Bases de Datos

Clase 12: Evaluación de consultas

Recordatorio: Paginas, Discos y Buffer

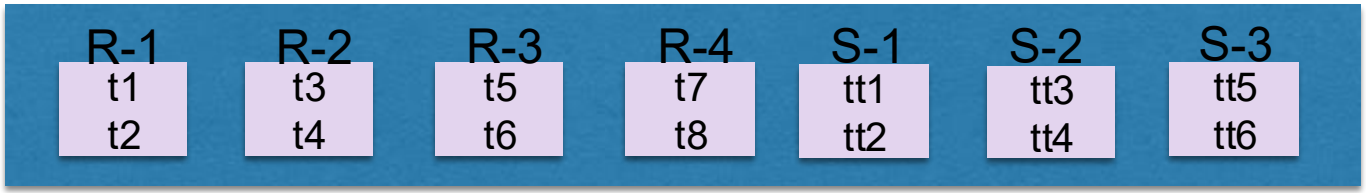
Páginas, disco y buffer

Para trabajar con las tuplas de una relación, la base de datos carga la página desde el disco con dicha tupla

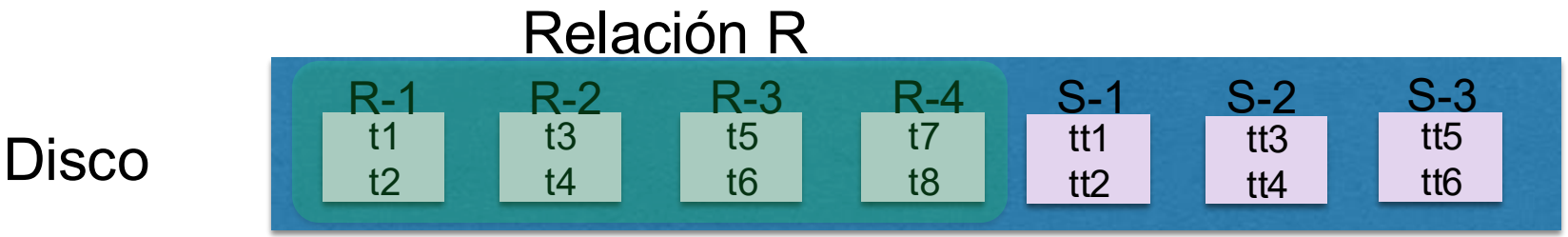
Para cargar estas páginas, la base de datos reserva un espacio en RAM llamado **Buffer**

