Bases de Datos

Clase 8: Algoritmos de los DBMS

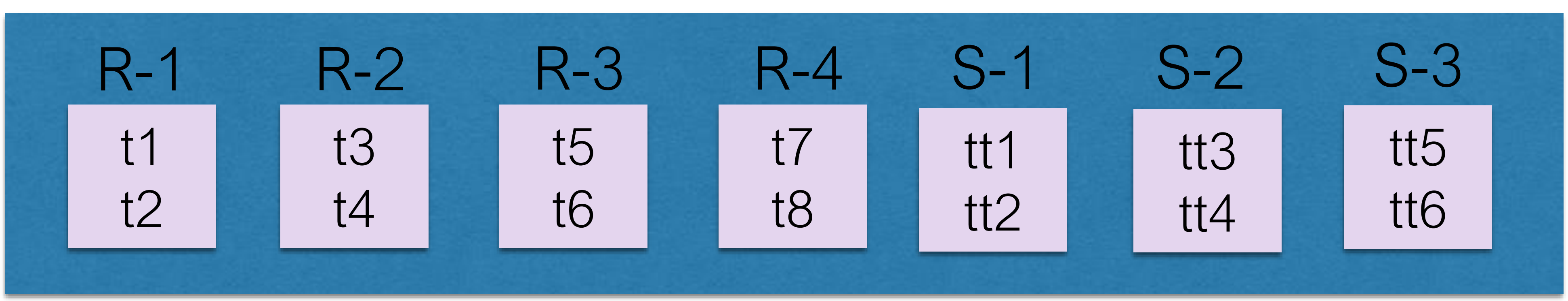
Páginas, disco y buffer

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**

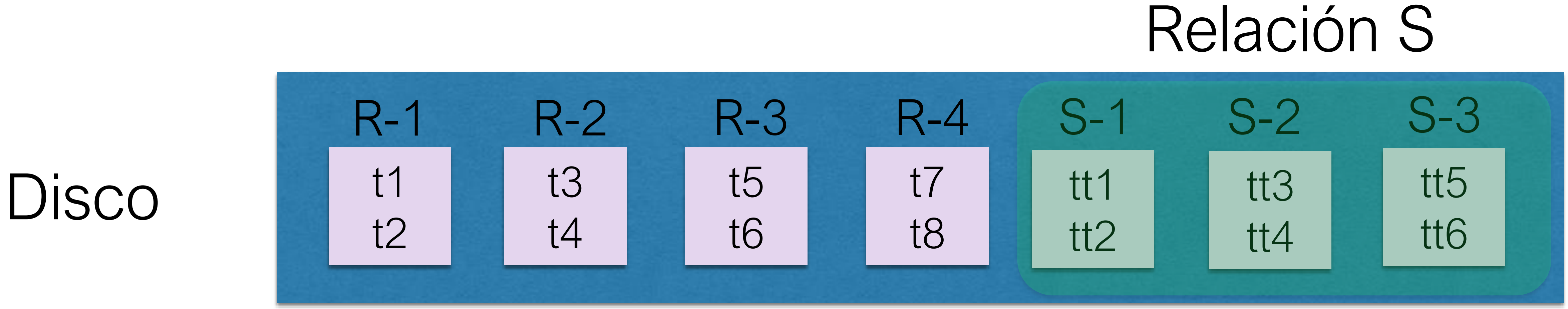
Páginas, disco y buffer

