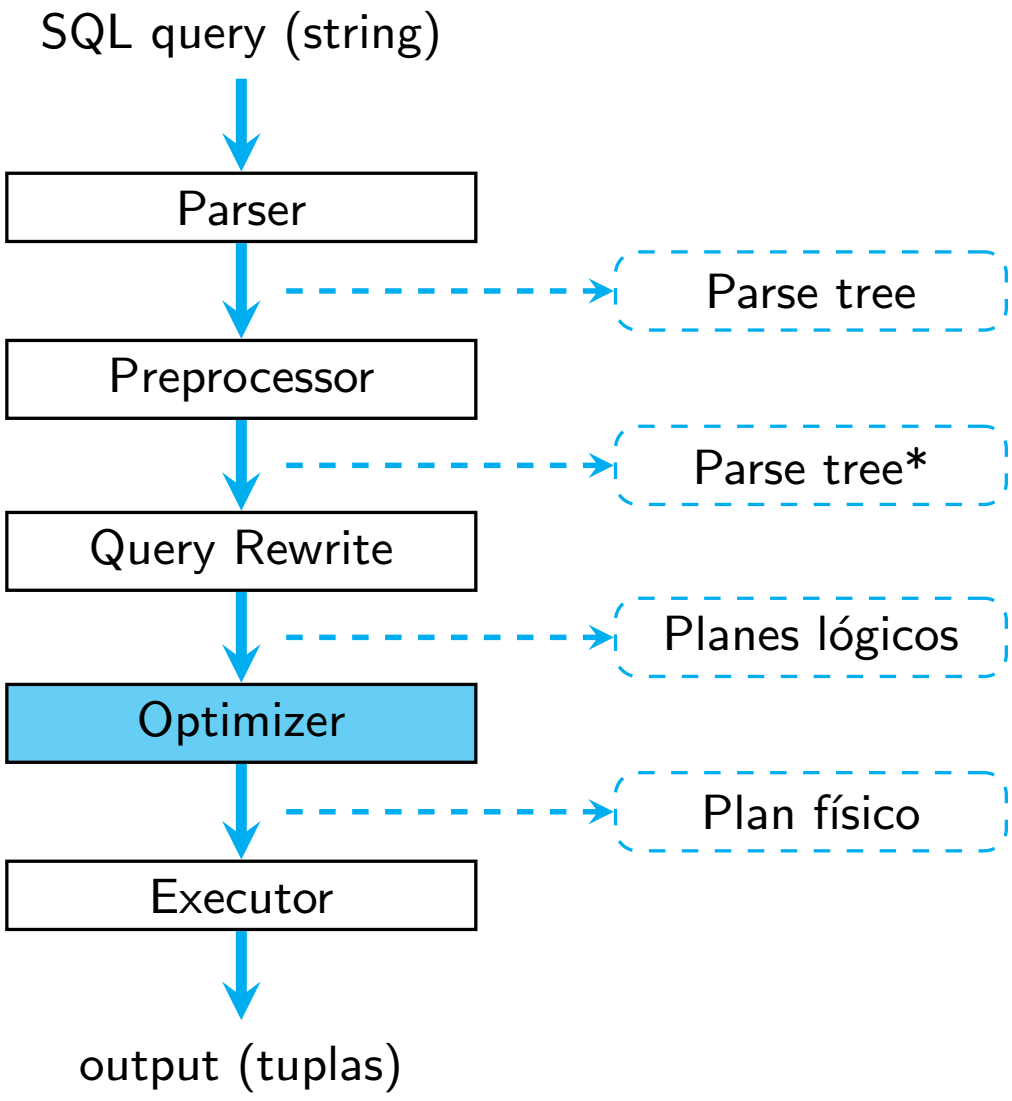
Re-escritura de consultas

1. Convertir el árbol de parsing en un plan lógico.
2. Reescribe consulta aplicando reglas de algebra relacional.
3. Crea un set de planes lógicos **prometedores** para ser optimizados.



Este paso lo veremos en detalle más adelante.