Páginas, disco y buffer

Disco



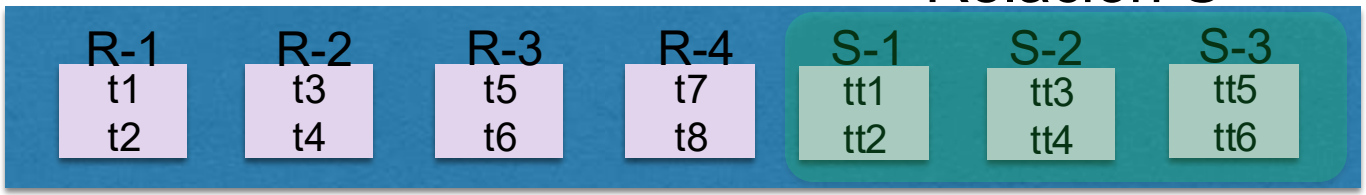
Páginas, disco y buffer



Páginas, disco y buffer

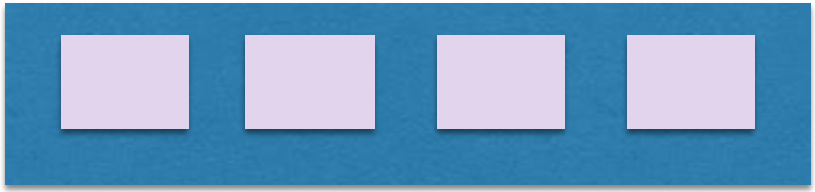
Relación S

Disco

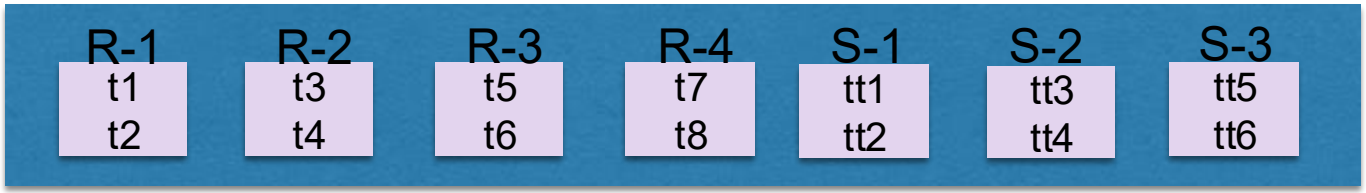


Páginas, disco y buffer

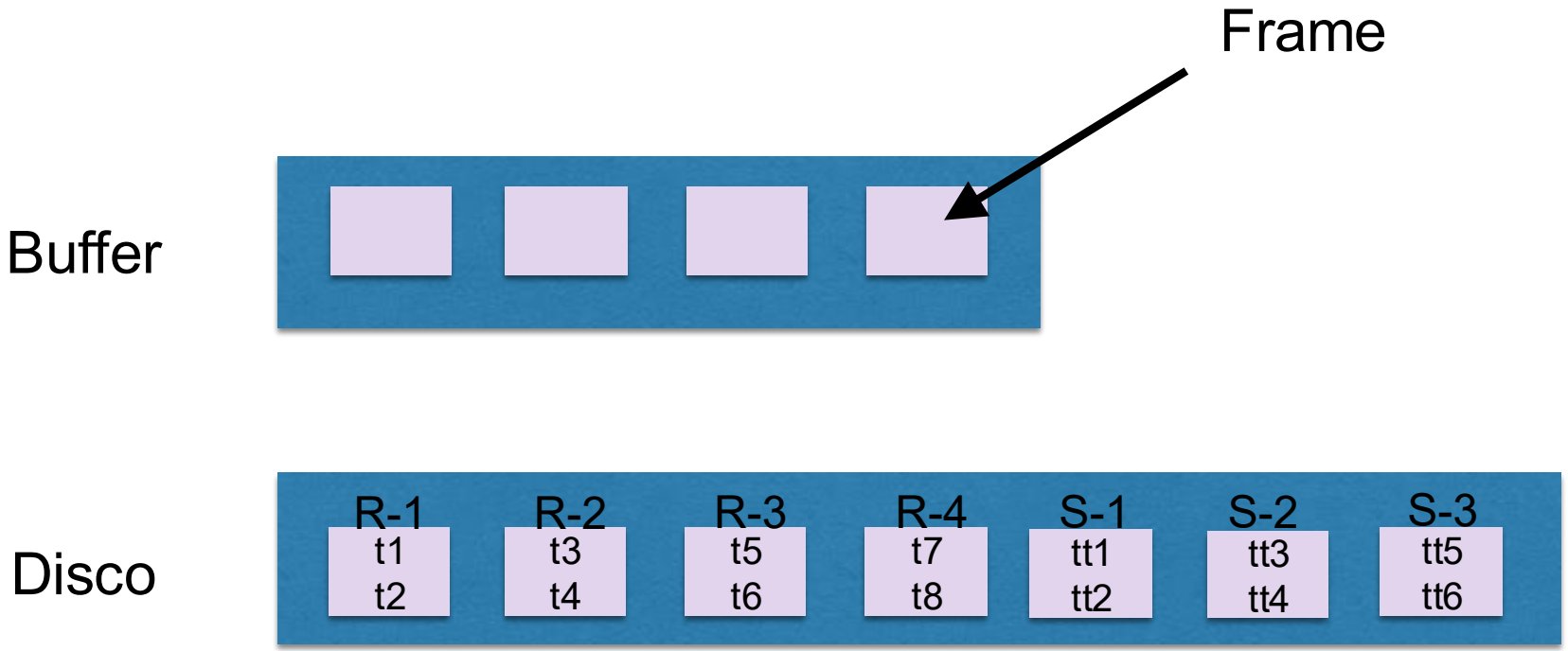
Buffer



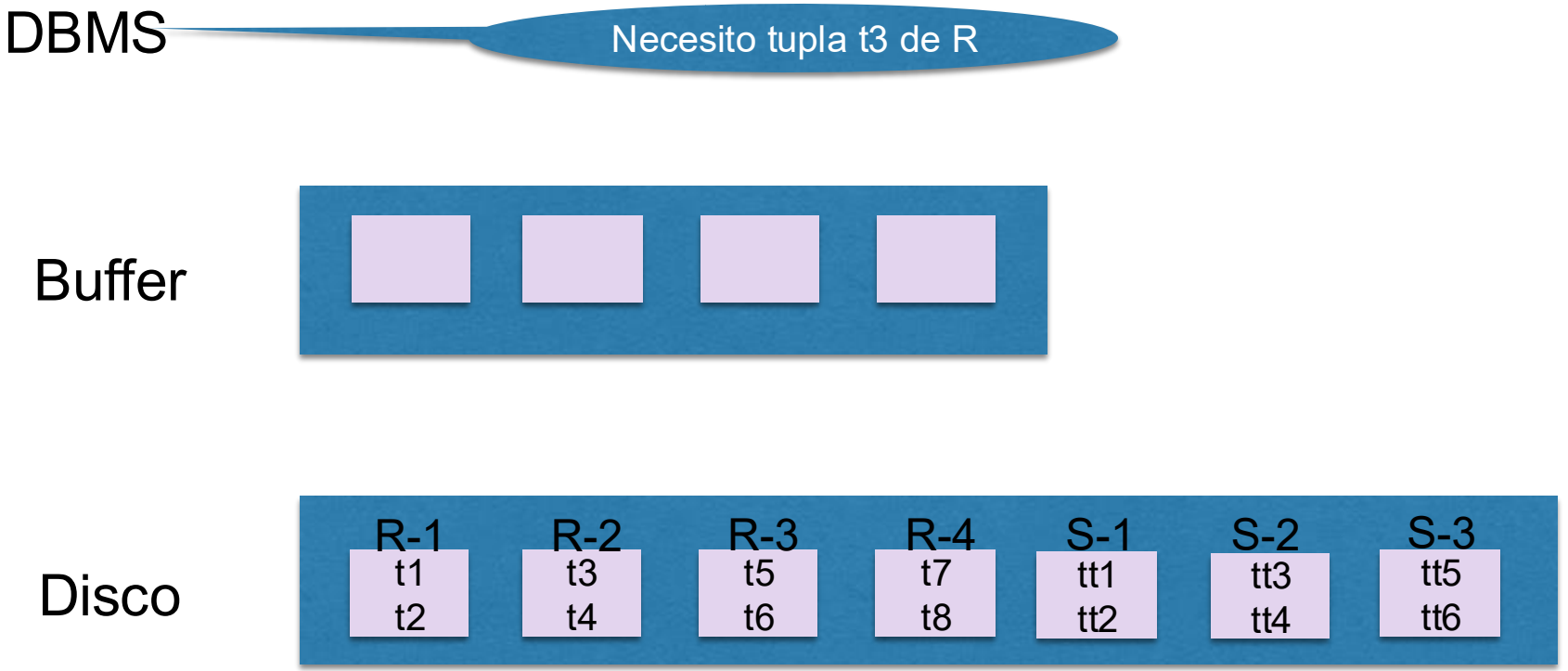
Disco



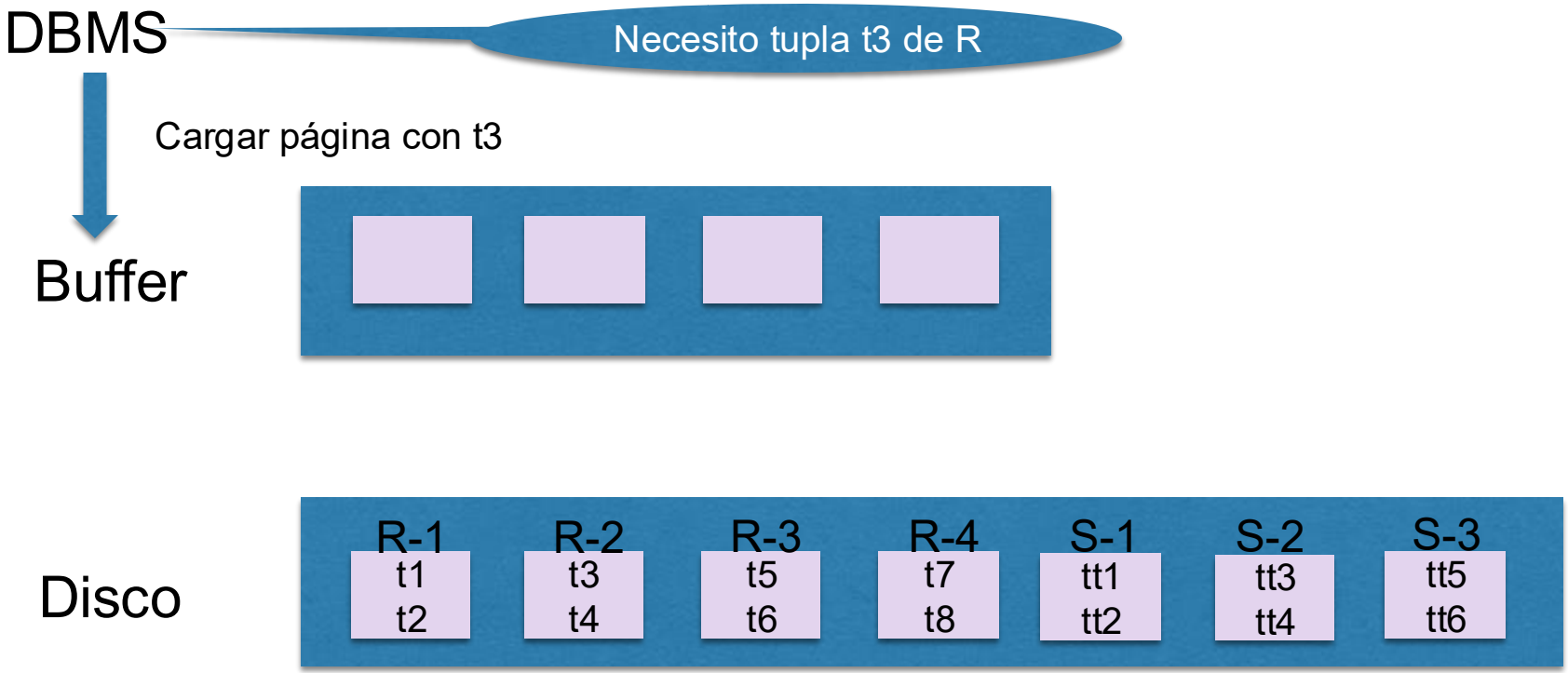
Páginas, disco y buffer



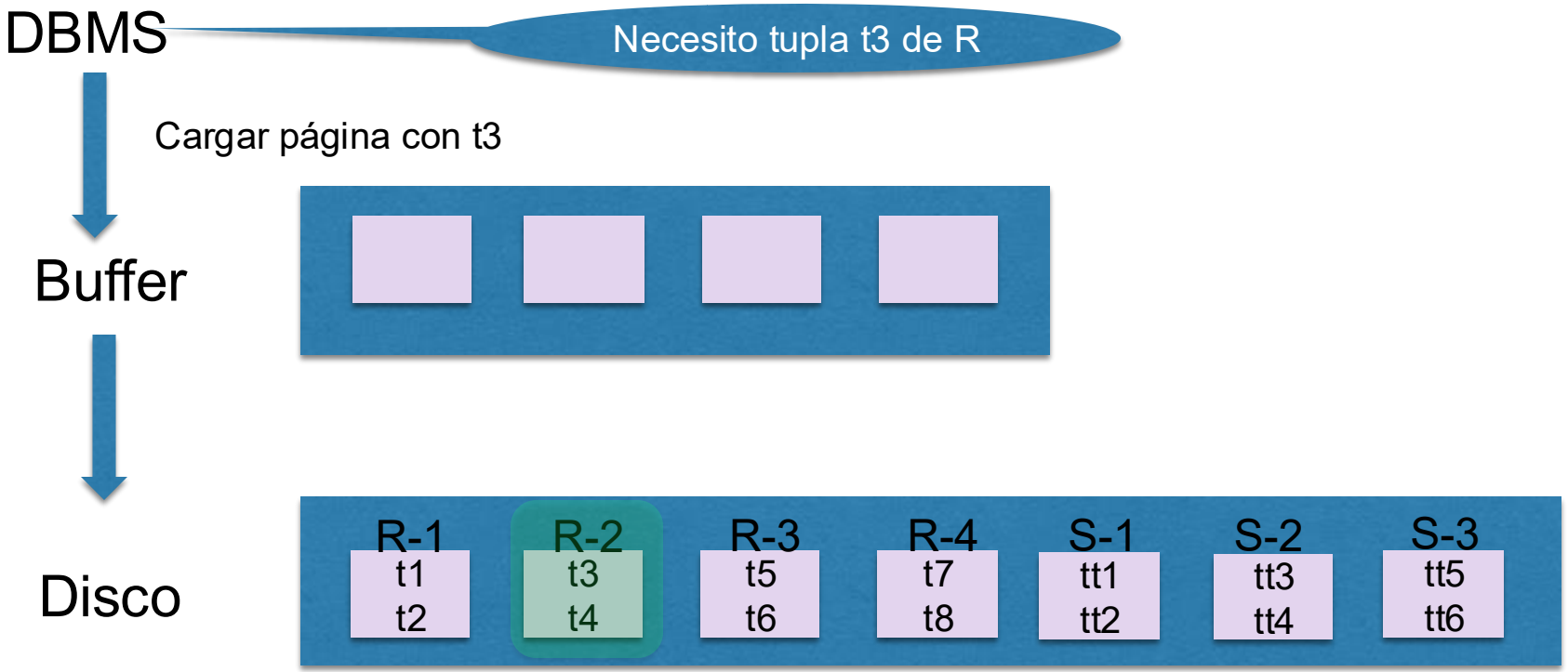
Páginas, disco y buffer



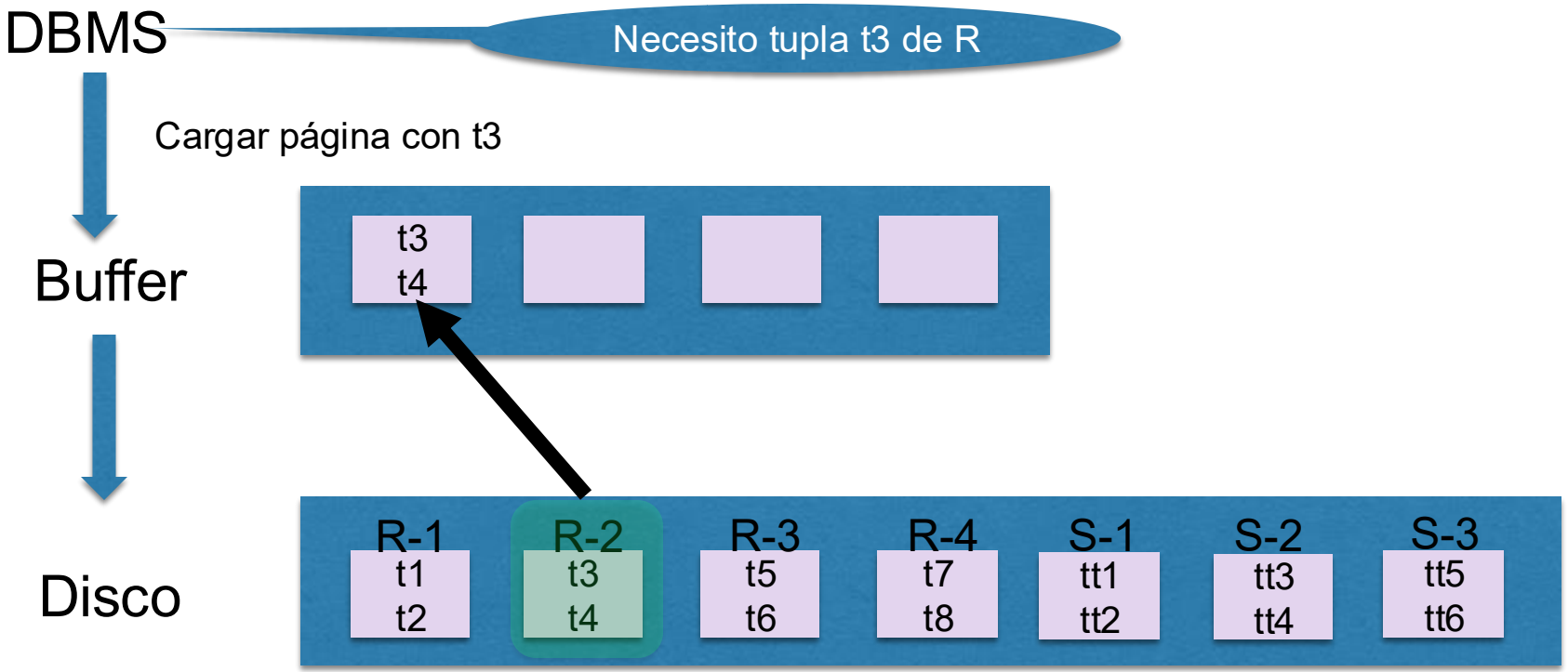
Páginas, disco y buffer



Páginas, disco y buffer



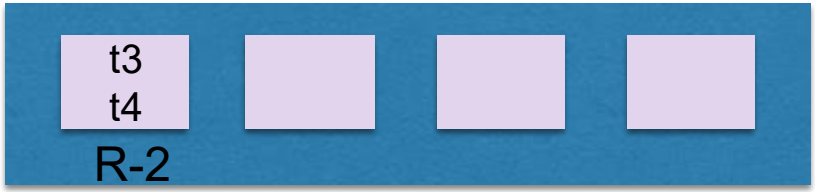
Páginas, disco y buffer



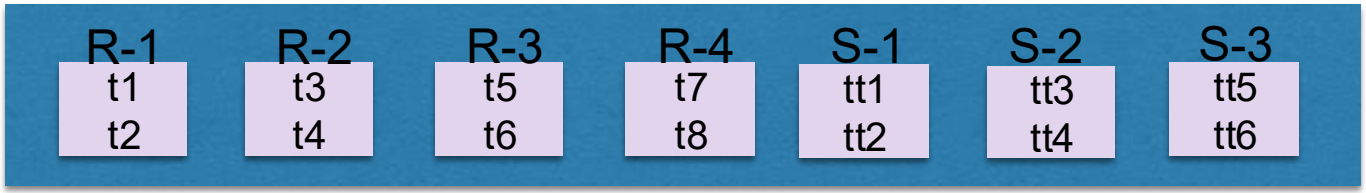
Páginas, disco y buffer

DBMS

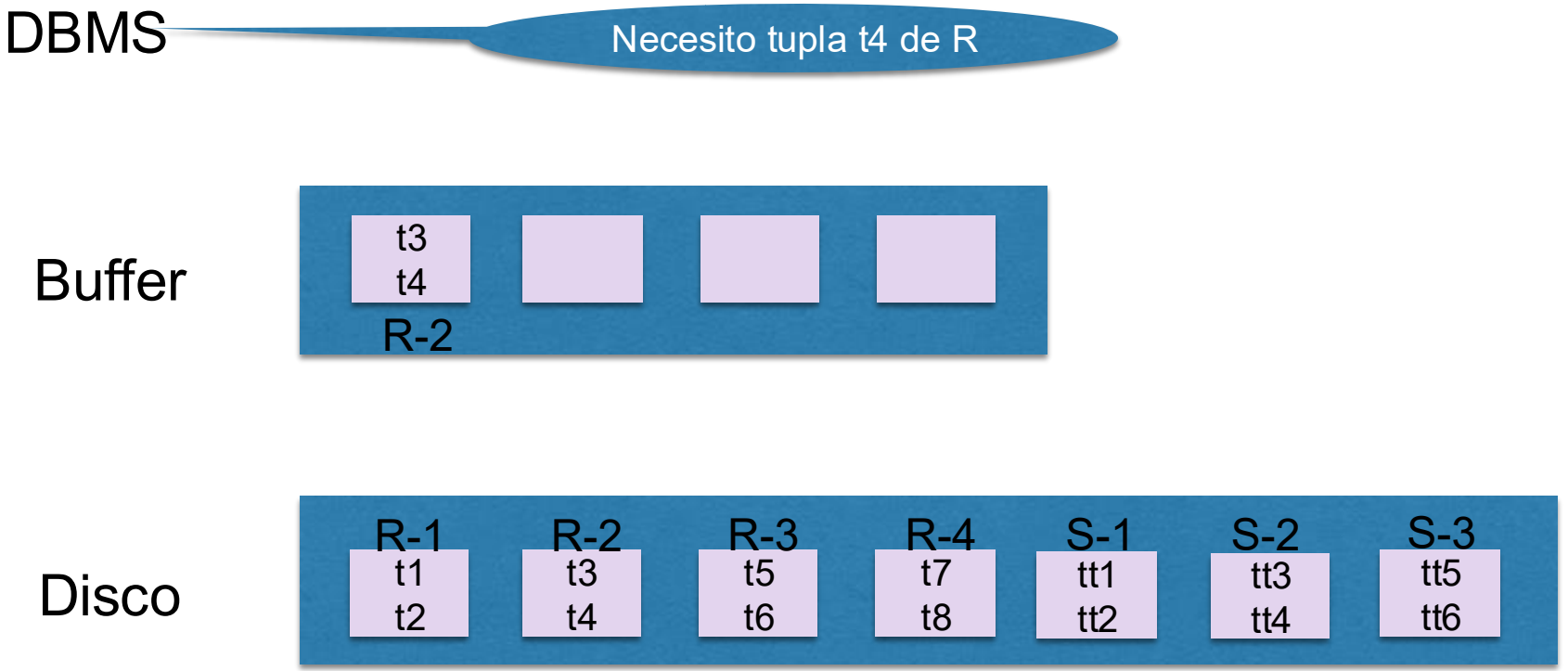
Buffer



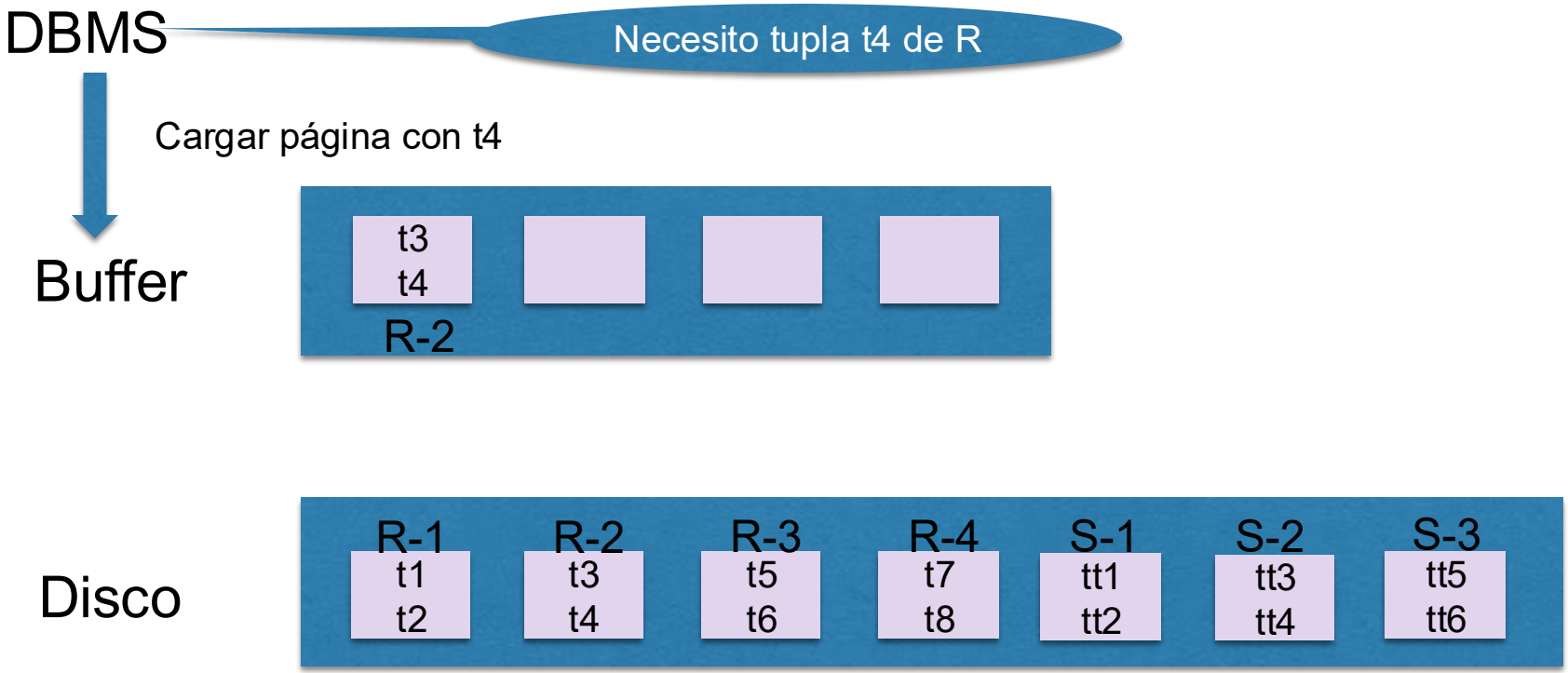
Disco



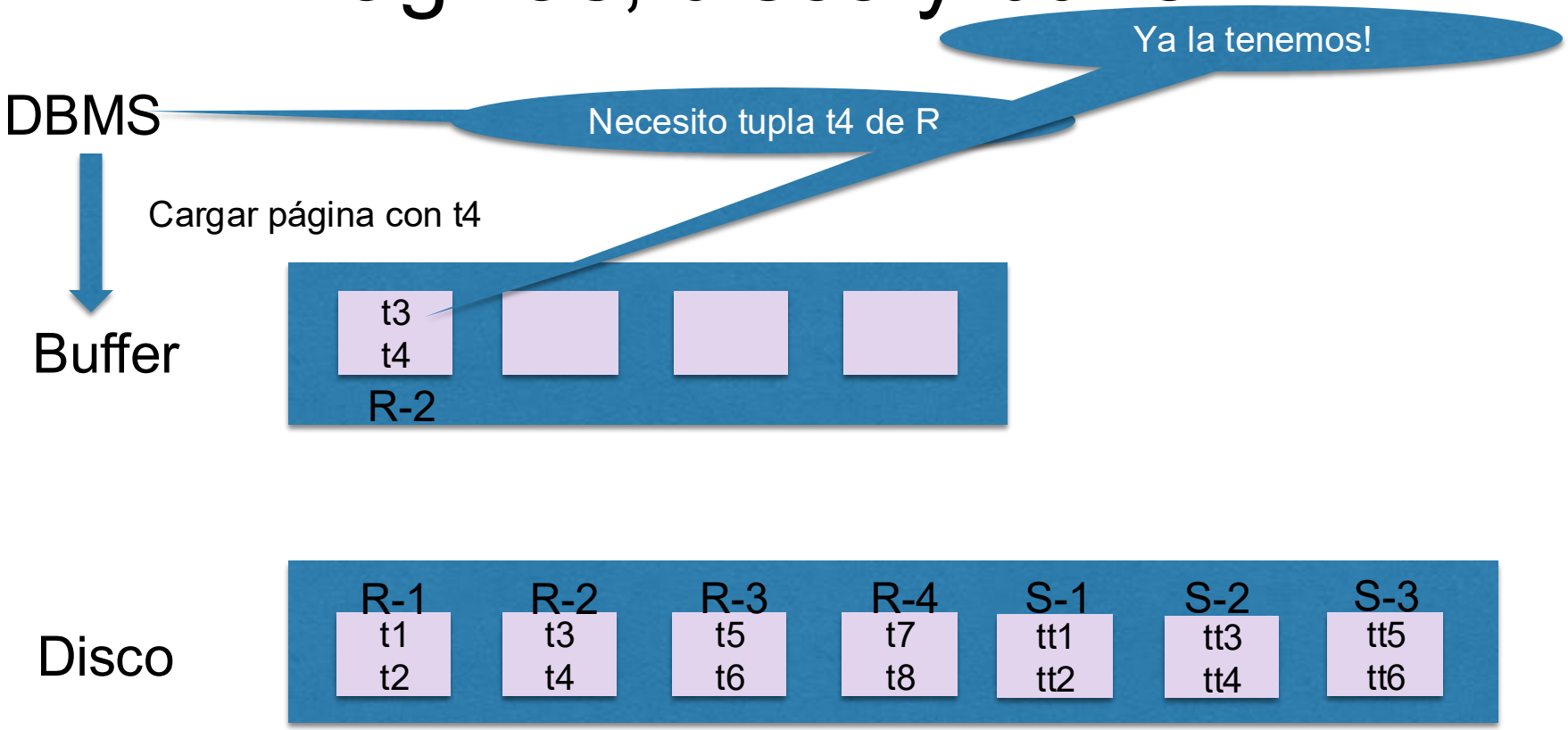
Páginas, disco y buffer



Páginas, disco y buffer



Páginas, disco y buffer



Algoritmos en Bases de Datos

Costo de un algoritmo

¿Cuántas veces tengo que leer una página desde el disco, o escribir una página al disco?

Las operaciones en buffer (RAM) son orden(es) de magnitud más rápidas que leer/escribir al disco – costo 0

Algoritmos en una BD

Los algoritmos implementan una interfaz de un iterador lineal:

- open()
- next()
- close()

Algoritmos en una BD

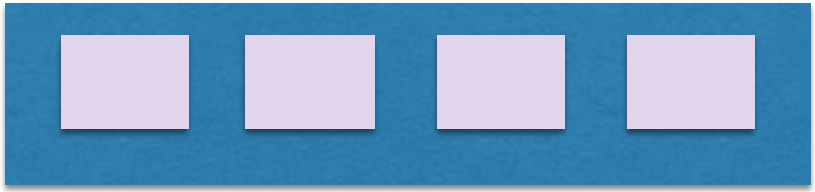
Para una relación R:

- R.open() – se posiciona **antes** de la primera tupla de R
- R.next() – devuelve la siguiente tupla o NULL
- R.close() – cierra el iterador

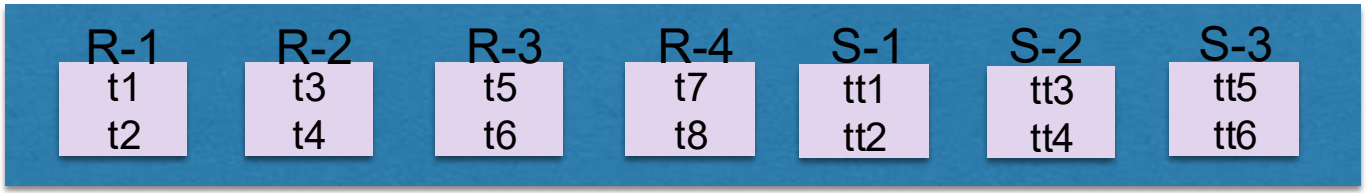
Páginas, disco y buffer

DB

Buffer



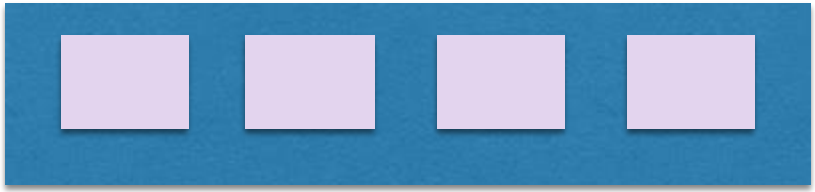
Disco



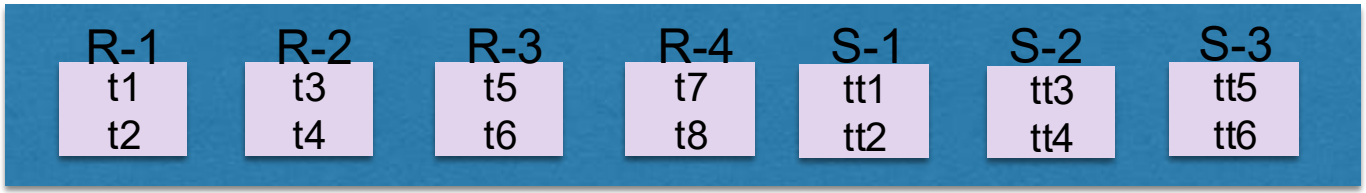
Páginas, disco y buffer

DBMS R.open()