Disco

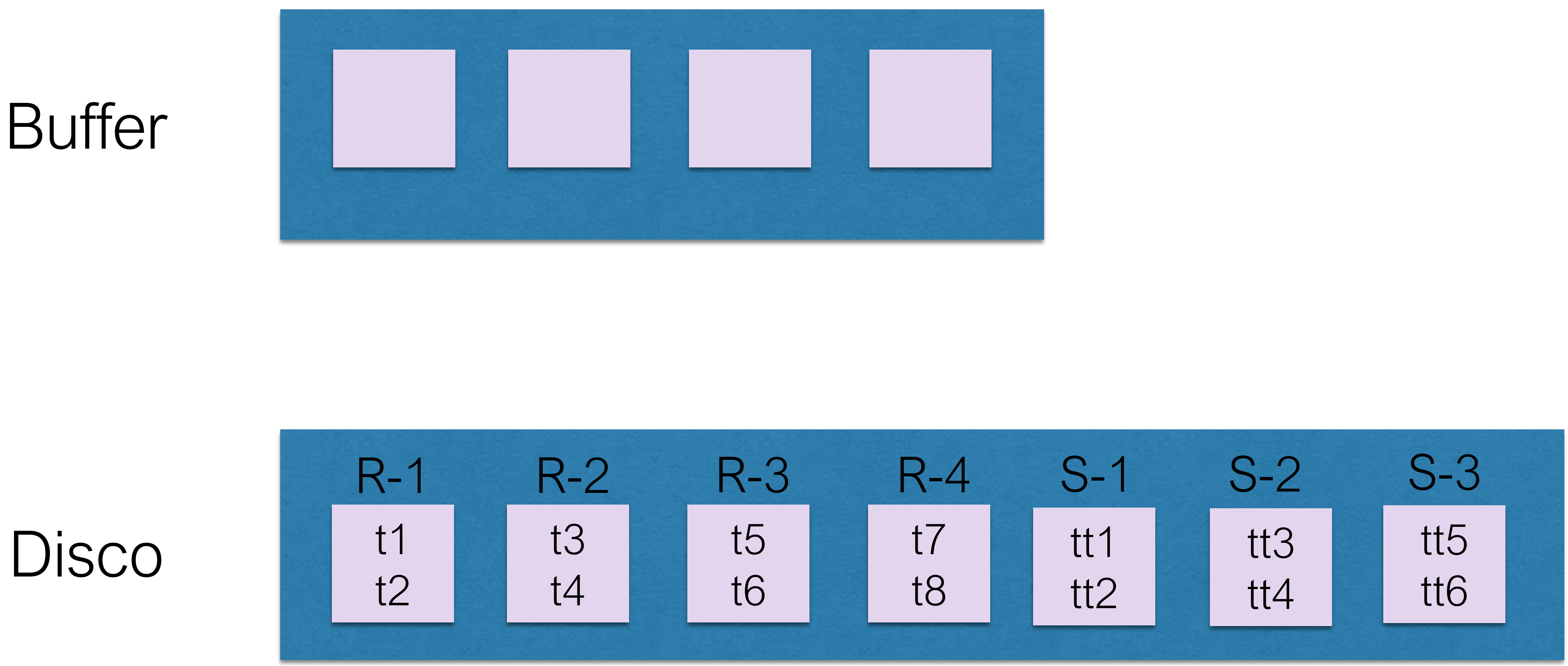


Páginas, disco y buffer

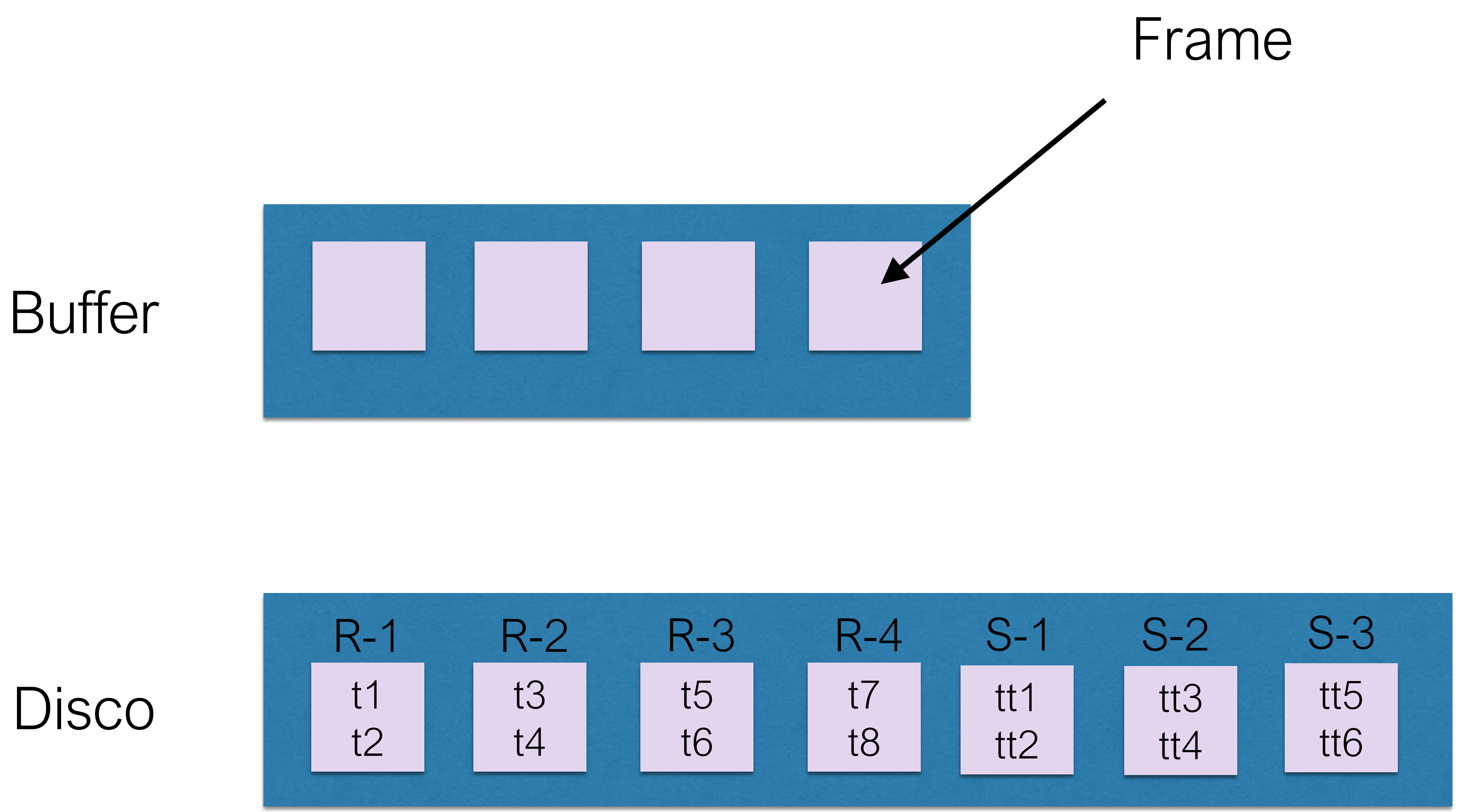
Páginas, disco y buffer



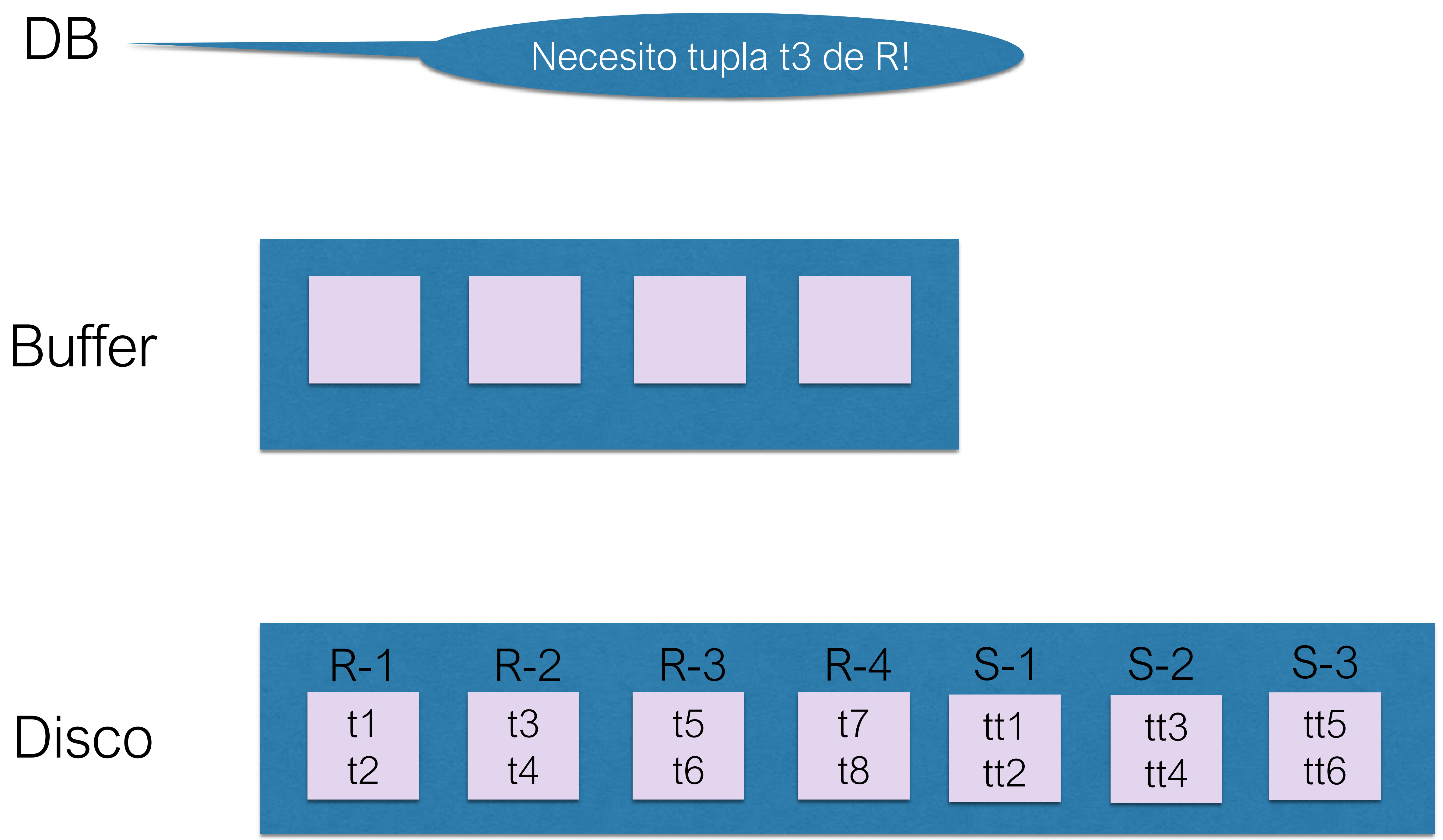
Páginas, disco y buffer