Zoom al procesador de consultas



Implementación de operadores relacionales

Diversas **implementaciones** por cada **operador** relacional.

Selection:

- Index/No Index.
- Sorted/unsorted.
- Multiples condiciones.

Join:

- Nested Loops Join.
- Sort-Merge Join.
- Hash Join.

Projección:

- Sorting.
- Hashing.

Unión / Diferencia:

- Sorting.
- Hashing.

Sorting:

- Merge-sort.
- B+-trees.

Agregación:

- Uso de índices.

Esta lista no es exhaustiva y no considera variaciones!

Implementación de operadores relacionales

Diversas **implementaciones** por cada **operador** relacional.

- No existe una implementación que sea mejor que todas.
- Mejor algoritmo depende de la distribución y tamaño de los datos.

Cada **implementación** la veremos en detalle la próxima clase.

Todos estas implementaciones coinciden en su interfaz

Cada operador implementa esta interfaz:

`open()`

- Inicializa el estado del operador.
- Ajusta parámetros (ej. condición de selección).

`next()`

- Procesa el input y responde la siguiente tupla.

`close()`

- Limpia elementos temporales.

Importante para componer y ejecutar el **plan físico**!

¿qué es un plan físico?

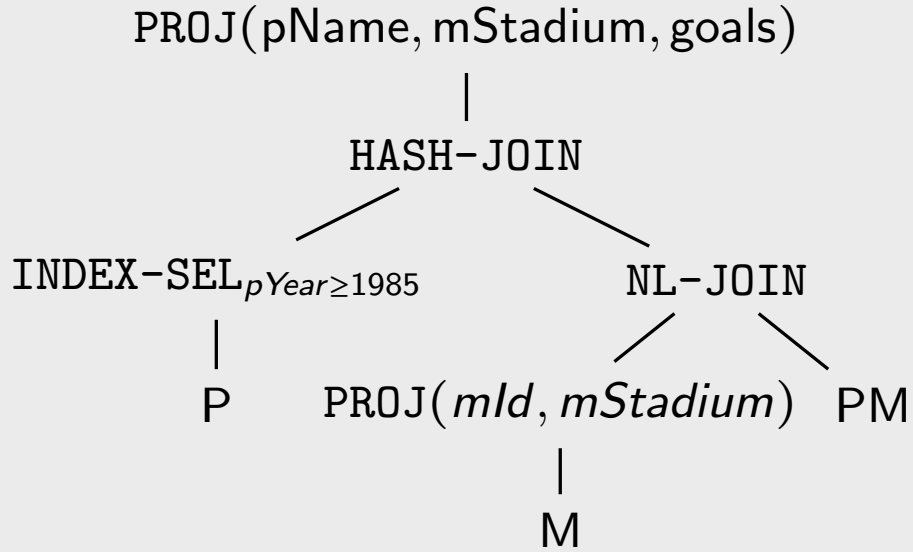
Definición

Un **plan físico** es un árbol ordenado y etiquetado $\mathcal{T}$ similar a un plan lógico:

- label(n) es una **implementación** de un operador en Algebra Relacional

para todo nodo interno $n \in \text{nodes}(\mathcal{T})$.