Buffer



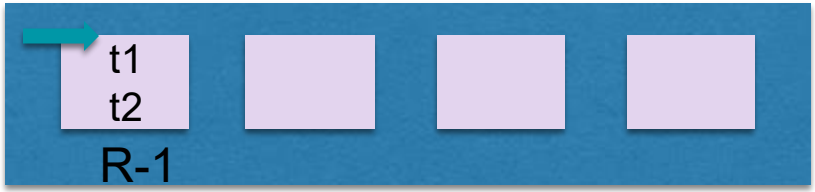
Disco



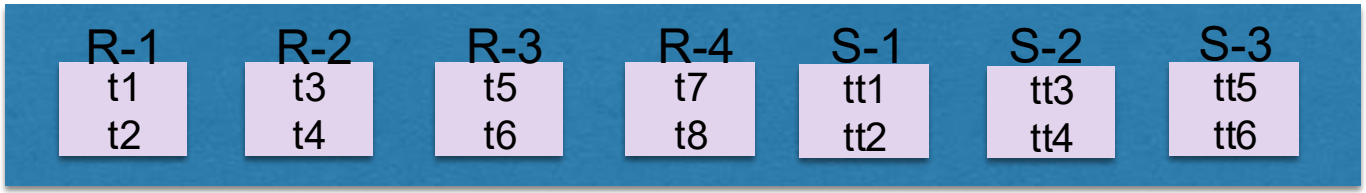
Páginas, disco y buffer

DBMS R.open()

Buffer



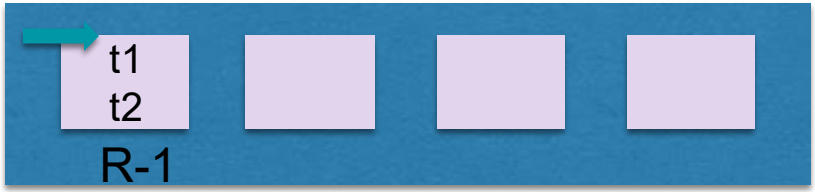
Disco



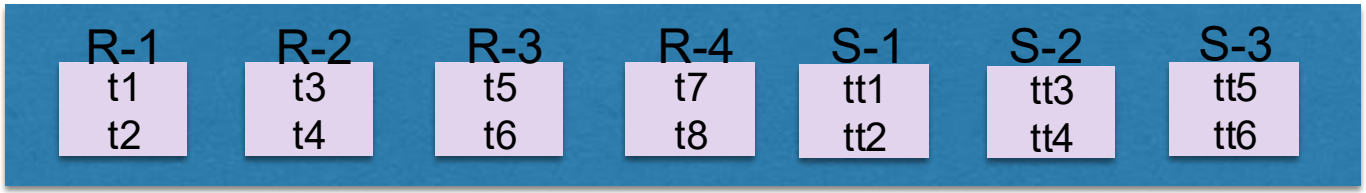
Páginas, disco y buffer

DBMS R.next()

Buffer



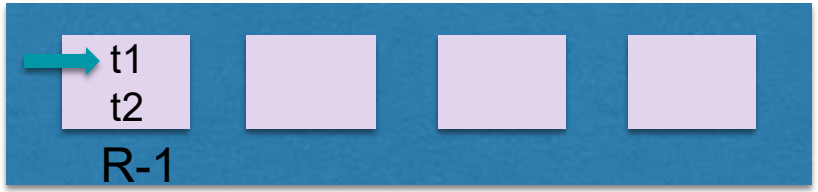
Disco



Páginas, disco y buffer

DBMS R.next()

Buffer

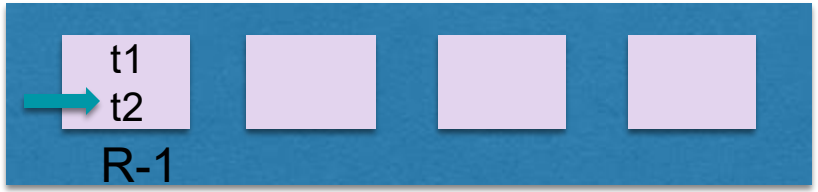


Disco

Páginas, disco y buffer

DBMS R.next()

Buffer

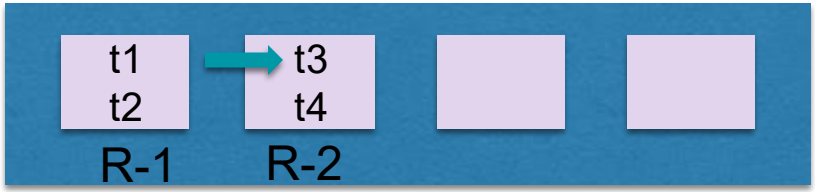


Disco

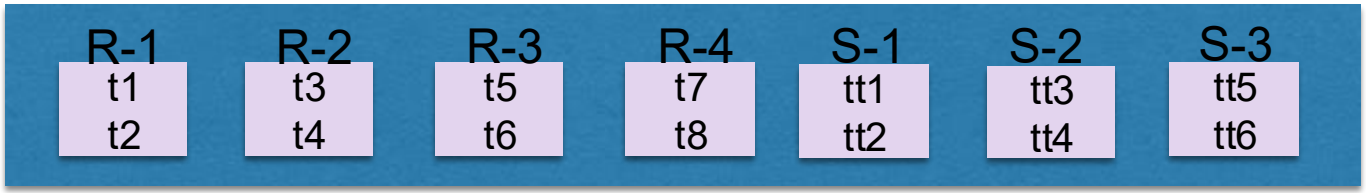
Páginas, disco y buffer

DBMS R.next()

Buffer



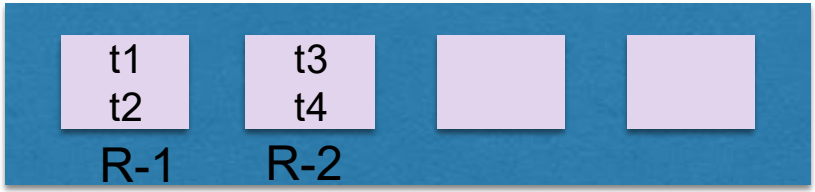
Disco



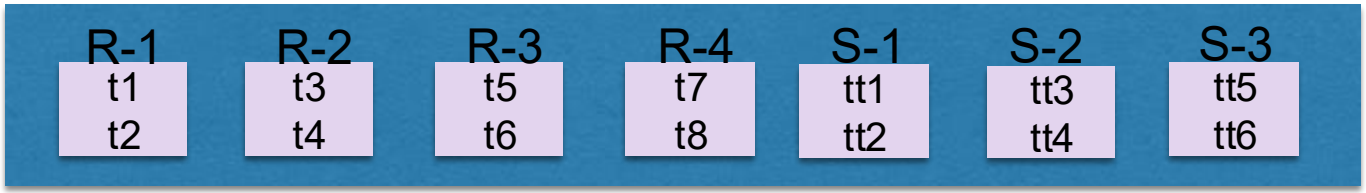
Páginas, disco y buffer

DBMS R.close()

Buffer



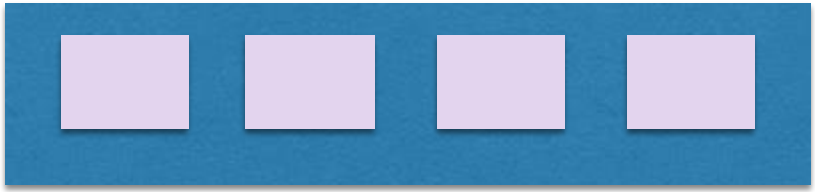
Disco



SELECT * FROM R

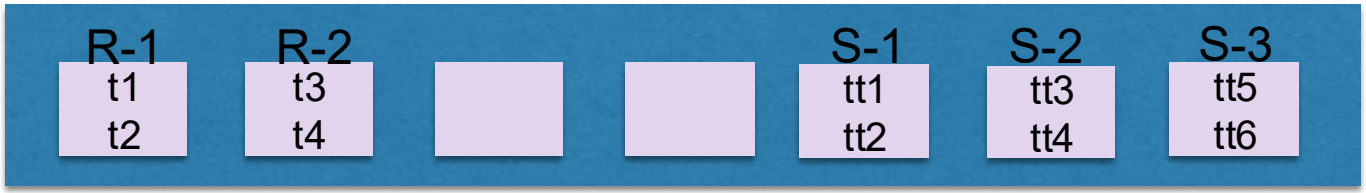
DBMS

Buffer



```
R.open()

 t:= R.next()
while t != null do
            output t
   t:= R.next()

R.close()
```

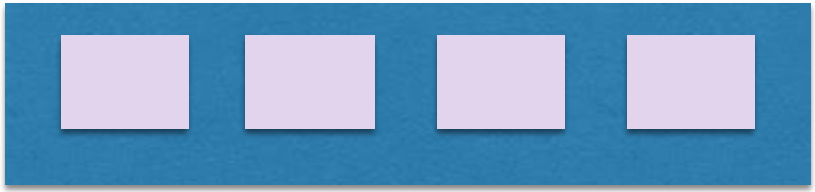
Disco



SELECT * FROM R

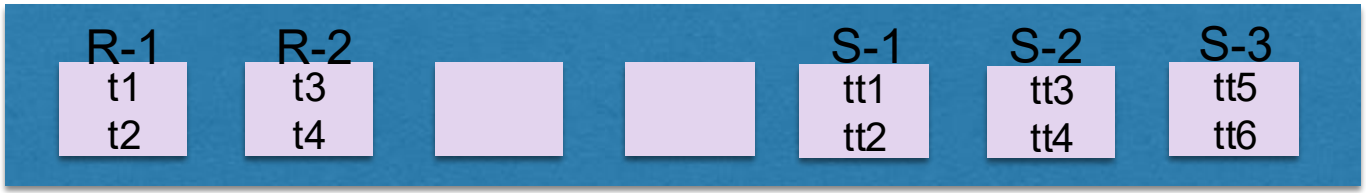
DBMS

Buffer



```
R.open()

 t:= R.next()
while t != null do
            output t
    t:= R.next()

R.close()
```

Disco

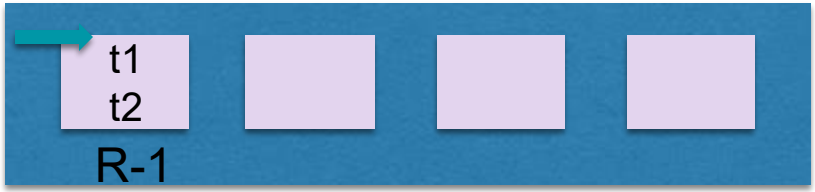


SELECT * FROM R

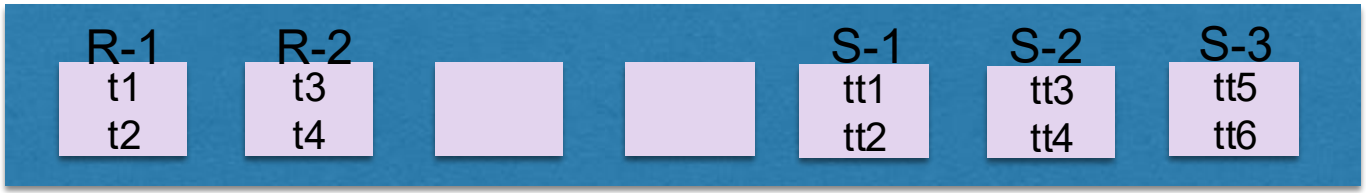
DBMS

```
R.open()

 t:= R.next()
while t != null do
            output t
    t:= R.next()

R.close()
```

Buffer



Disco

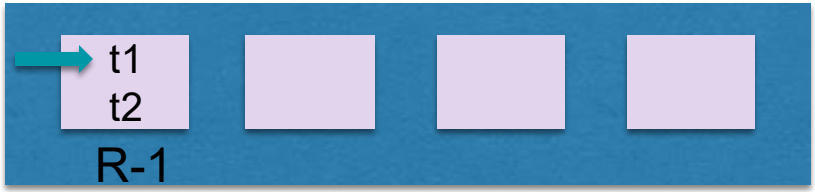


SELECT * FROM R

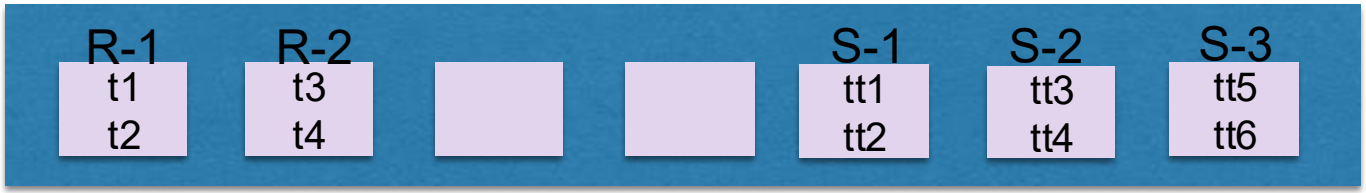
DBMS

```
R.open()

 t:= R.next()
while t != null do
            output t
    t:= R.next()

R.close()
```

Buffer



Disco

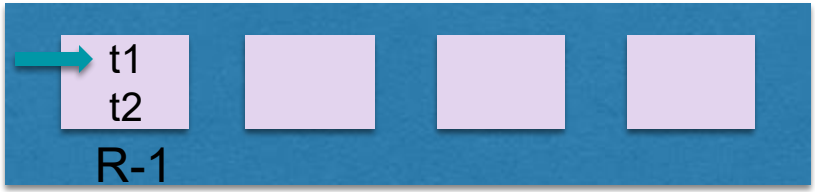


SELECT * FROM R

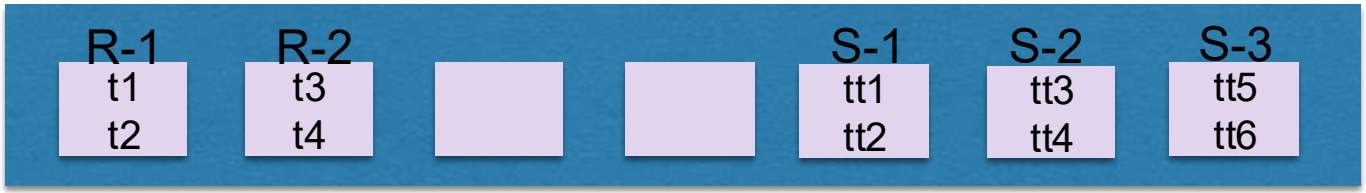
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

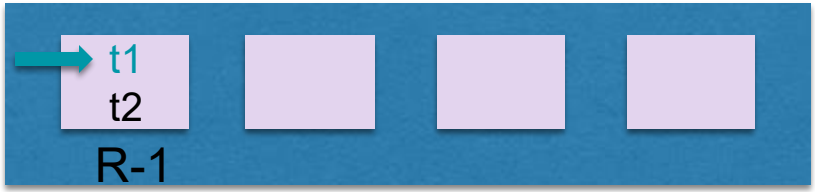


SELECT * FROM R

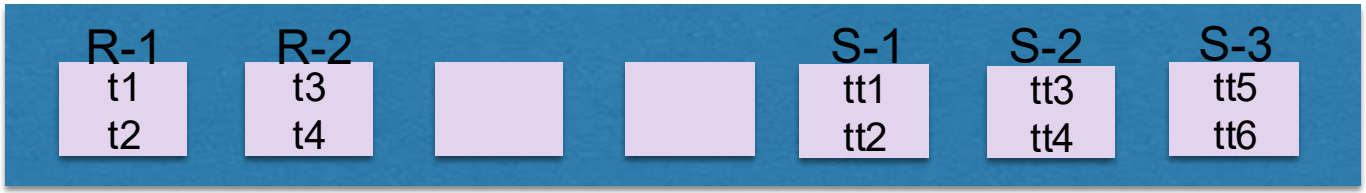
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

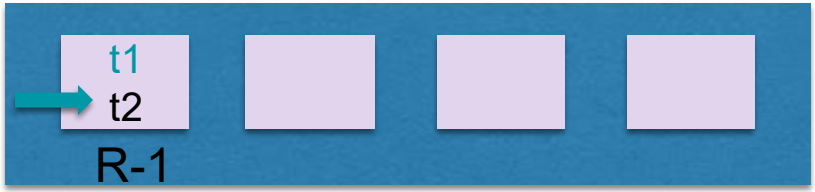


SELECT * FROM R

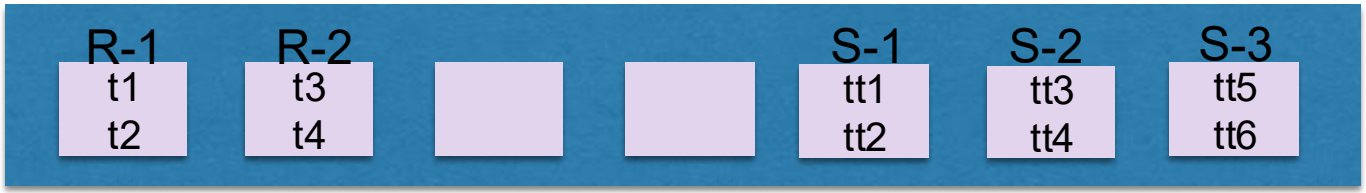
DBMS

```
R.open()

 t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

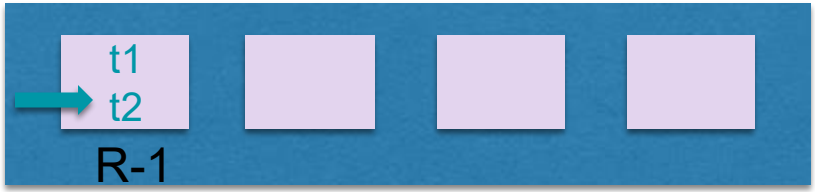


SELECT * FROM R

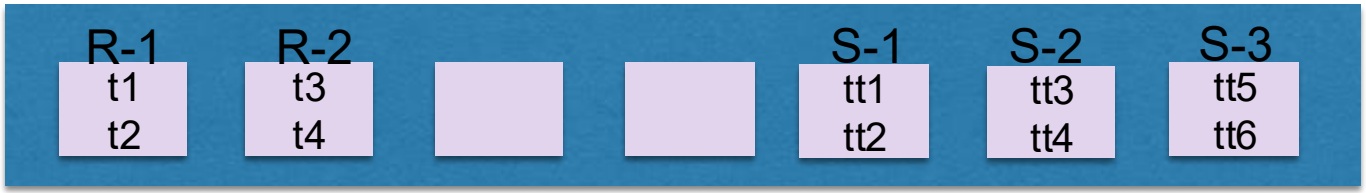
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

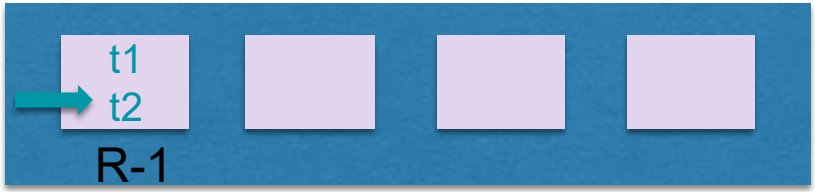


SELECT * FROM R

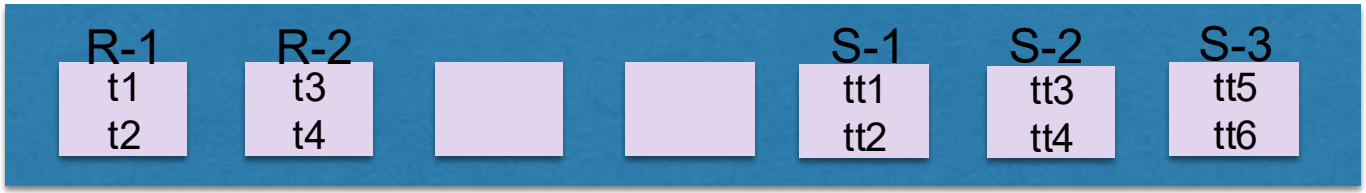
DBMS