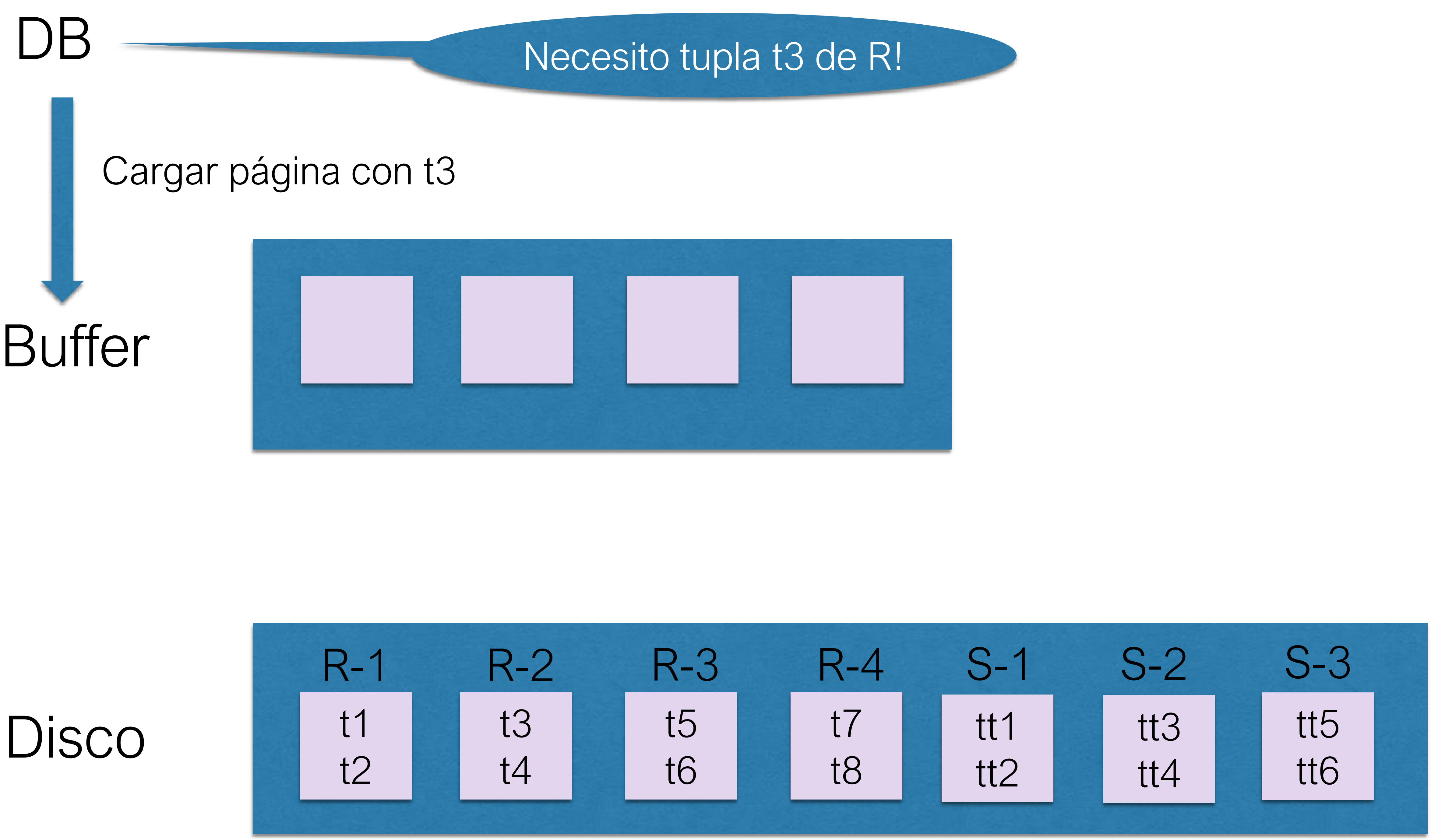
Páginas, disco y buffer



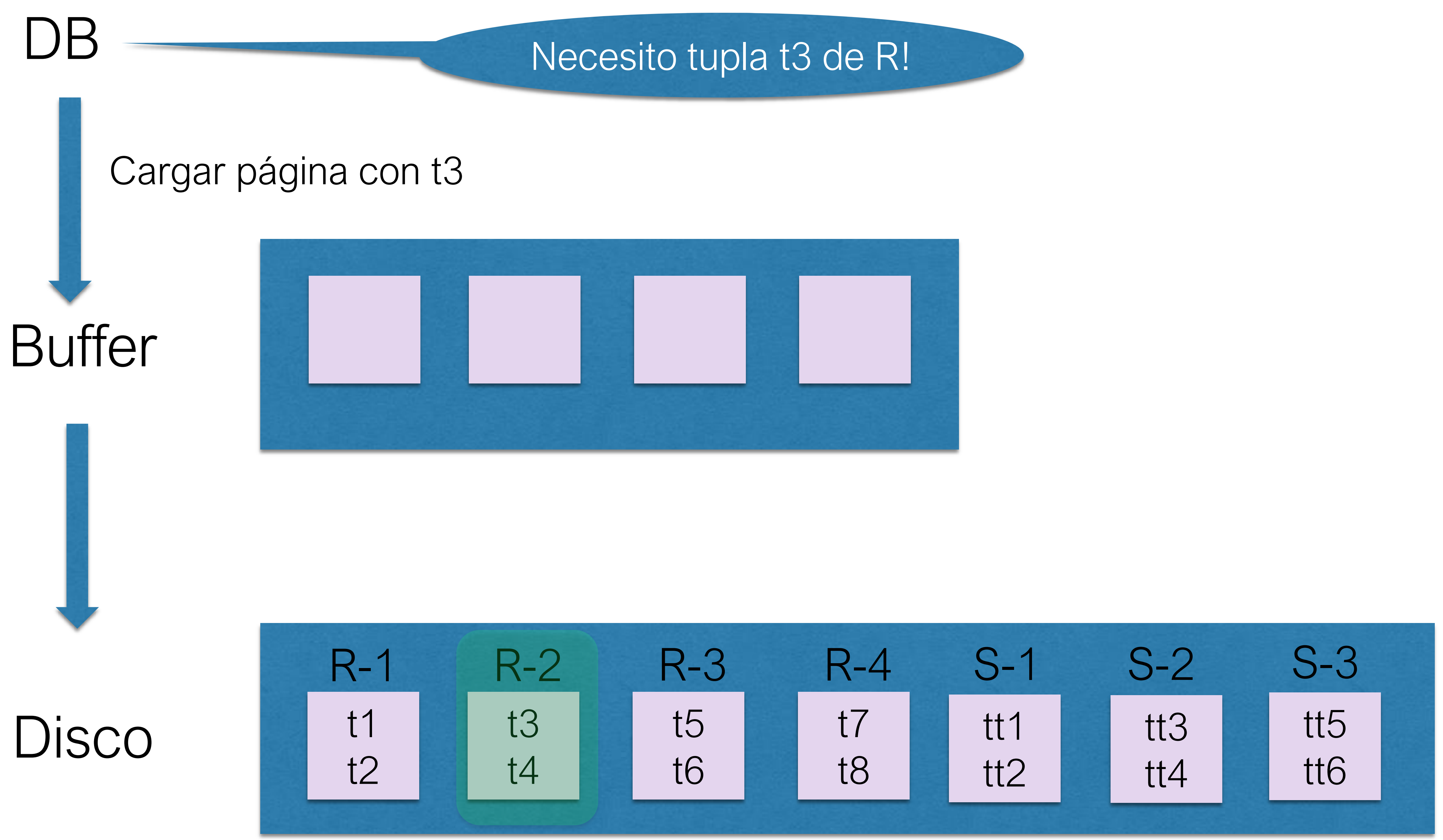
Páginas, disco y buffer



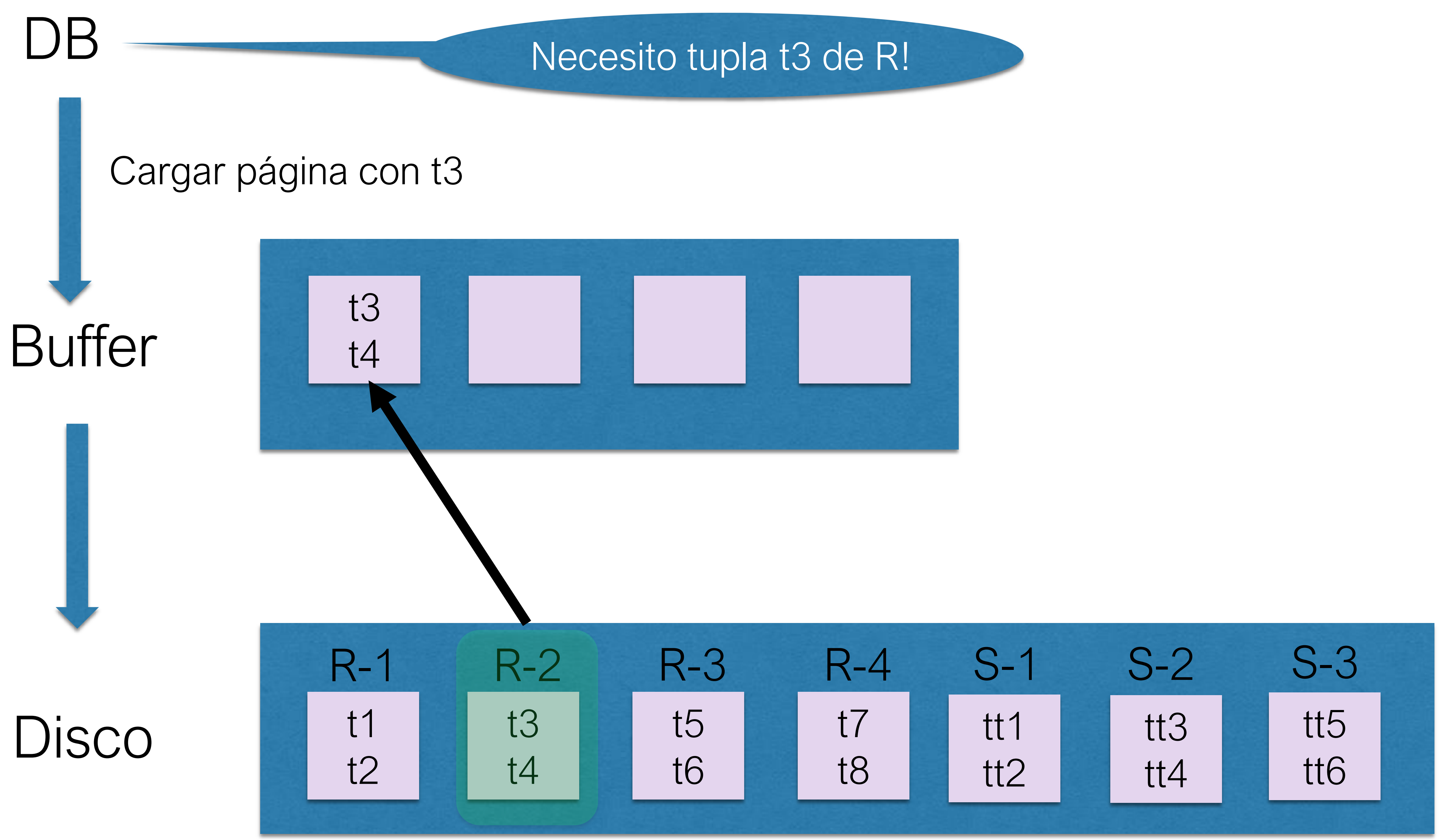
Páginas, disco y buffer



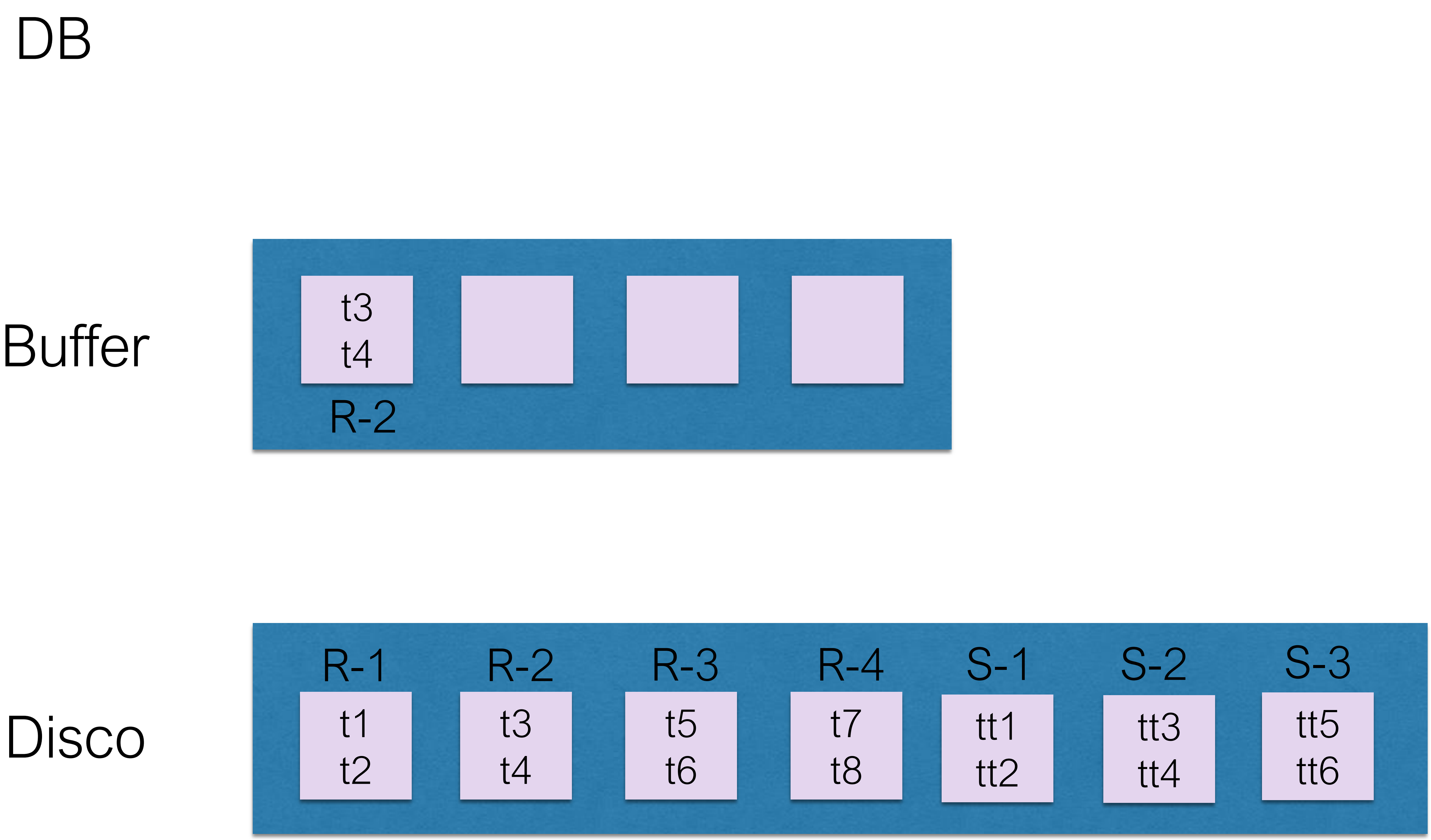