Ejemplo



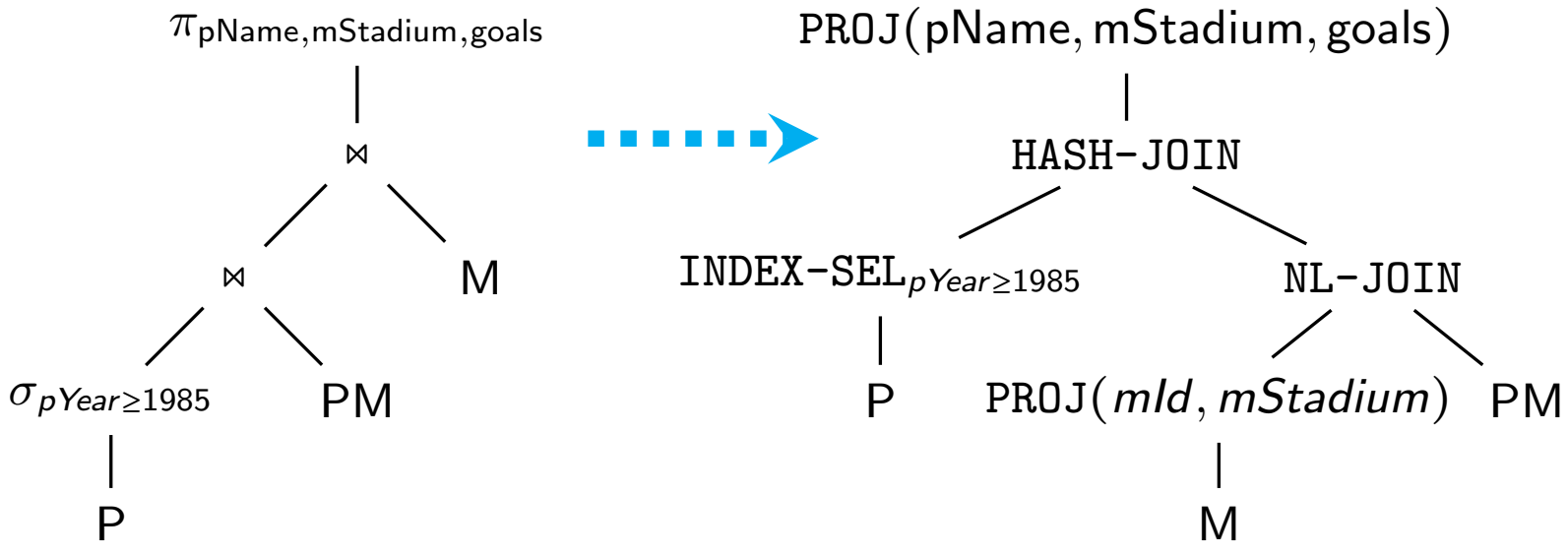
Detalles de un plan físico

Un plan físico también cuenta con un poco más de información como:

1. Selección del **access method** para cada relación.
2. Detalles de **implementación**.
   - Orden del input importa.
3. Detalles de la **ejecución**.
   - Pipeline.
   - Materialización.

Optimización de consultas

1. Convertir los planes lógicos en planes físicos.
2. Calcular el **costo** de cada plan físico.
3. Escoger el plan con menor costo.



¿cuál costo? ¿cómo calculamos ese costo?

¿cuál costo considerar?

Algunas opciones:

- Acceso a disco (estándar).
- Ciclos de CPU.
- Uso de memoria.
- Balance de la carga (computación en paralelo).
- Comunicación.

En este curso nos concentraremos en acceso a disco (I/O).