```
R.open()

 t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

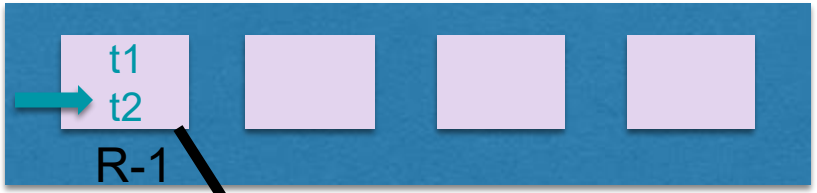


SELECT * FROM R

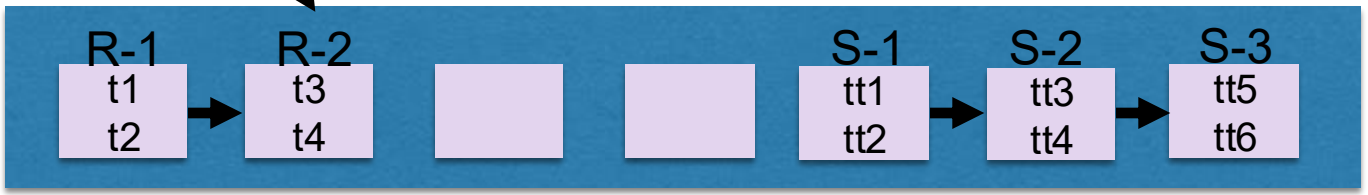
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

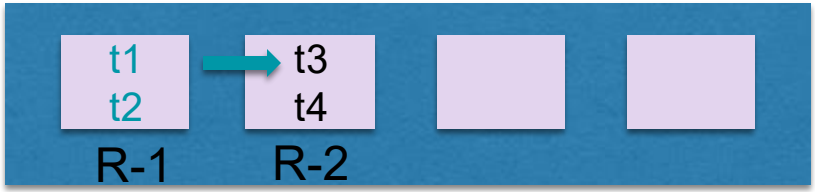


SELECT * FROM R

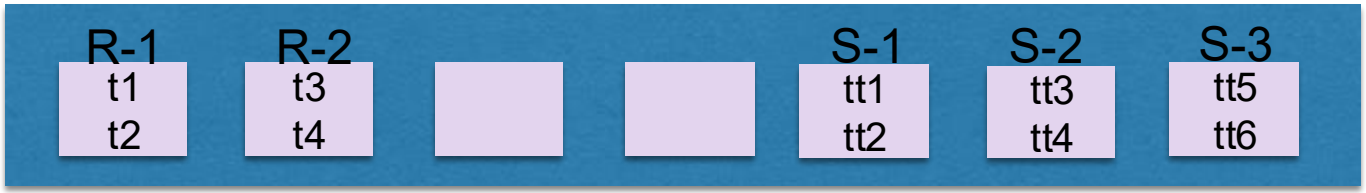
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

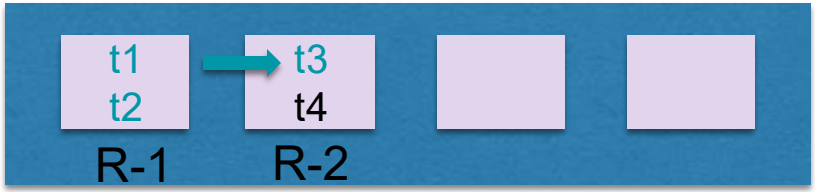


SELECT * FROM R

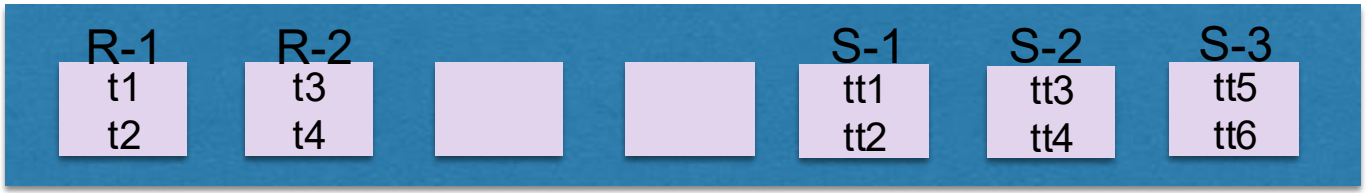
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco

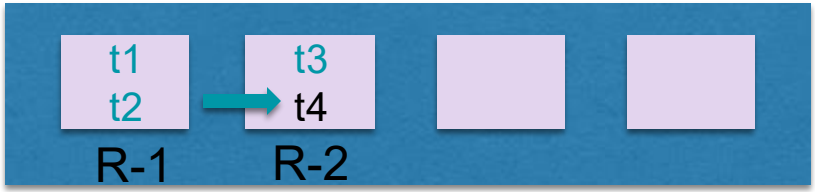


SELECT * FROM R

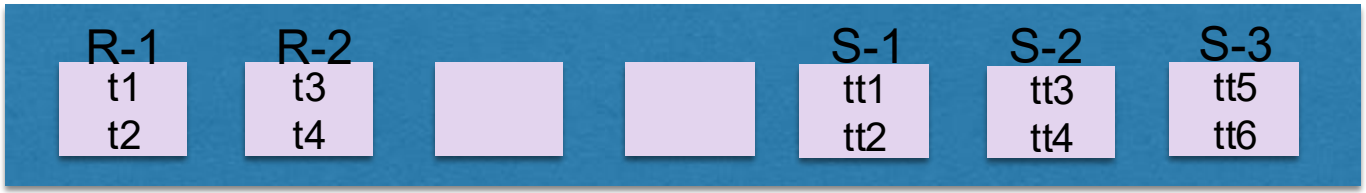
DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer



Disco



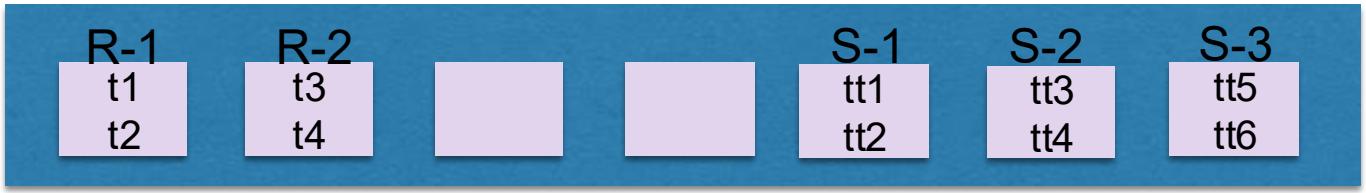
SELECT * FROM R

DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer

Disco



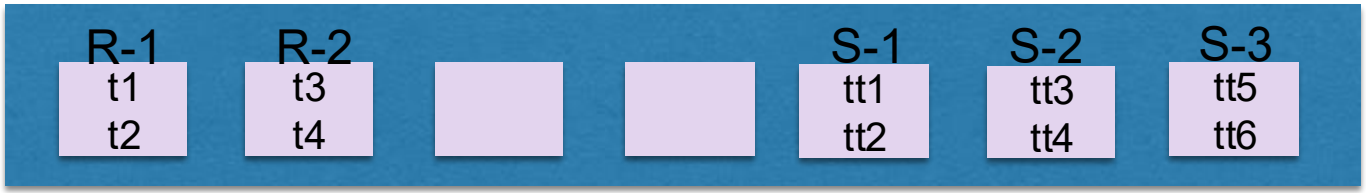
SELECT * FROM R

DBMS

```
R.open()

t:= R.next()
while t != null do
        output t
    t:= R.next()

R.close()
```

Buffer

Disco



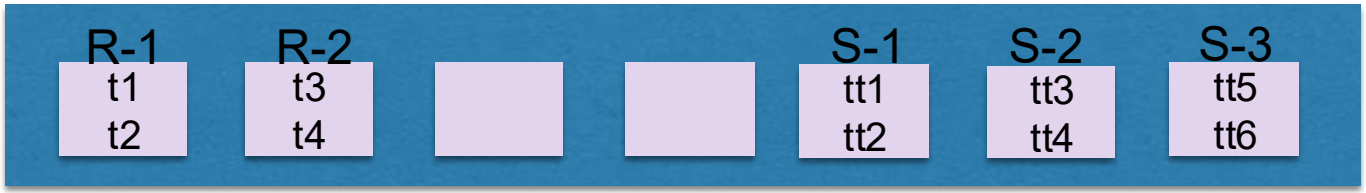
SELECT * FROM R

DBMS

```
R.open()

 t:= R.next()
while t != null do
            output t
    t:= R.next()

R.close()
```

Buffer

Disco



En realidad

Cada operador de algebra relacional implementa interfaz de un iterador lineal:

- open()
- next()
- close()

Algoritmos – Operadores Lineales

Selección

El algoritmo de selección cambia dependiendo si es una consulta de igualdad (=) o de rango (<, >)

También depende si el atributo a seleccionar está indexado

Este implementa la interfaz de iterador lineal

Selección

Sin índice:

Si queremos hacer una selección sobre una tabla **R**

```
open()
    R.open()


next() // retorna el siguiente seleccionado
    t:= R.next()
    while t != null do
        if t satisface condición then
            return t
        t:= R.next()
    return null
```

Selección

Sin índice

Si queremos hacer una selección sobre una tabla **R**

```
open()
    R.open()

next() // retorna el siguiente seleccionado
    t:= R.next()
    while t != null do
        if t satisface condición then
            return t
        t:= R.next()
    return null
close()
    R.close()
```

Para recorrer la selección Sel = σ_{cond} (**R**)

```
Sel.open()

t := Sel.next()
while t != null do
        output t
        t:= Sel.next()
Sel.close()
```

Selección

Sin indice, **necesariamente** tenemos que recorrer todo **R “FULL SCAN”**

Selección

Con índice y consulta de igualdad:

Si queremos hacer una selección sobre una tabla **R** a un atributo indexado con un índice **I**

```
open()
    I.open()
    I.search(Atributo = valor)

next()
    t:= I.next()
    while t != null do
        return t
    return null

close()
    I.close()
```

Selección

Con índice y consulta de igualdad:

Sólo tenemos que leer las páginas que satisfacen la condición (más I/O si muchas tuplas satisfacen la condición)

Cambia un poco si el índice es Clustered o Unclustered (¿Por qué?)

Si el atributo es llave primaria entonces la operación prácticamente tiene I/O ~ 1

Selección

¿Cómo podemos hacer este tipo de consultas de forma eficiente?

Usando **Proyección**

Proyección

Algoritmo muy sencillo

```
open()
    R.open()

next()
    t:= R.next()
    while t != null do
        return project(t, atributos)
    return null

close()
    R.close()
```

Proyección

Necesariamente tenemos que recorrer todo **R**

Joins

Operación muy costosa

Supondremos solamente restricciones de igualdad (por ejemplo, R.a = S.a)

Nested Loop Joins

Nested Loop Join

Queremos hacer un join entre **R** y **S**, cuando se satisface un predicado **p**

```
open()
    R.open()
    S.open()
    r:= R.next()

close()
    R.close()
    S.close()
```

Nested Loop Join

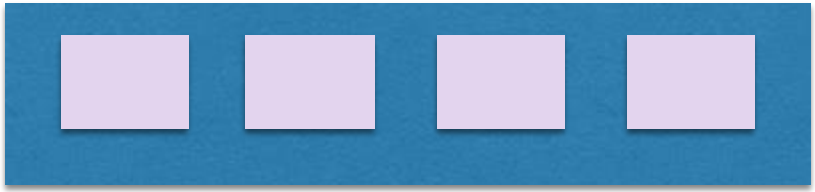
Queremos hacer un join entre **R** y **S**, cuando se satisface un predicado **p**

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if (r, s) satisfacen p then
            return (r, s)
    return null
```

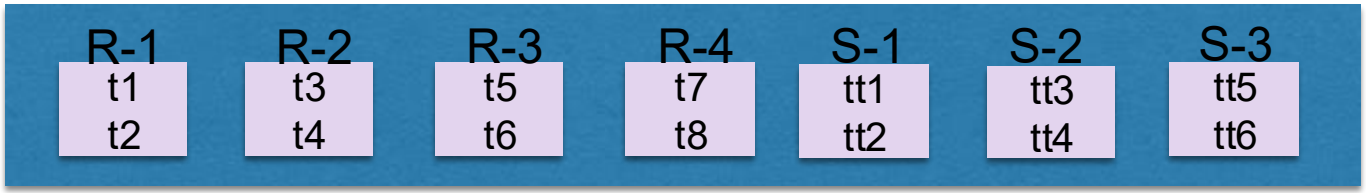
Nested Loop Join

DB

Buffer



Disco




```
Join = R⋈pS

Join.open()

 t:= Join.next()
while t != null do
            output t
    t:= Join.next()

Join.close()
```

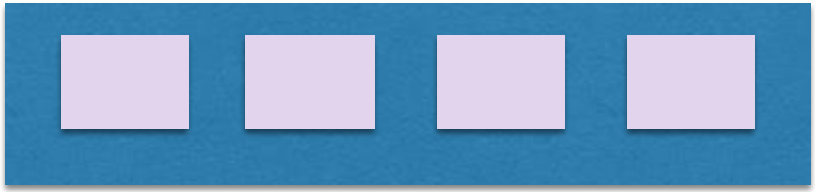
Nested Loop Join

DB

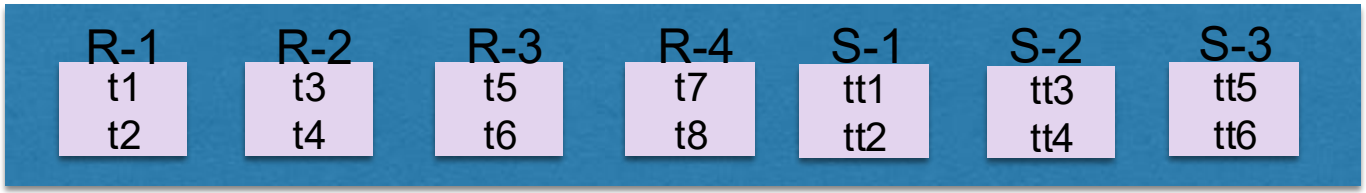
$R \bowtie_p S$ …Join.open()

```
R.open()
S.open()
r:= R.next()
```

Buffer



Disco



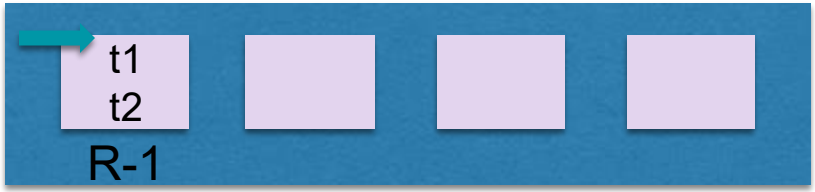
Nested Loop Join

DB

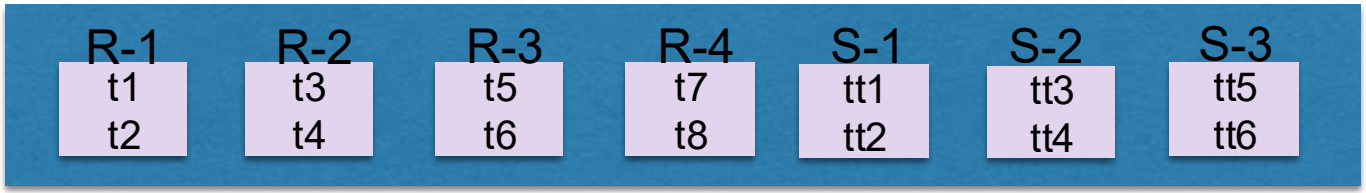
$R \bowtie_p S$ …Join.open()

```
R.open()
S.open()
r:= R.next()
```

Buffer



Disco



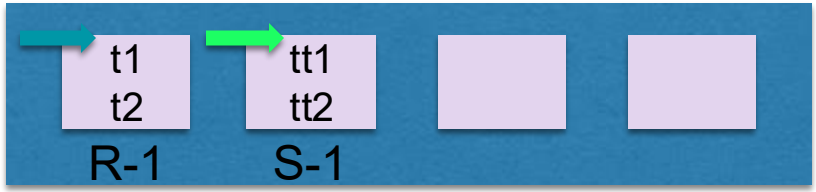
Nested Loop Join

DB

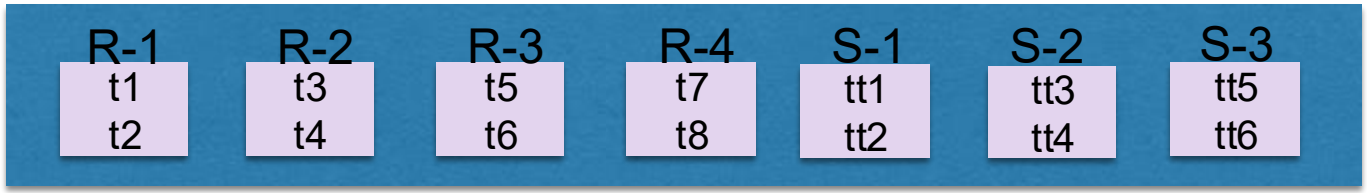
$R \bowtie_p S$ …Join.open()