Páginas, disco y buffer



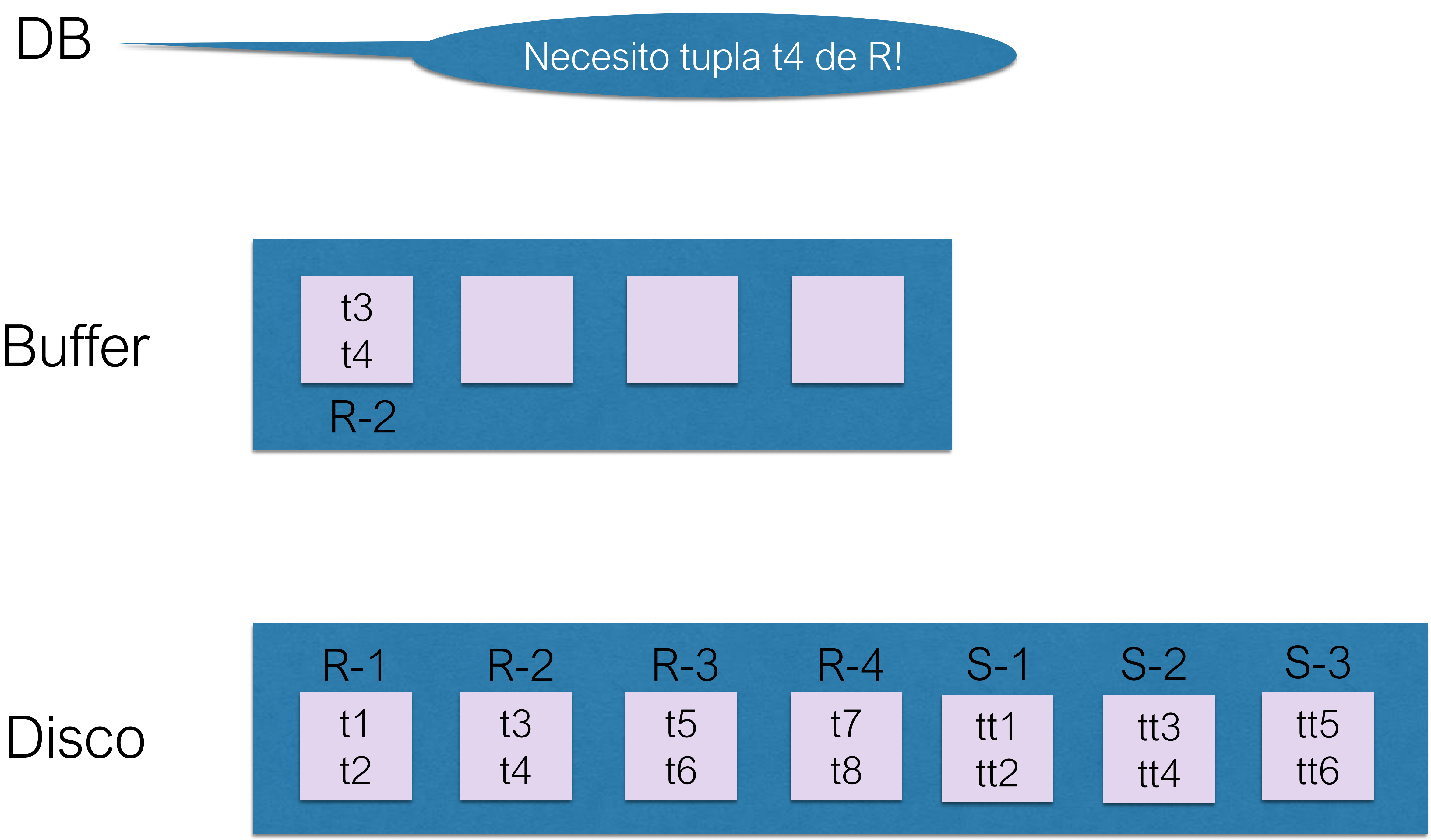
Páginas, disco y buffer



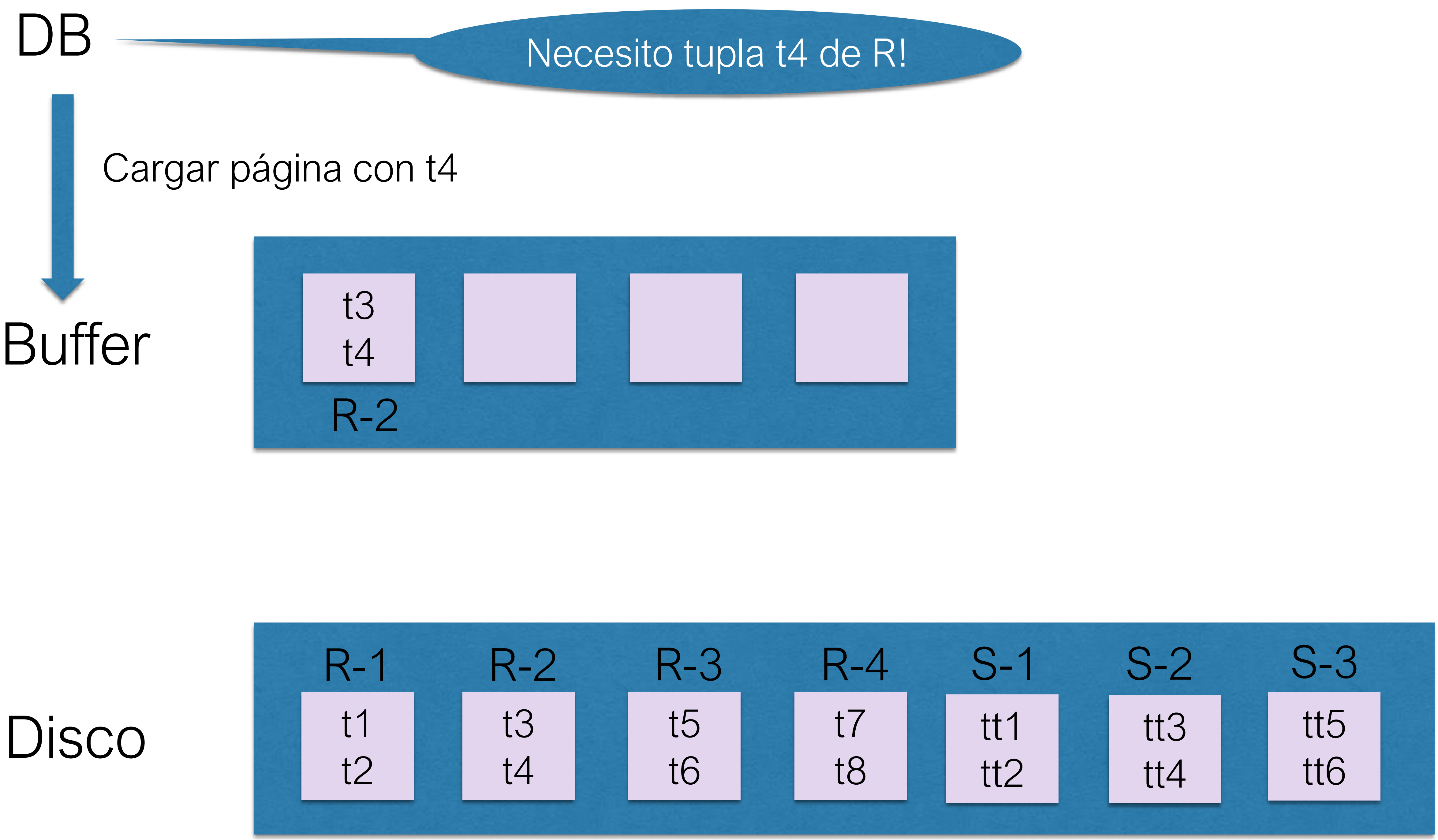
Páginas, disco y buffer



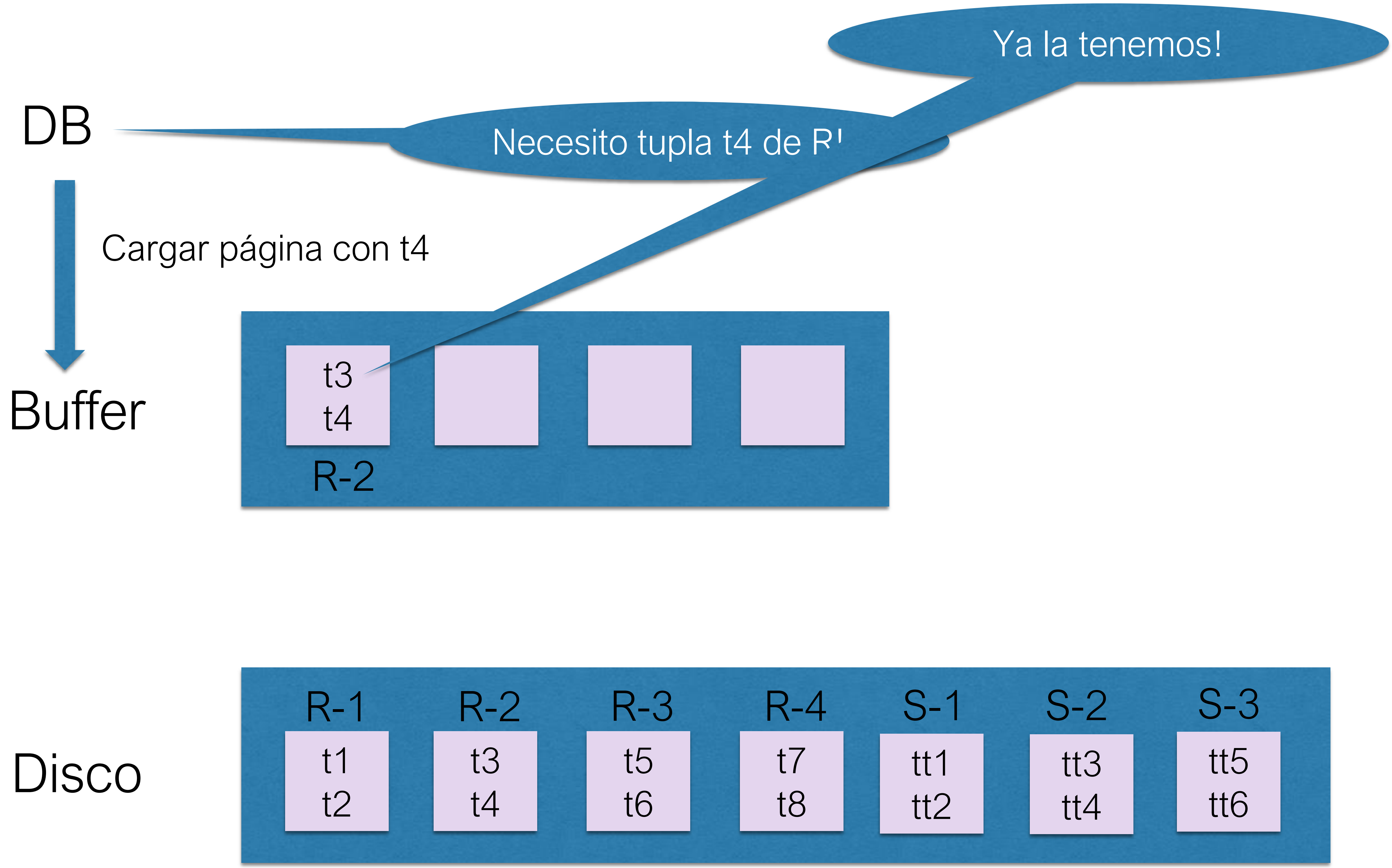
Páginas, disco y buffer