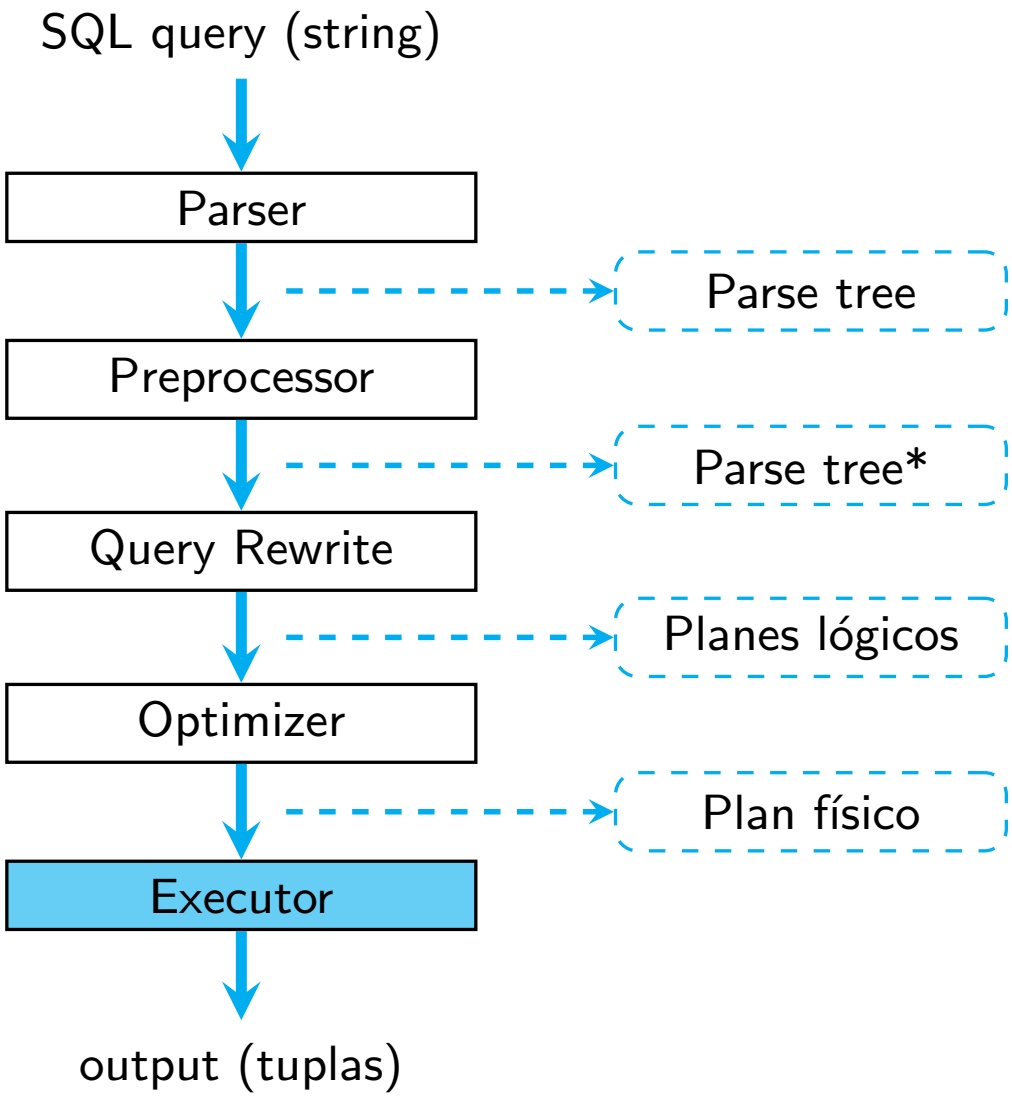
Costo de un plan físico

1. Estimamos el costo (en I/O) de cada operador.
   - Estimación usa las estadísticas del catálogo.
2. El costo del plan es la suma del costo (I/O) de cada operador.

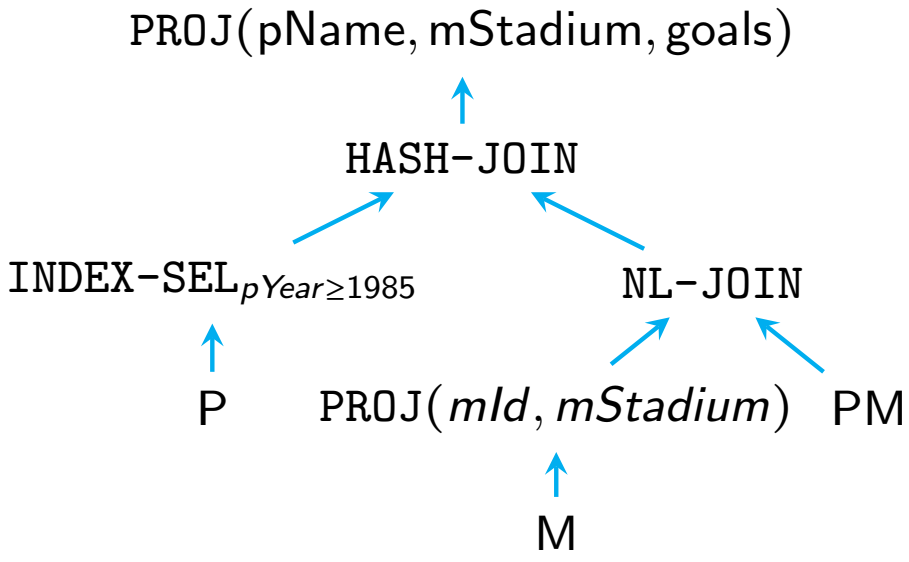
"Performance del DBMS esta fuertemente ligado a una buena estimación del costo de sus planes físicos."