```
R.open()
S.open()
r:= R.next()
```

Buffer



Disco



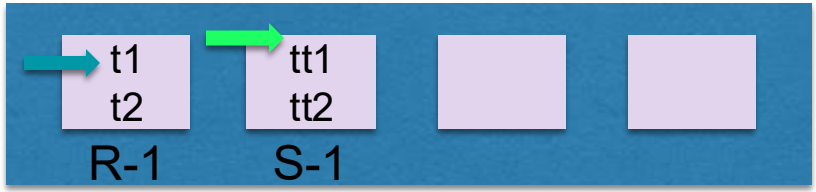
Nested Loop Join

DB

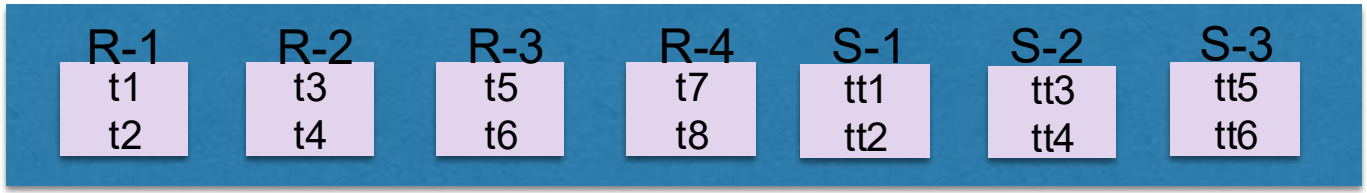
$R \bowtie_p S$ …Join.open()

```
R.open()
S.open()
r:= R.next()
```

Buffer



Disco

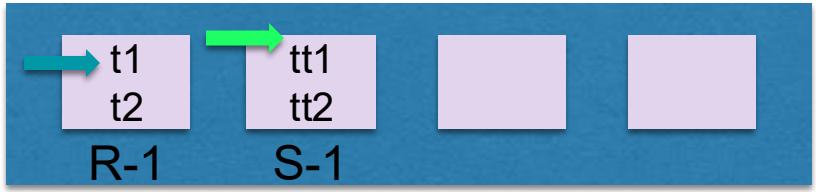


Nested Loop Join

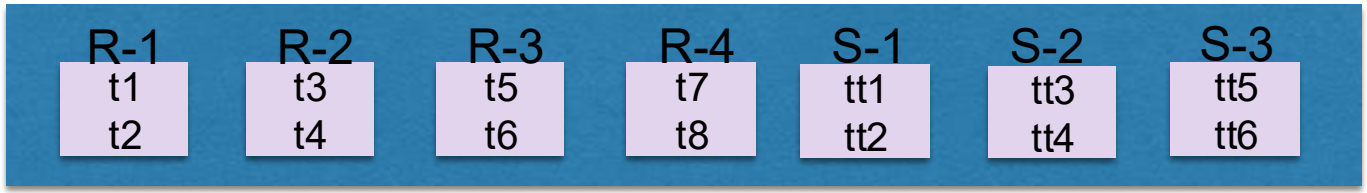
DB

```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

Buffer



Disco



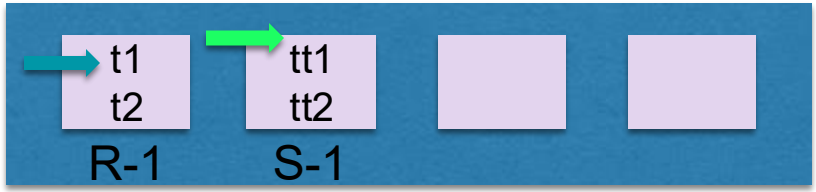
Nested Loop Join

DB

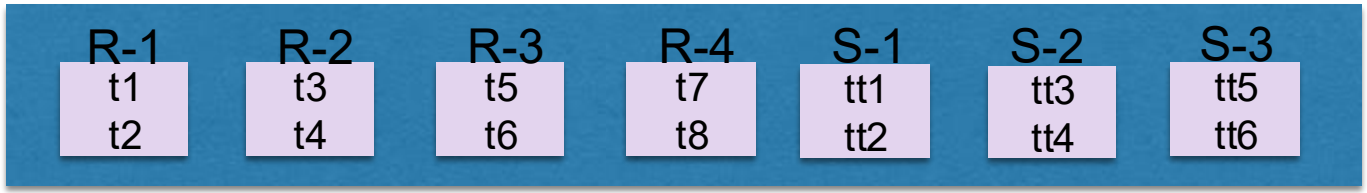
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



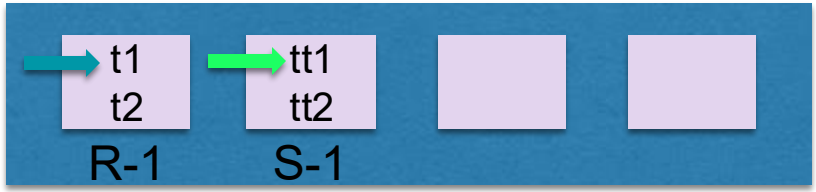
Nested Loop Join

DB

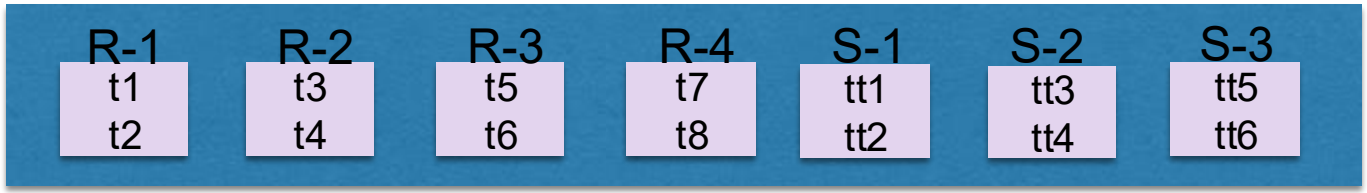
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



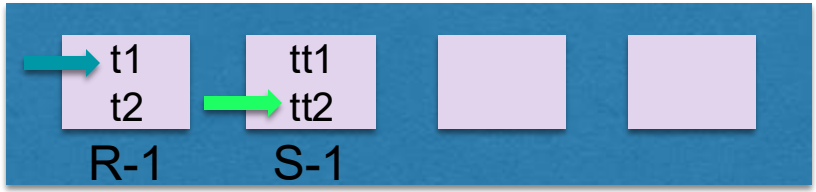
Nested Loop Join

DB

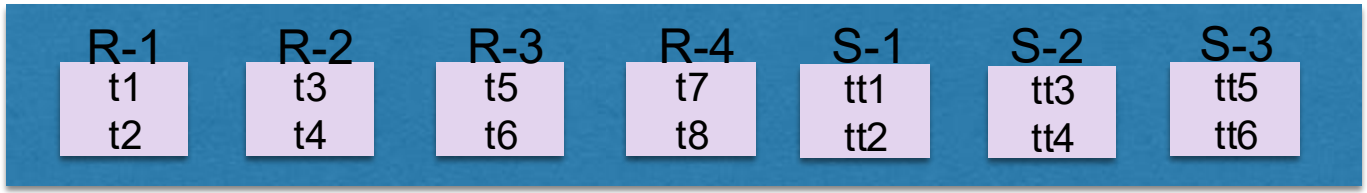
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



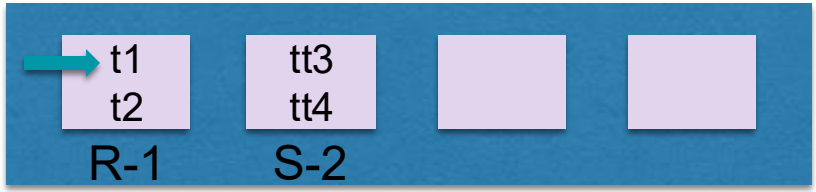
Nested Loop Join

DB

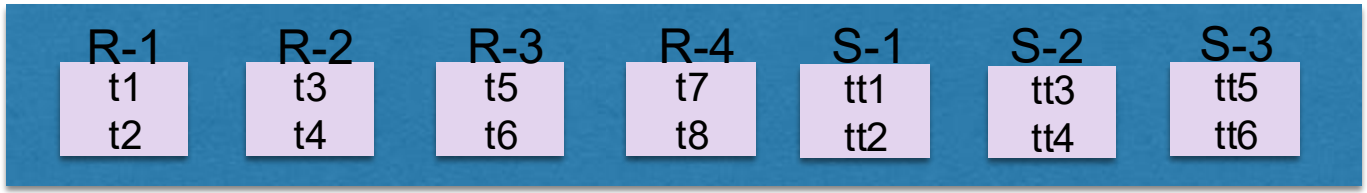
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



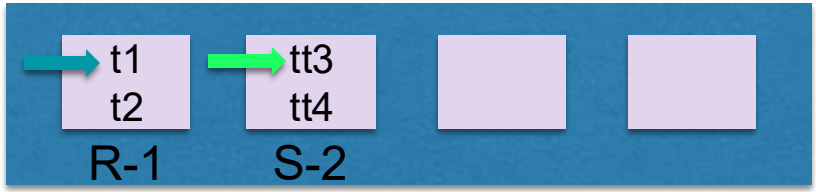
Nested Loop Join

DB

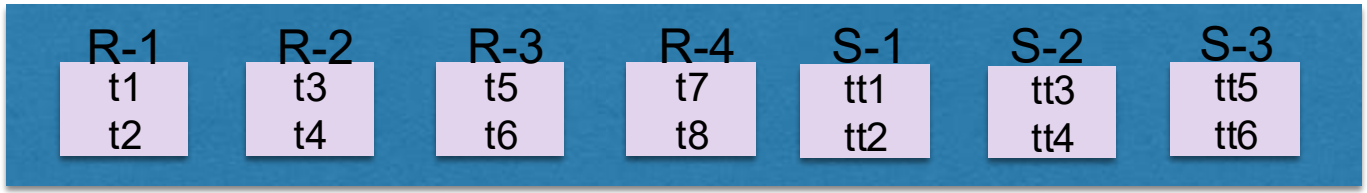
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



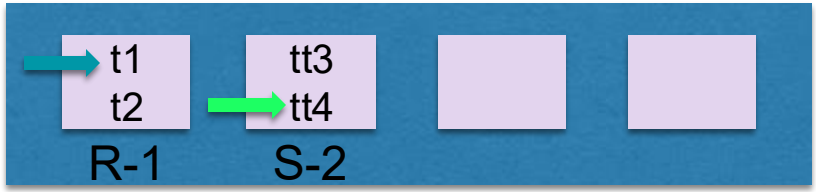
Nested Loop Join

DB

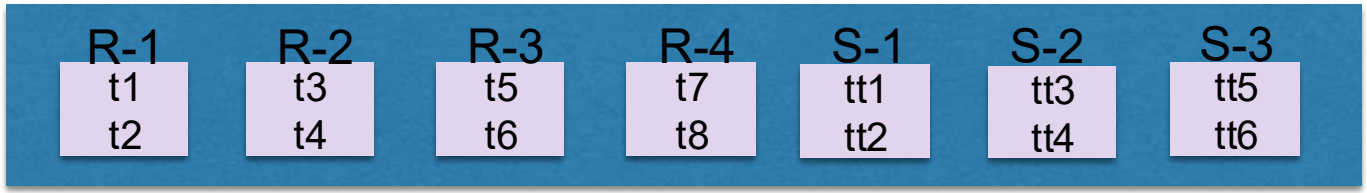
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



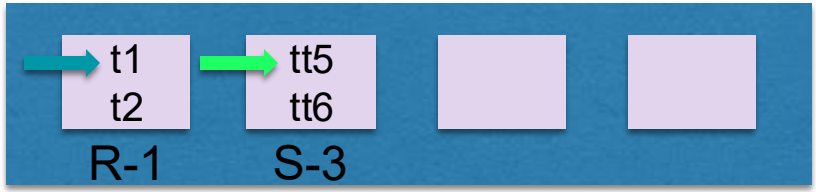
Nested Loop Join

DB

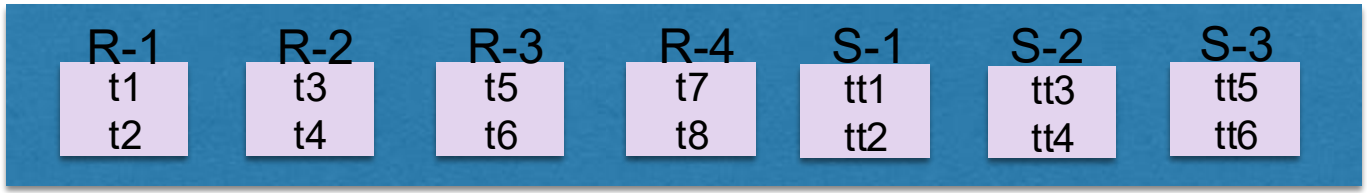
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



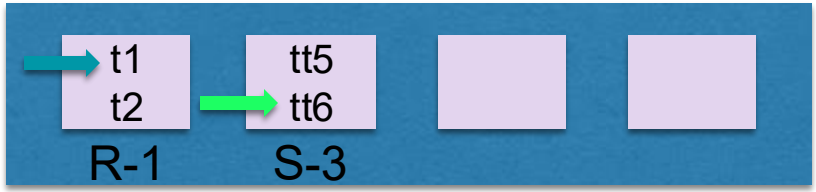
Nested Loop Join

DB

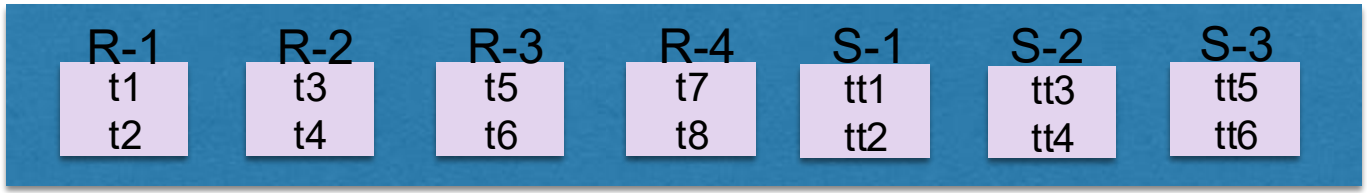
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