Páginas, disco y buffer



Páginas, disco y buffer



Costo de un algoritmo

Cuantas 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

Implementan interfaz de un iterador lineal:

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

Algoritmos en una BD

Para una relación R:

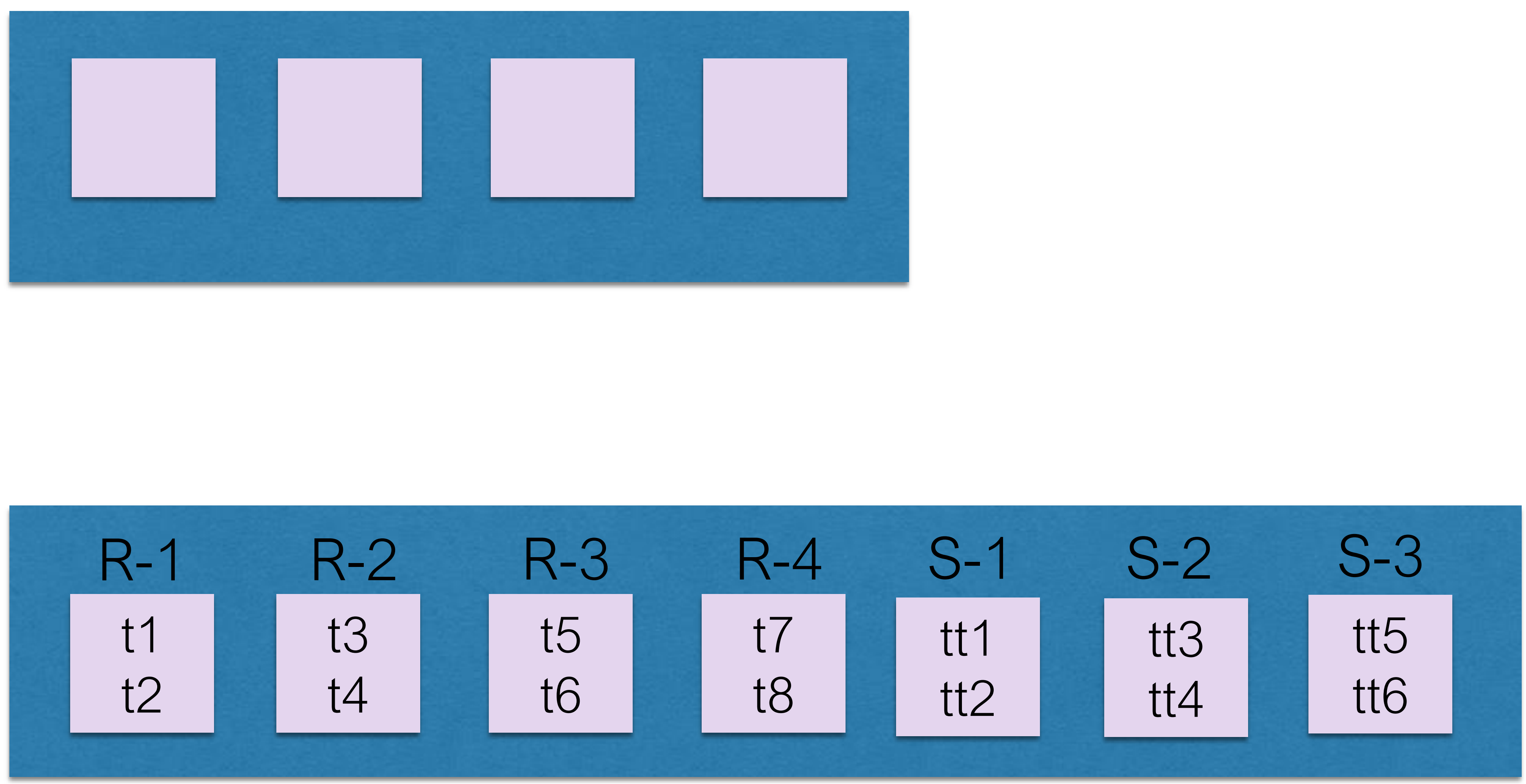
- R.open() – se posiciona **antes** de la primera tupla de R
- R.next() – devuelve la siguiente tupla o NULL
- R.close() – cierra el iterador