La **estimación de costo** lo veremos en detalle más adelante.

Zoom al procesador de consultas



Pipeline



- Ejecución en serie del plan físico.
- Cada operador genera una tupla a la vez.

Cada operador no almacena sus resultados intermedios y los retorna a su operador padre.

Beneficios del Pipelinine

- Simplifica sincronización.
- Evita lectura/escritura intermedia de los resultados.
- Posible paralelización de operadores.

¿es siempre posible usar pipeline de los resultados?

Bloqueo de operadores

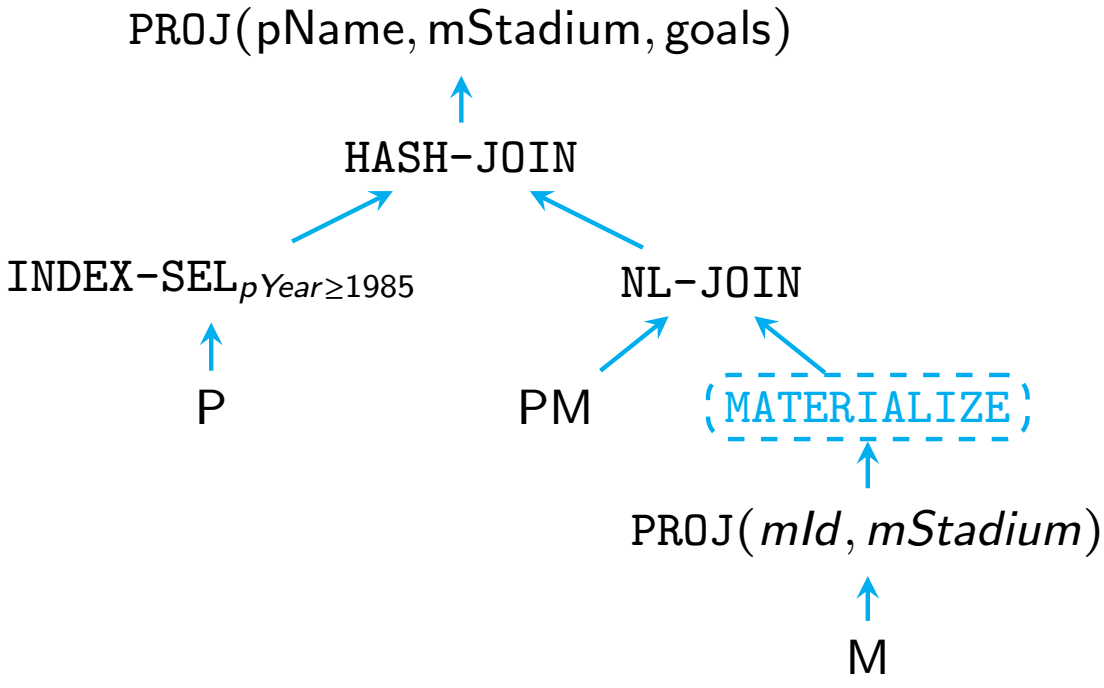
Algunos operadores deben esperar a terminar todo el proceso:

- external-sorting.
- joins.
- group-by y eliminación de duplicados.

Estos operadores usualmente requieren:

materialización (alternativa a Pipeline).

Materialización



- Almacena el resultado de la subconsulta en disco.
- Solo necesario si la subconsulta es requerida reiteradas veces.

Pipeline vs Materialización

Siempre se prefiere pipeline sobre materialización

¿cuándo usaremos materialización?

1. Subconsulta es llamada reiteradas veces.
2. Costo de almacenamiento es menor a recalcular la consulta.

Finalmente, retornamos el resultado...