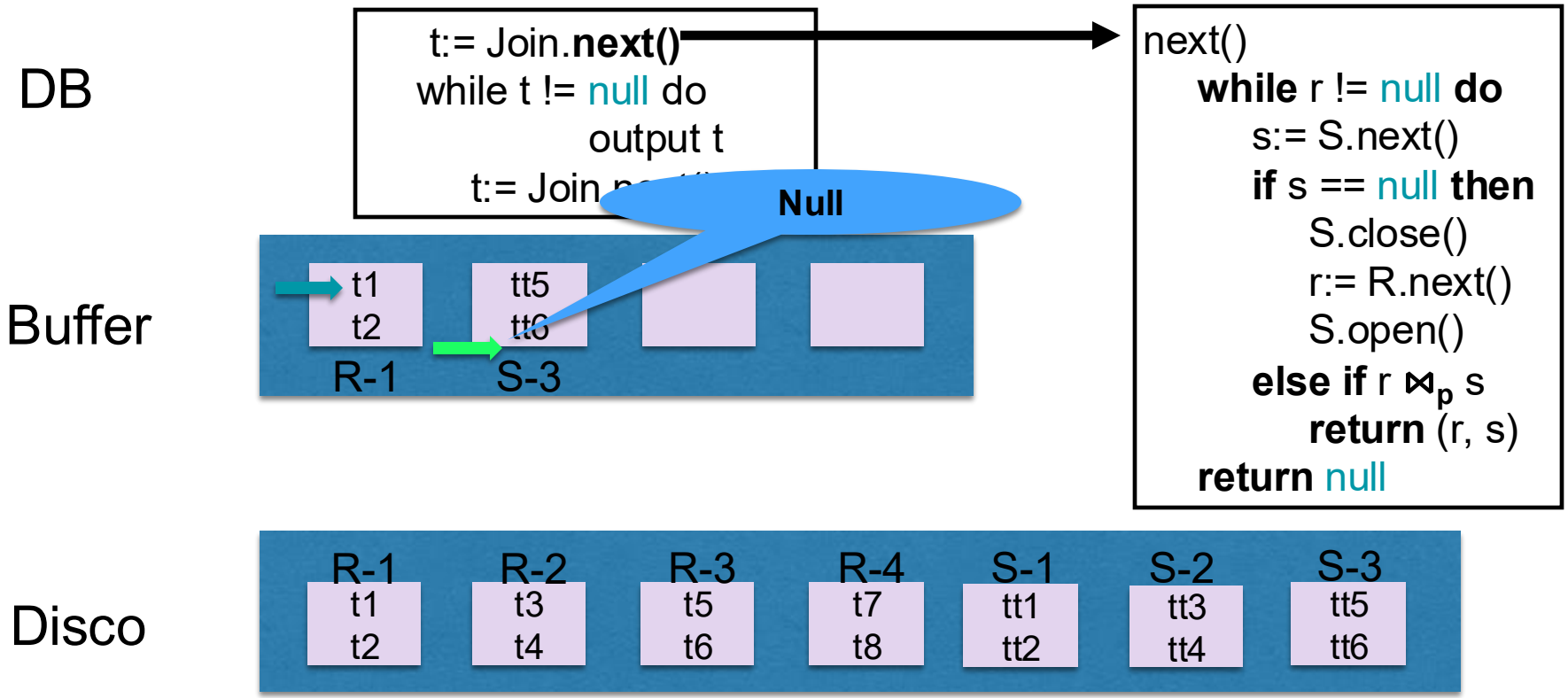
Buffer



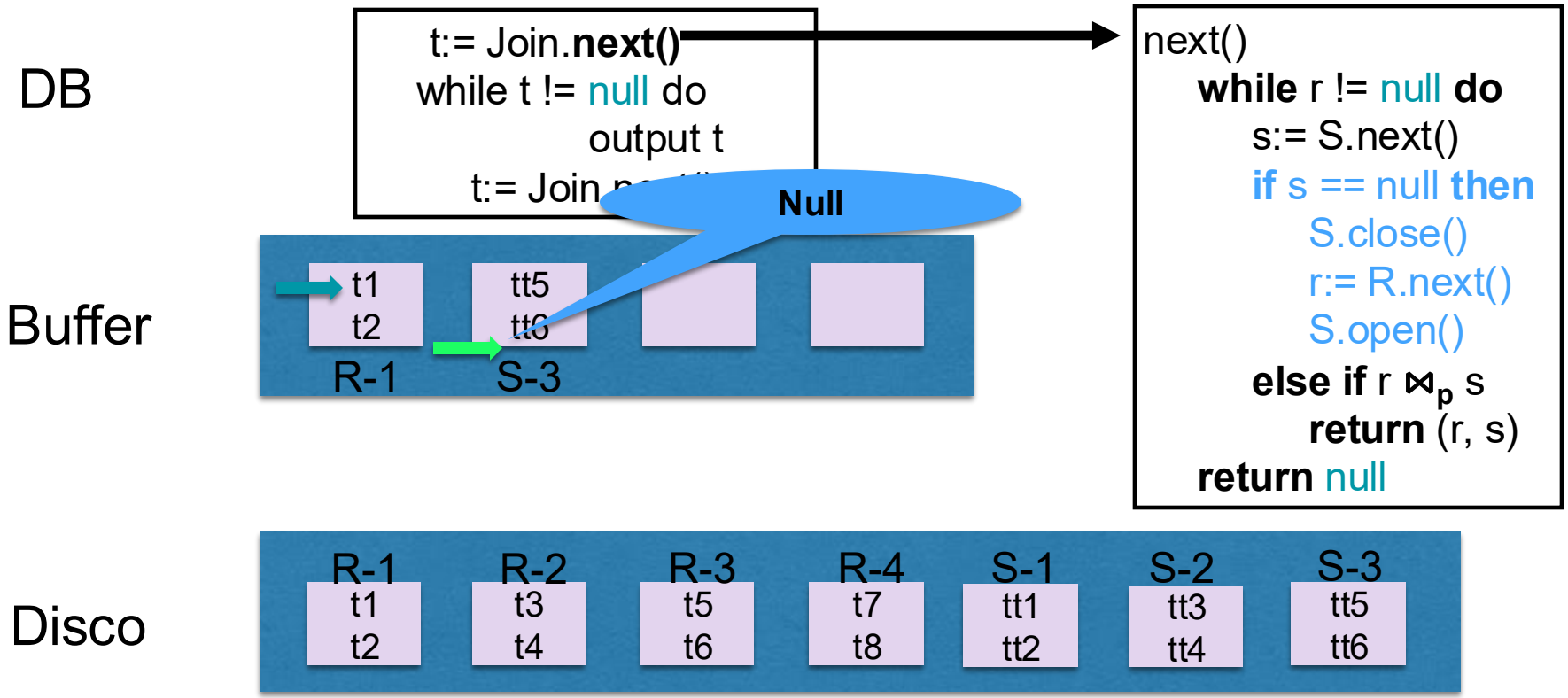
Disco



Nested Loop Join



Nested Loop Join



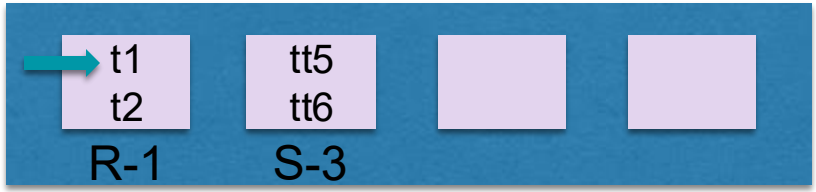
Nested Loop Join

DB

```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco

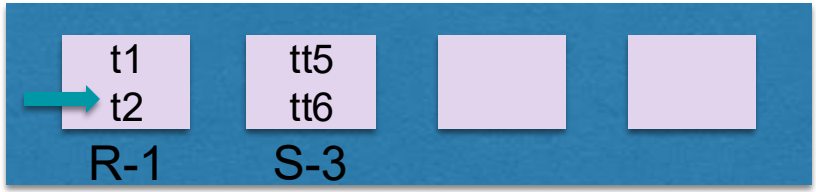
Nested Loop Join

DB

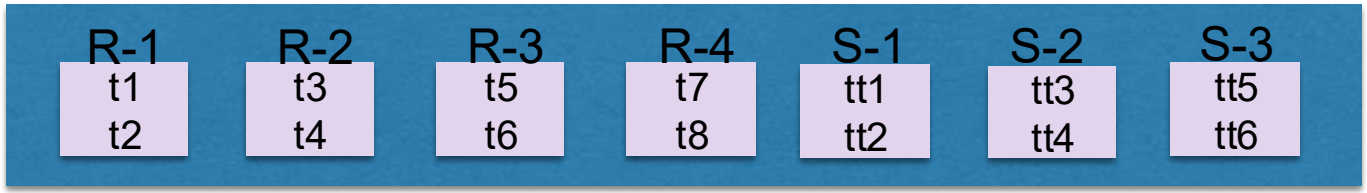
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



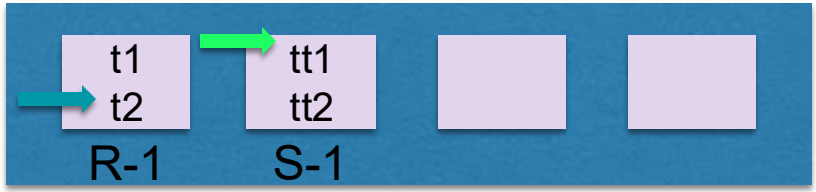
Nested Loop Join

DB

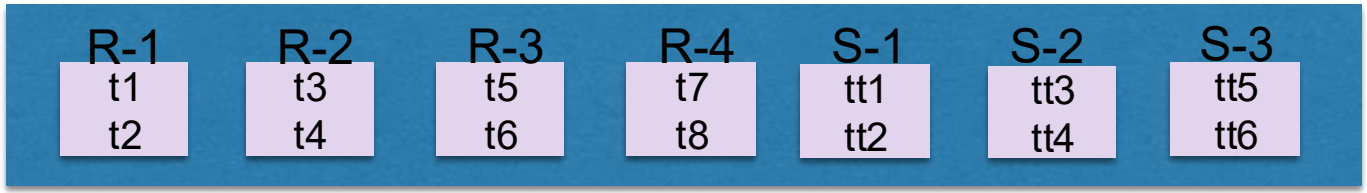
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



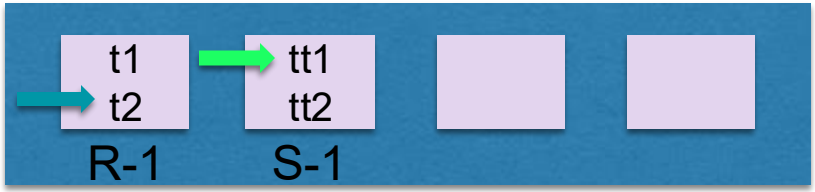
Nested Loop Join

DB

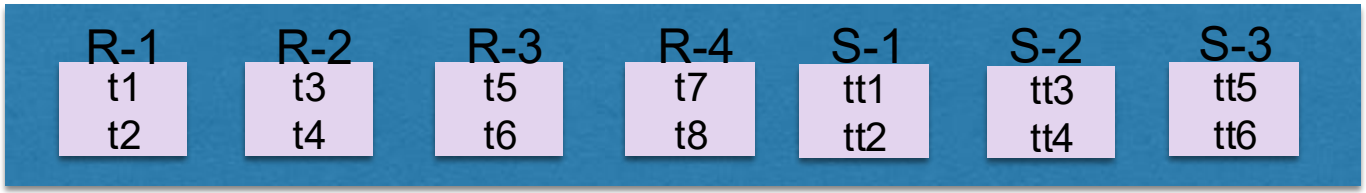
```
t:= Join.next()
while t != null do
        output t
    t:= Join.next()
```

```
next()
    while r != null do
        s:= S.next()
        if s == null then
            S.close()
            r:= R.next()
            S.open()
        else if r ⋈p s
            return (r, s)
    return null
```

Buffer



Disco



Nested Loop Join

Es una implementación directa basada en un loop

Para cada tupla de **R** debemos leer **S** entera, aparte de leer **R** entera una vez

Costo en I/O es:

$$\text{Costo}(\mathbf{R}) + \text{Tuplas}(\mathbf{R}) \cdot \text{Costo}(\mathbf{S})$$

Nested Loop Join

Si **R** y **S** son tablas de 16 MB, cada página es de 8 KB y las tuplas son de 300 bytes

Cada relación tiene 2048 páginas y 55.000 tuplas aproximadamente

Costo de un I/O es 0.1 ms, entonces el join tarda:

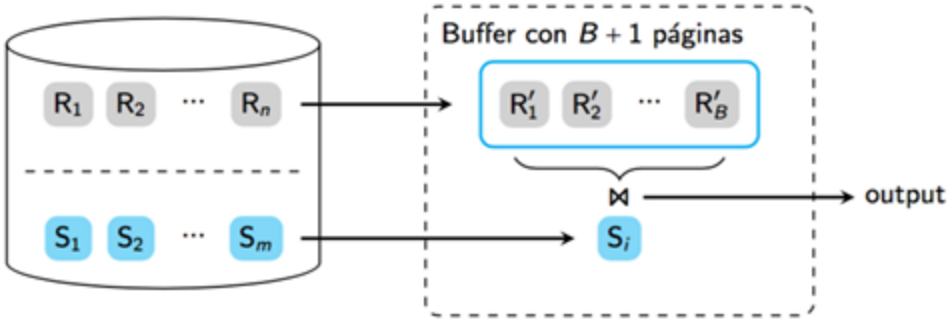
3.1 horas

¿Cómo podemos optimizar esto? Con **Block Nested Loop Joins**

Block Nested Loop Join

Block Nested Loop Join

Aprovechamos mejor el buffer



Block Nested Loop Join

Queremos hacer un join entre **R** y **S**, cuando se satisface un predicado **p**

```
open()
    R.open()
    fillBuffer()

close()
    R.close()
    S.close()
```

```
fillBuffer()
    Buff = empty
    r:= R.next()
    while r != null do
        Buff = Buff union r
        if Buff.isFull() then
            break
        r:= R.next()
    S.open()
    S.next()
```

Block Nested Loop Join

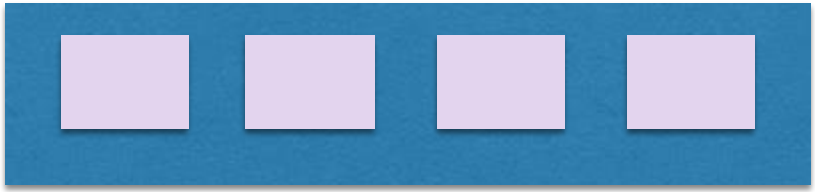
Queremos hacer un join entre **R** y **S**, cuando se satisface un predicado **p**