Páginas, disco y buffer

DB

Buffer

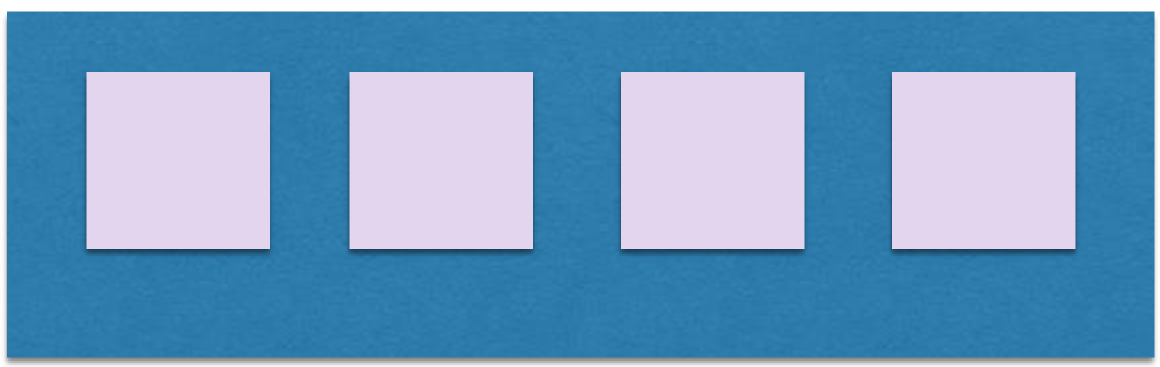
Disco



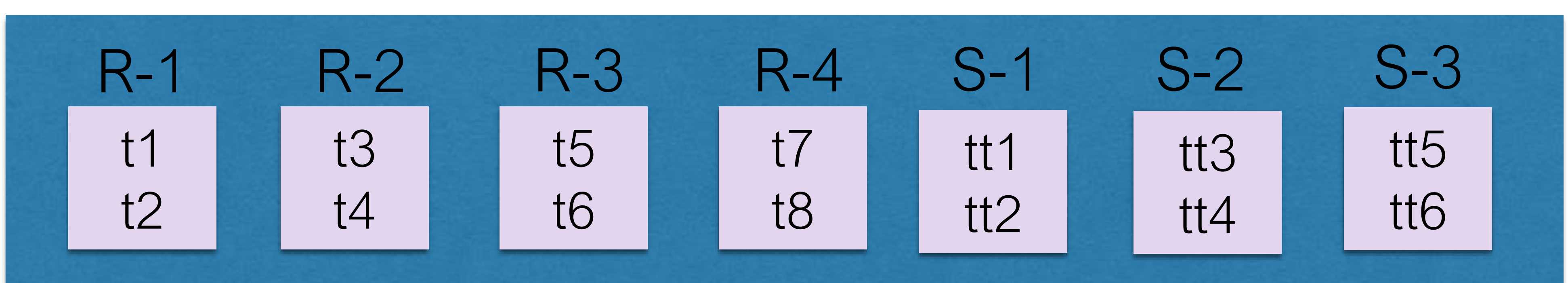
Páginas, disco y buffer

DB R.open()

Buffer



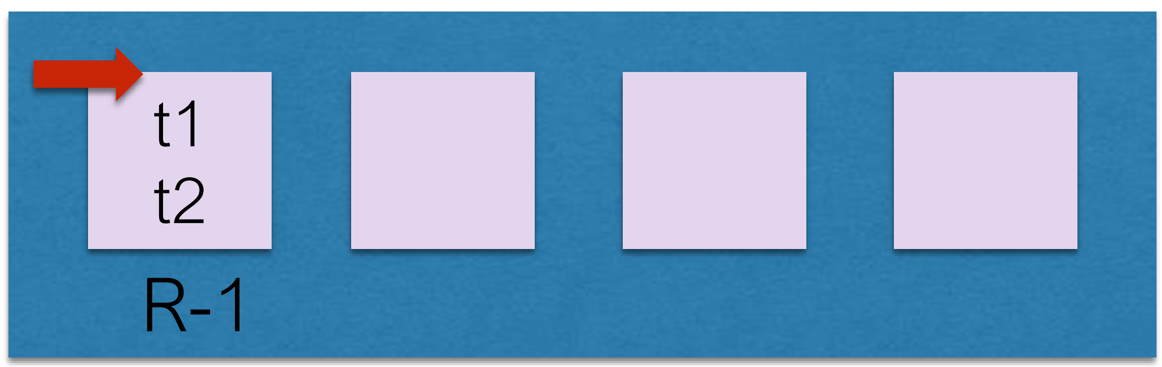
Disco



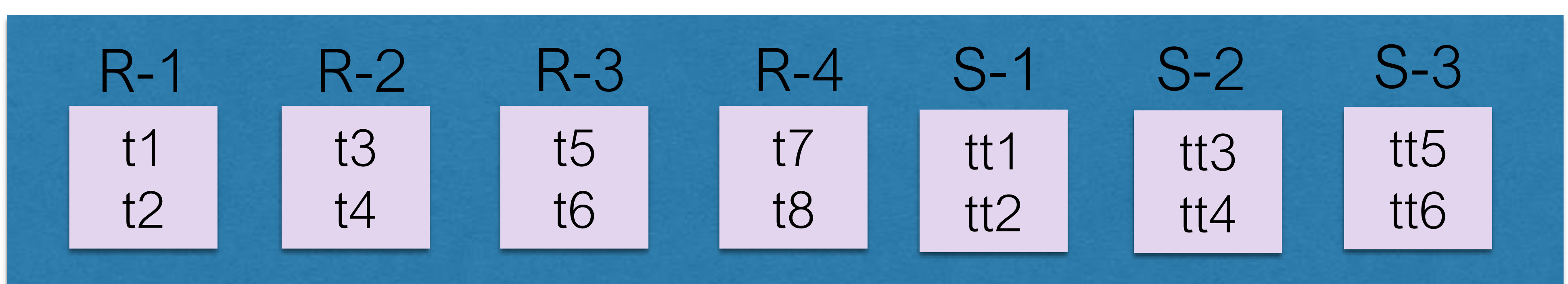
Páginas, disco y buffer

DB R.open()