```
next()
    while Buff != empty do
        while s != null do
            r:= Buffer.next()
            if r == null then
                Buffer.reset()
                s:= S.next()
            else if (r,s) satisfacen p then
                return (r,s)
        fillBuffer()
    return null
```

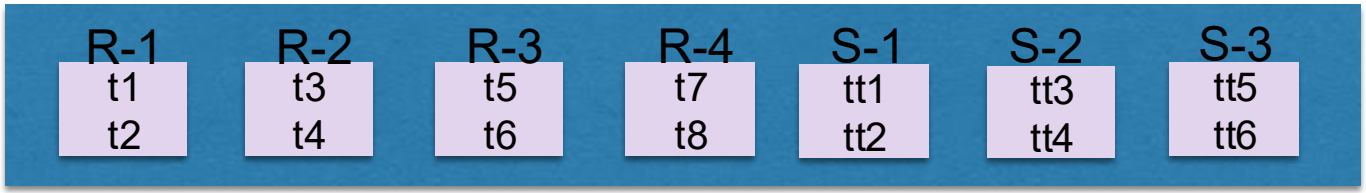
Block Nested Loop Join

DB $R \bowtie_p S$

Buffer



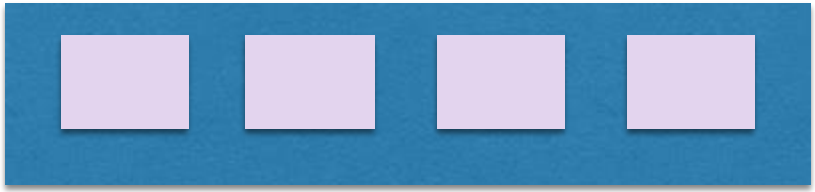
Disco



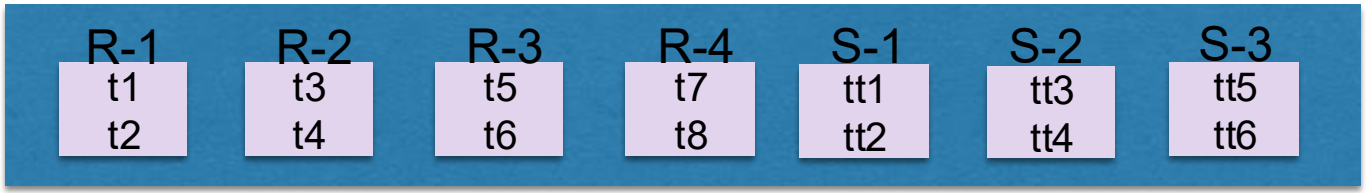
Block Nested Loop Join

DB $R \bowtie_p S$.open()

Buffer



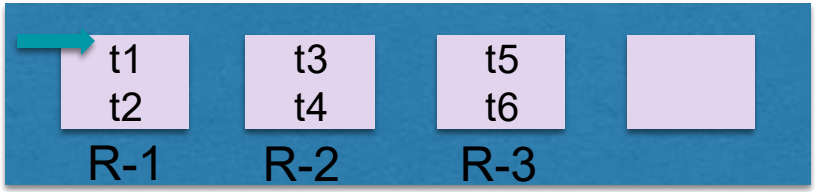
Disco



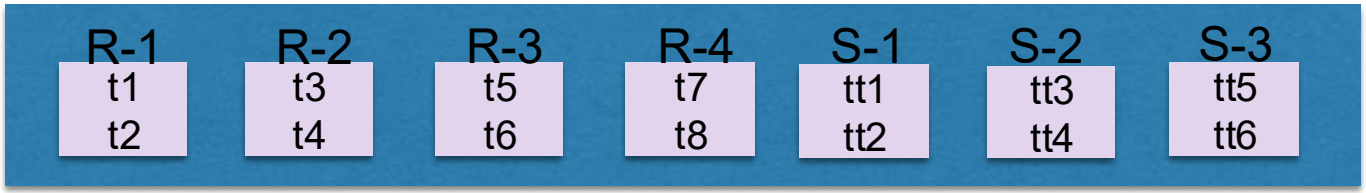
Block Nested Loop Join

DB $R \bowtie_p S$.open()

Buffer



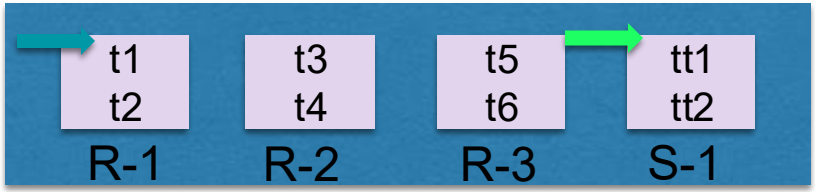
Disco



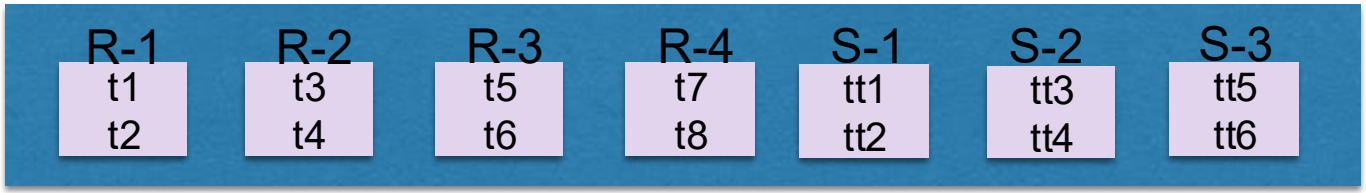
Block Nested Loop Join

DB $R \bowtie_p S$.open()

Buffer



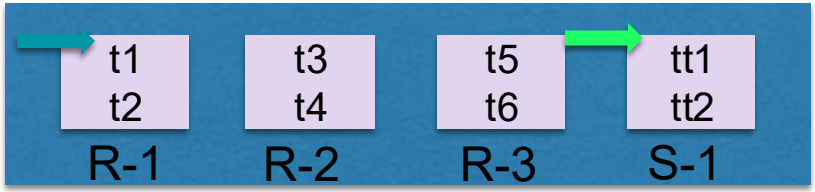
Disco



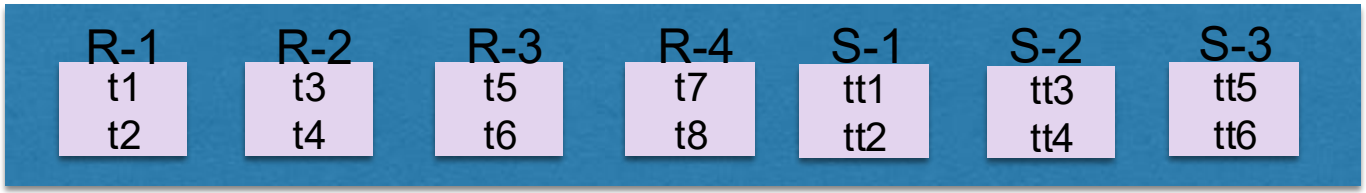
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



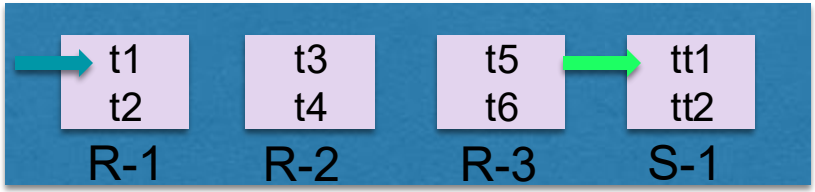
Disco



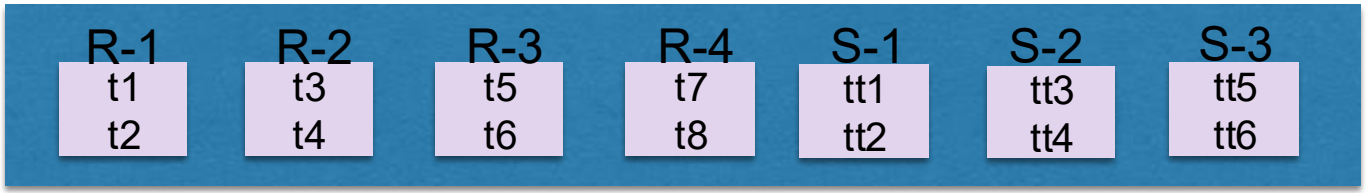
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



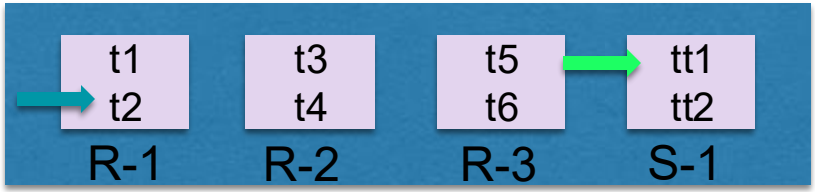
Disco



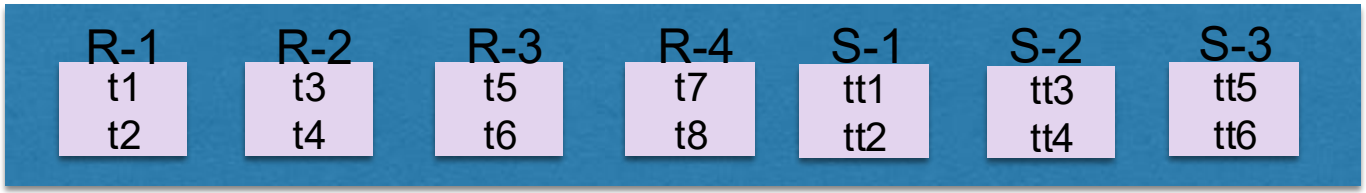
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



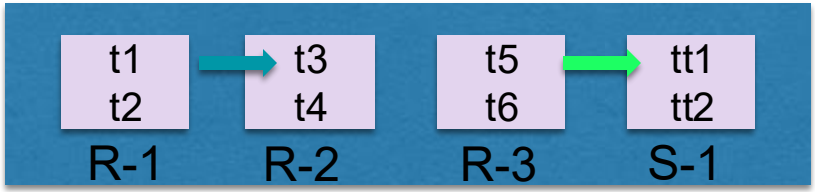
Disco



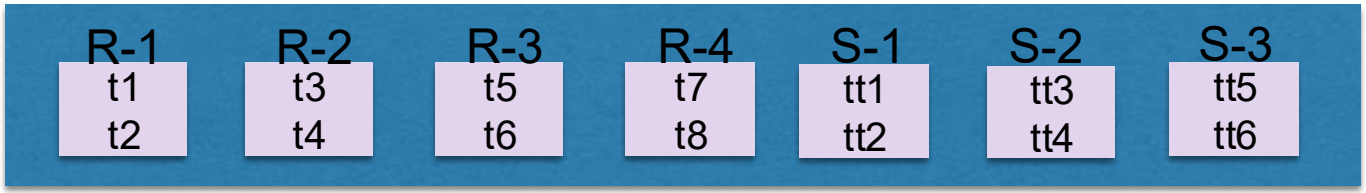
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



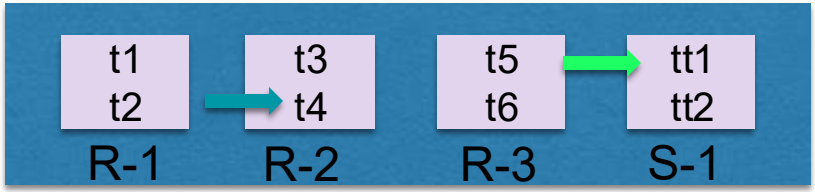
Disco



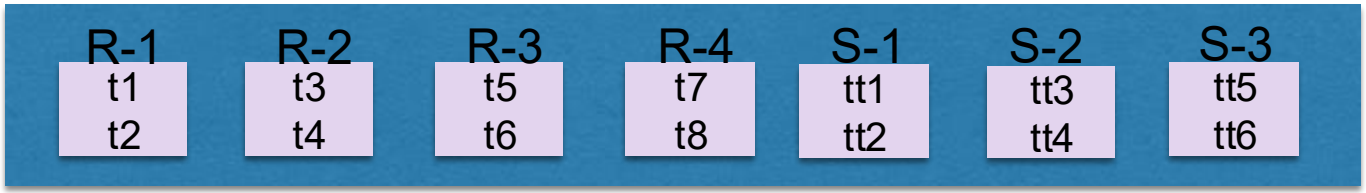
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



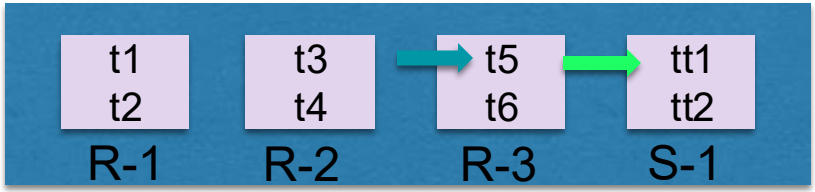
Disco



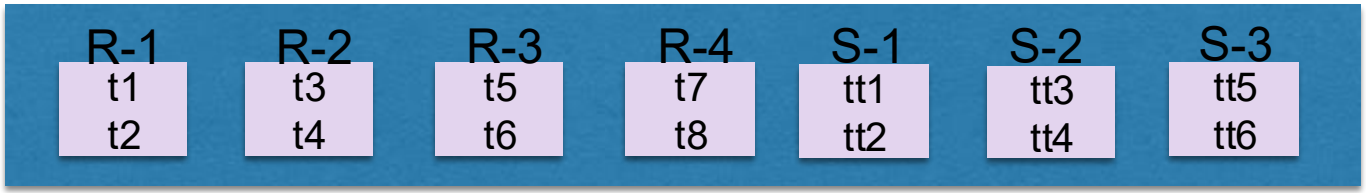
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



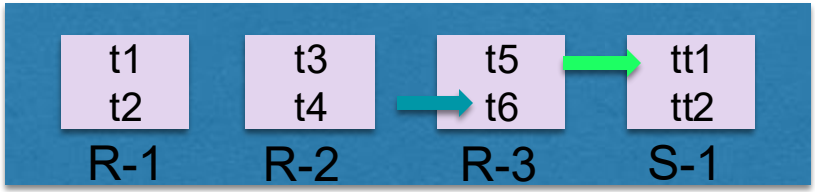
Disco



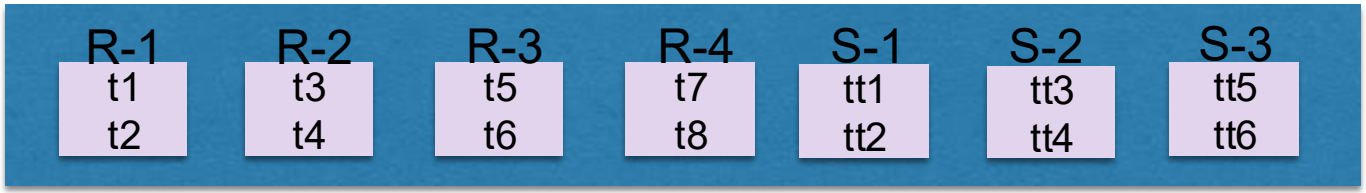
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer

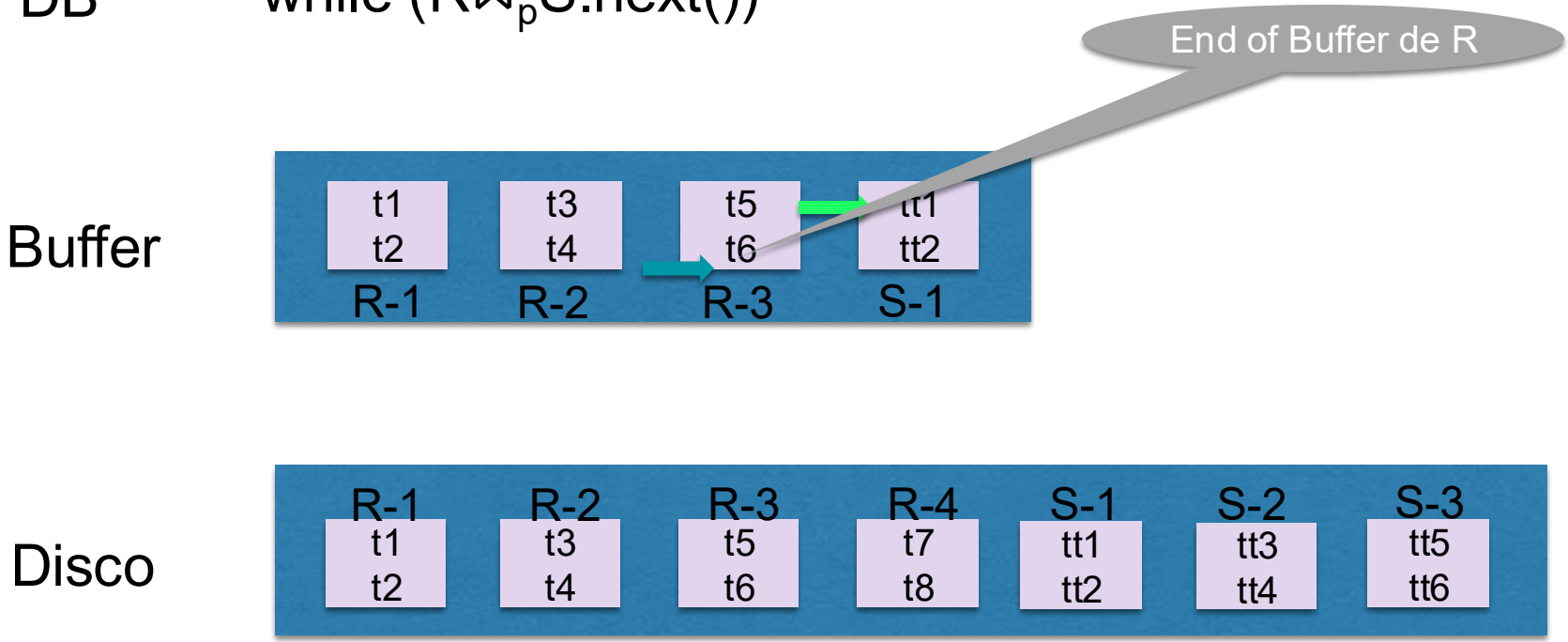


Disco



Block Nested Loop Join

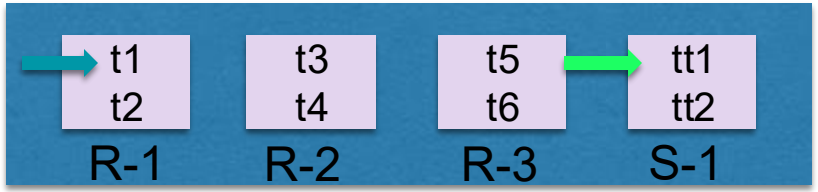
DB while ($R \bowtie_p S$.next())



Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer

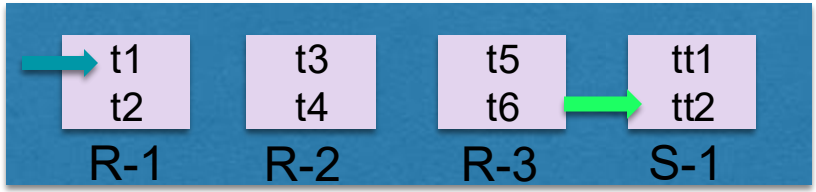


Disco

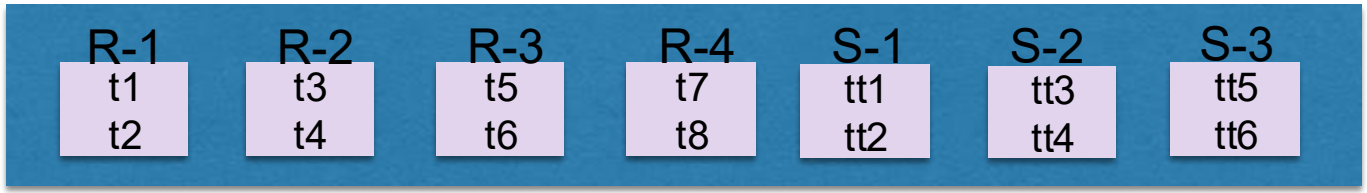
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



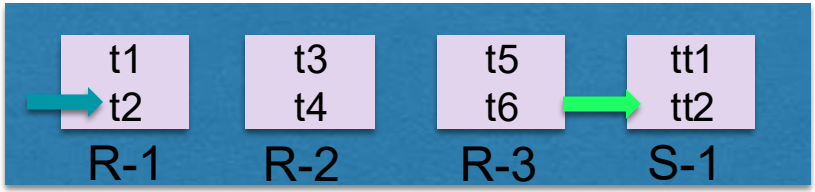
Disco



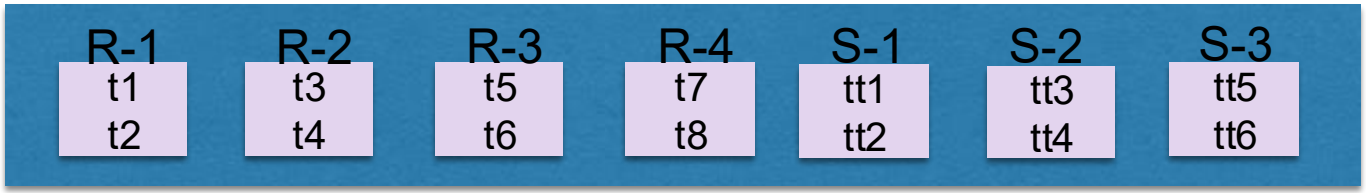
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



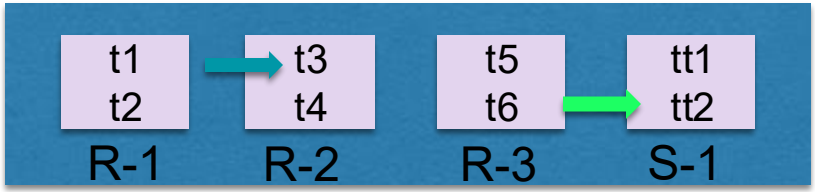
Disco



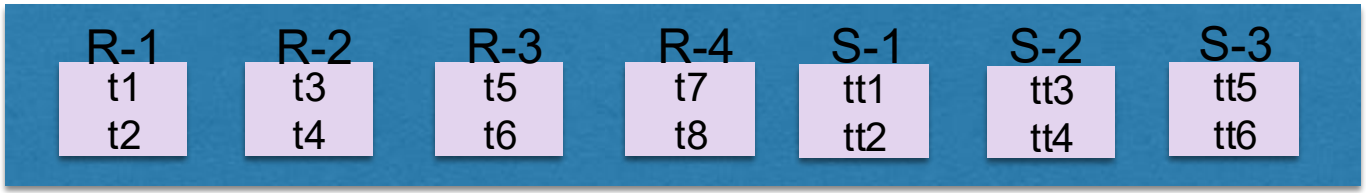
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



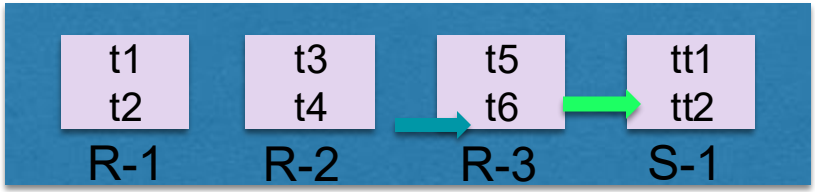
Disco



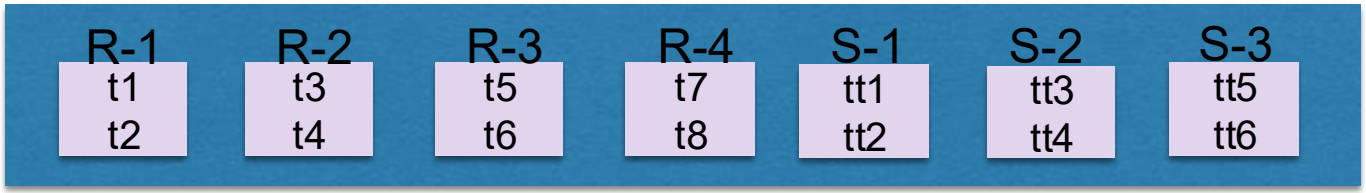
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



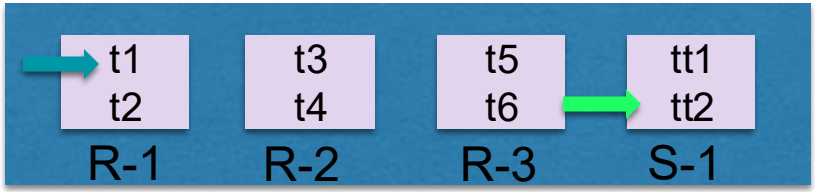
Disco



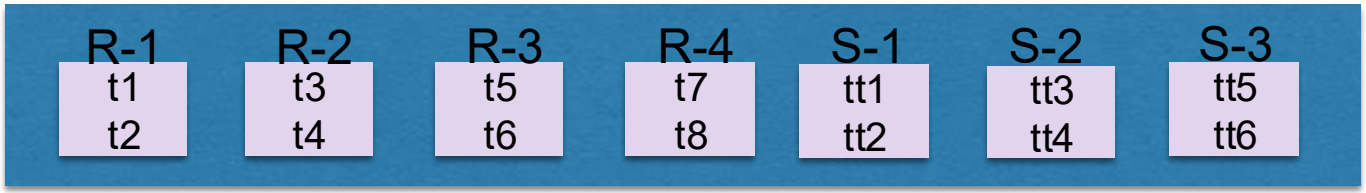
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



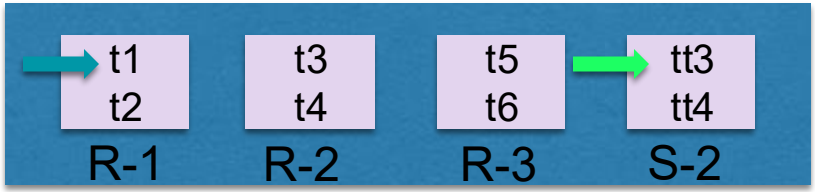
Disco



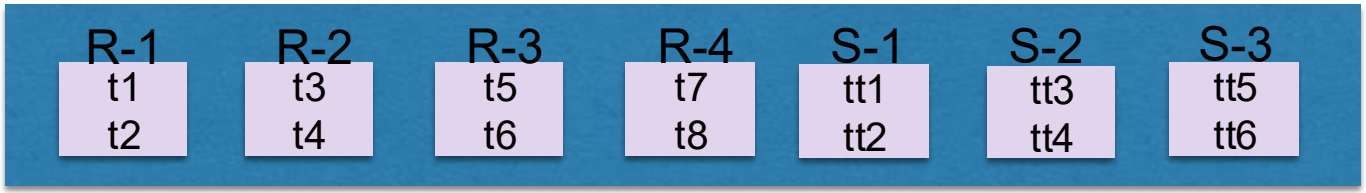
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



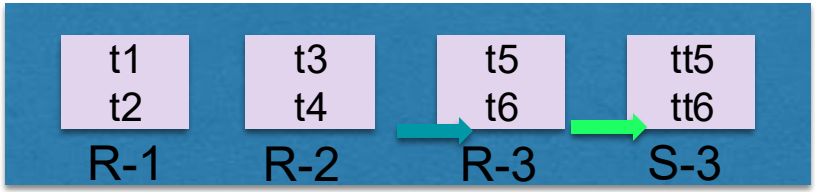
Disco



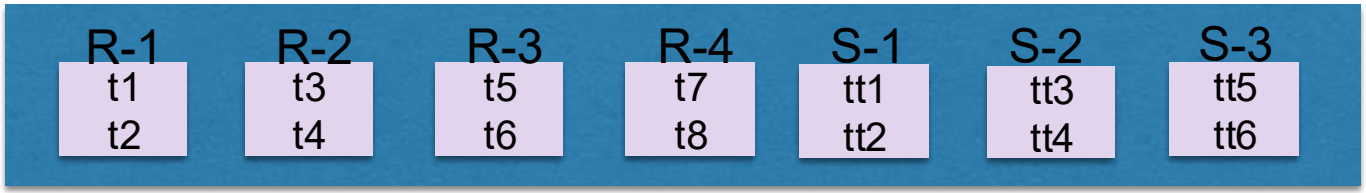
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



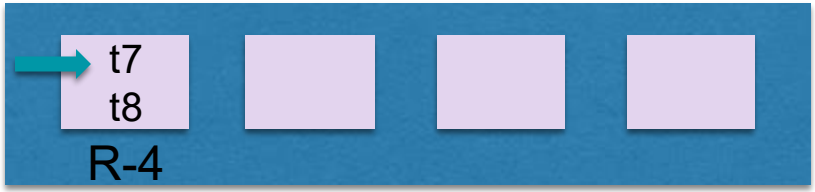
Disco



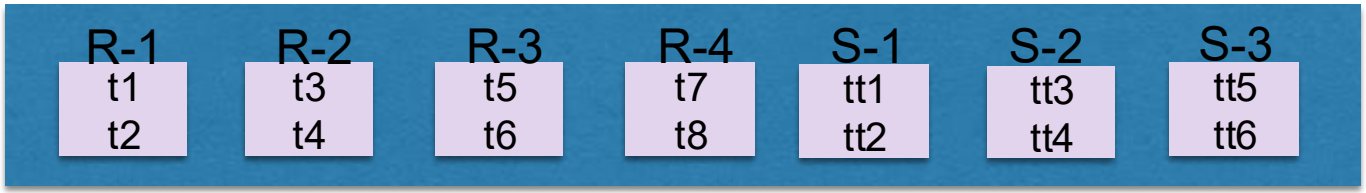
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



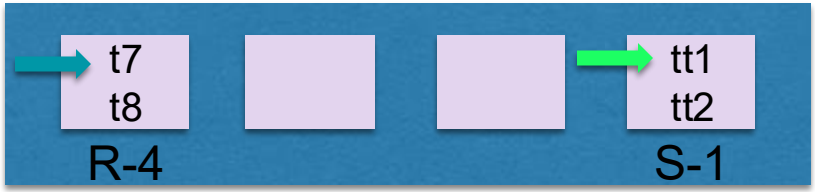
Disco



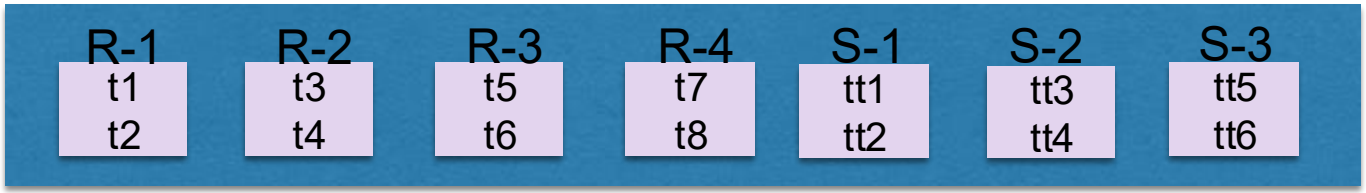
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



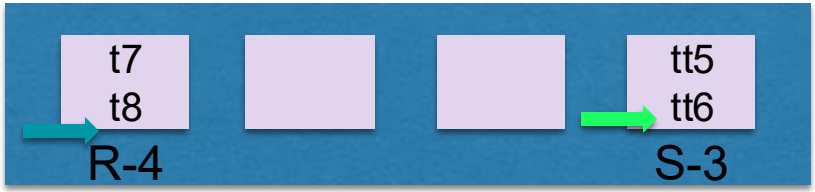
Disco



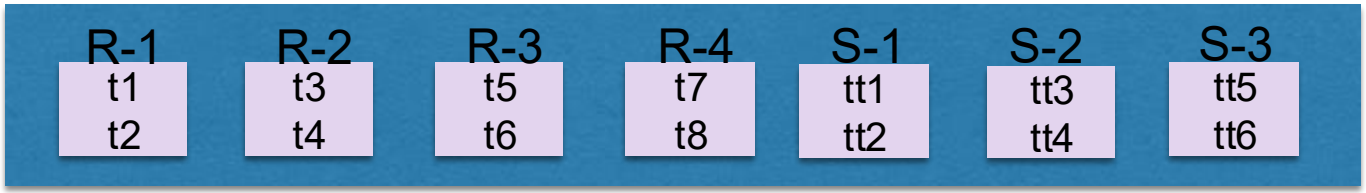
Block Nested Loop Join

DB while ($R \bowtie_p S$.next())

Buffer



Disco



Block Nested Loop Join

Ahora cargamos muchas páginas de **R** a buffer

Por cada vez que llenamos el buffer recorremos a **S** entera una vez

Costo en I/O es:

$$\text{Costo}(\mathbf{R}) + (\text{Páginas}(\mathbf{R})/(\text{Buffer}-2)) \cdot \text{Costo}(\mathbf{S})$$

Block Nested Loop Join

Si **R** y **S** son tablas de 16 MB, cada página es de 8 KB con un **buffer** de 1 MB

Cada relación tiene 2048 páginas y en **buffer** caben 128 páginas

Costo de un I/O es 0.1 ms, entonces el join tarda:

3.4 segundos

Block Nested Loop Join

Sin embargo existen algoritmos muchos más eficientes

Estos algoritmos se basan en **Hashing** o en **Sorting**

Además hacen usos de índices, como por ejemplo el B+ Tree

Sorting

Los algoritmos de sorting son conocidos en programación

¿Pero qué estudiarlos otra vez?

Para ordenar tuplas que exceden por mucho el tamaño de la memoria RAM

En los DBMS, se utiliza el algoritmo **External Merge Sort**

Pero antes, veamos **Merge y MergeSort**

Merge y MergeSort

Mezcla (*merge*) de secuencias ordenadas

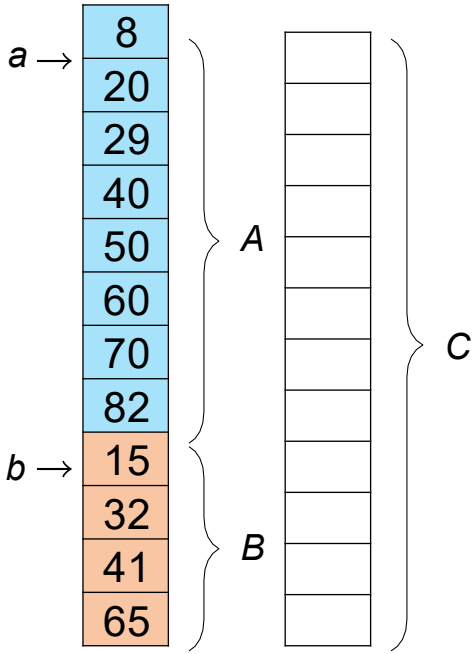
Proponemos el siguiente algoritmo para combinar dos secuencias ordenadas para formar una nueva **ordenada**

input : Secuencias ordenadas *A* y *B*
output: Nueva secuencia ordenada *C*

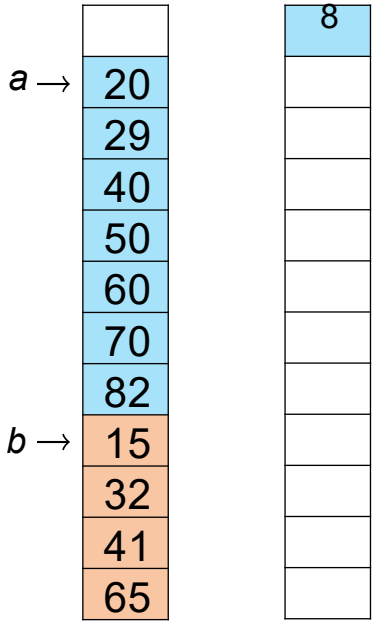
Merge(A, B):

1. Iniciamos *C* vacía
2. Sean *a* y *b* los primeros elementos de *A* y *B*
3. Extraer de su secuencia respectiva el menor entre *a* y *b*
4. Insertar el elemento extraído al final de *C*
5. Si quedan elementos en *A* **y** *B*, volver a la línea 2
6. Concatenar *C* con la secuencia que aún tenga elementos
7. **return** *C*

Merge: Ejemplo de ejecución

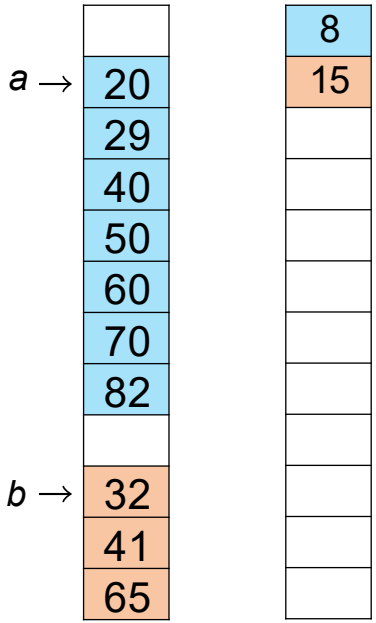


Estado inicial

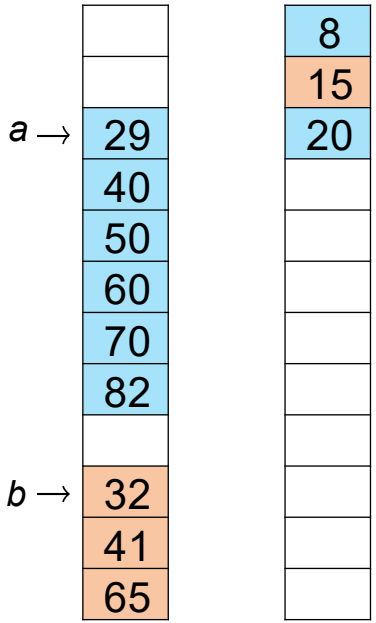


Estado luego de la primera iteración

Merge: Ejemplo de ejecución

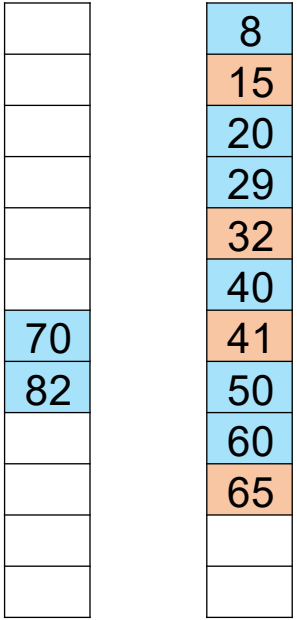


Estado luego de la segunda iteración

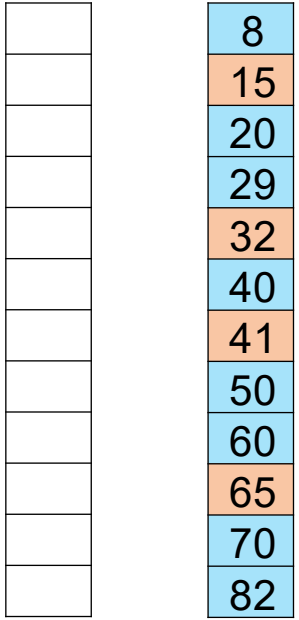


Estado luego de la tercera iteración

Merge: Ejemplo de ejecución



Estado luego de insertar en *C*
el último elemento de *B*



Estado luego de
concatenar el resto de *A*

Dividir para conquistar

El plan para usar Merge en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos:

1. Dividir el problema original en dos (o m´as) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

Los sub-problemas son instancias más pequeñas del problema a resolver

Dividir para conquistar y Merge

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando Merge

¿En qué parte del dividir para conquistar usaremos Merge?

La idea general para ordenar usando Mergedefine un nuevo algoritmo que llamaremos **MergeSort**

1. Dividir la secuencia original en dos sub-secuencias
2. Llamamos recursivamente a MergeSort sobre las dos sub-secuencias
3. Combinamos las secuencias ordenadas resultantes mediante Merge

El algoritmo MergeSort

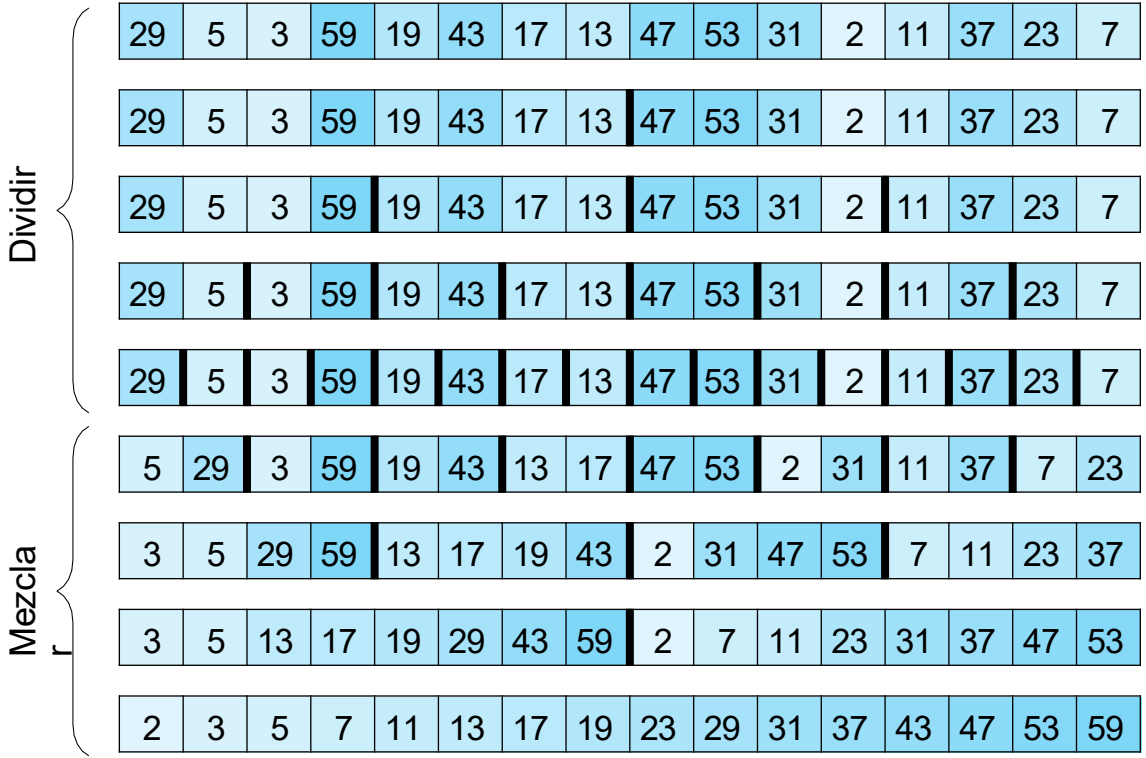
A continuación, tenemos el pseudocodigo del algoritmo recursivo MergeSort

input : Secuencia A
output: Secuencia ordenada B

MergeSort *(A)*:

1 **if** $|A| = 1$ **: return** A
2 Dividir A en mitades A_1 y A_2
3 $B_1 \leftarrow \text{MergeSort}(A_1)$
4 $B_2 \leftarrow \text{MergeSort}(A_2)$
5 $B \leftarrow \text{Merge}(B_1, B_2)$
6 **return** B

MergeSort: Ejemplo de ejecución



External Merge Sort

External Merge Sort

En los DBMS, se utiliza el algoritmo External Merge Sort

Hablaremos de **Run** como una secuencia de páginas que contiene una conjunto ordenado de tuplas

Algoritmo funciona por **fases**

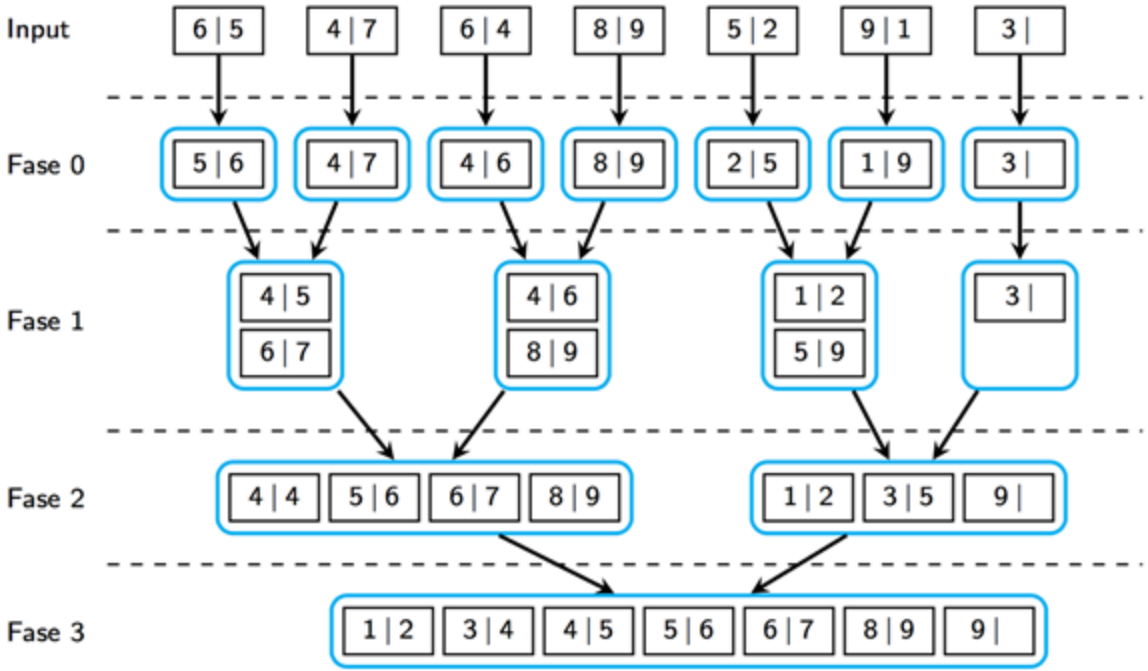
External Merge Sort

Fase 0: creamos los runs iniciales

Fase i:

- Traemos los runs a memoria
- Hacemos el merge de cada par de runs
- Almacenamos el nuevo run a disco (i.e. materializamos resultados intermedios)

External Merge Sort



Recursos para estudiar

Database Management Systems, 3rd edition, de Raghu Ramakrishnan y Johannes Gehrke. Part IV