Buffer

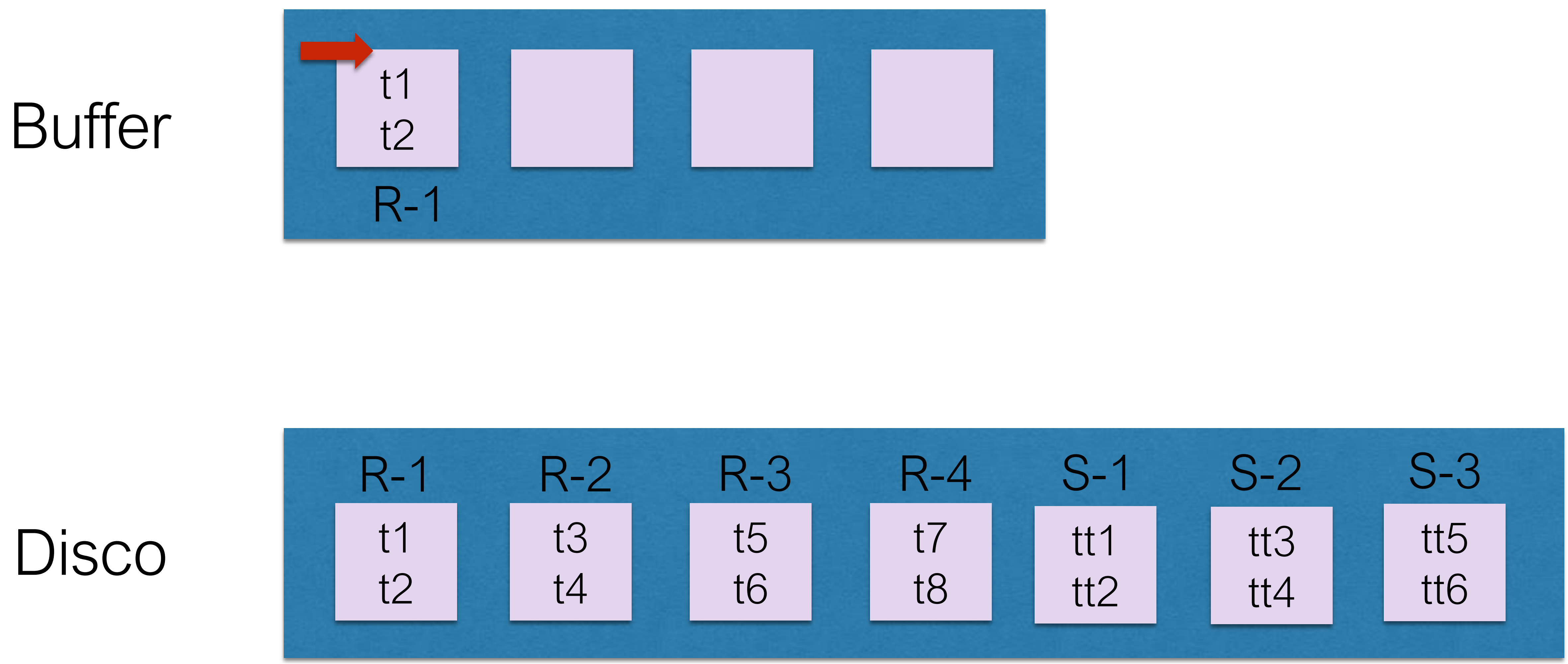


Disco



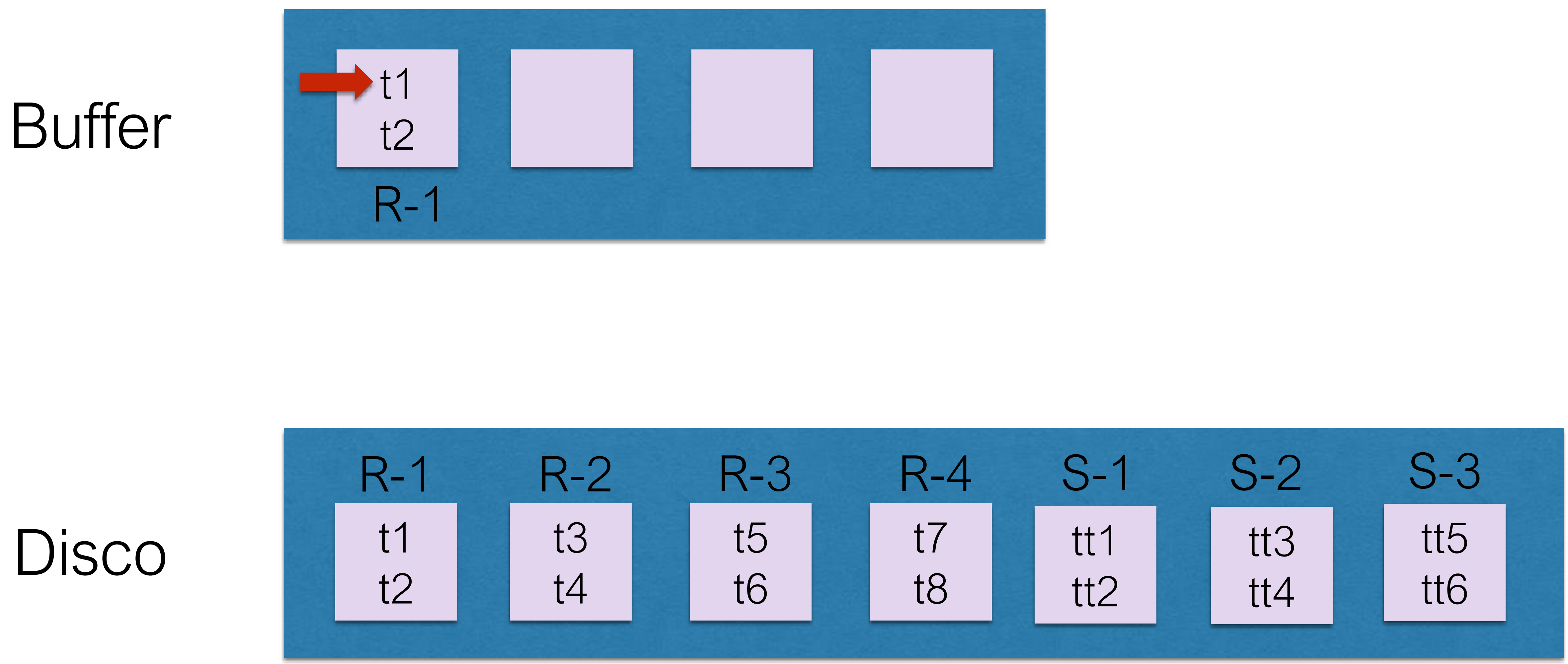
Páginas, disco y buffer

DB R.next()



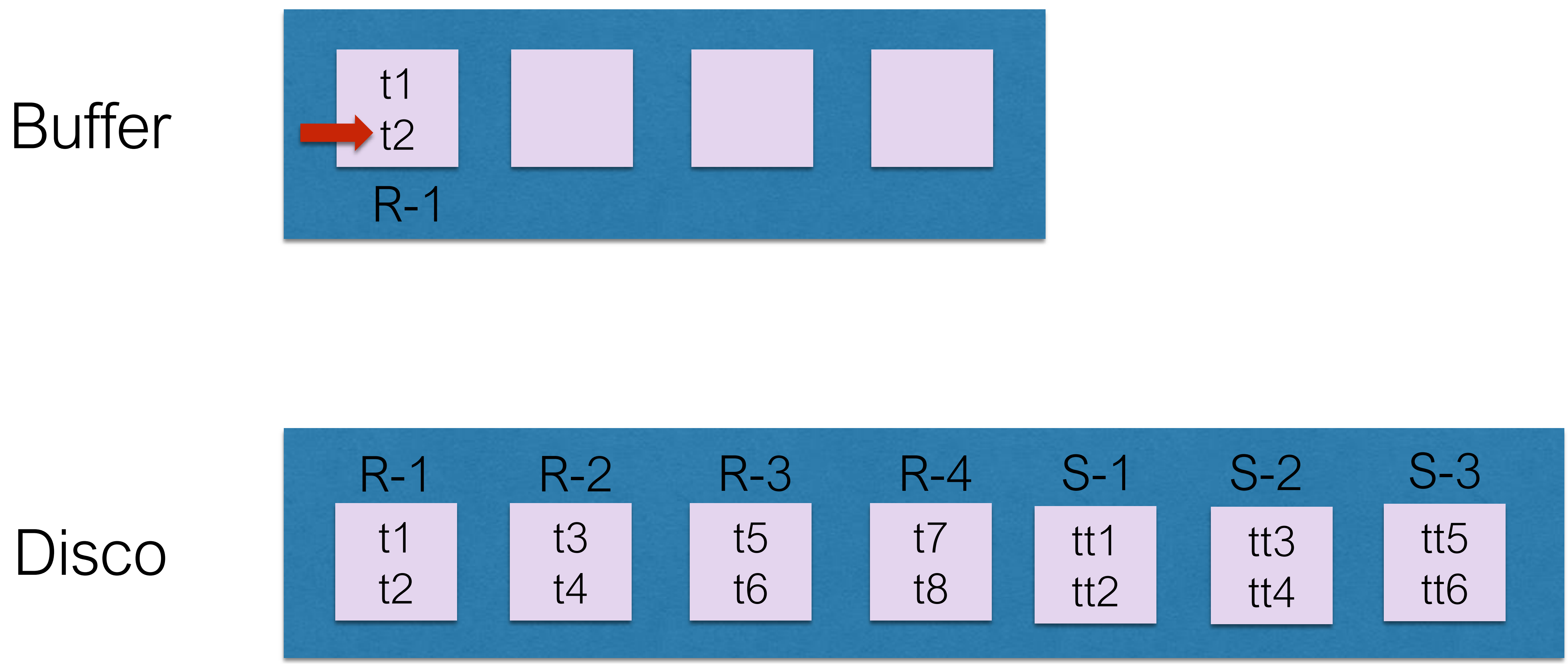
Páginas, disco y buffer

DB R.next()



Páginas, disco y buffer

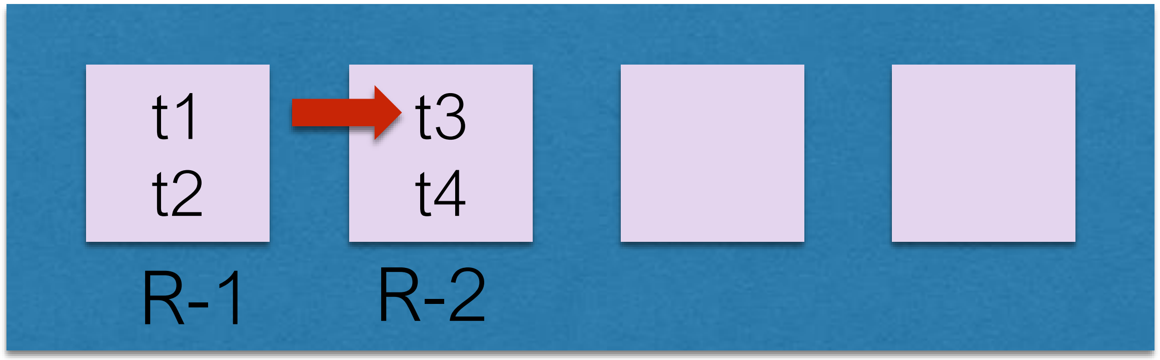
DB R.next()



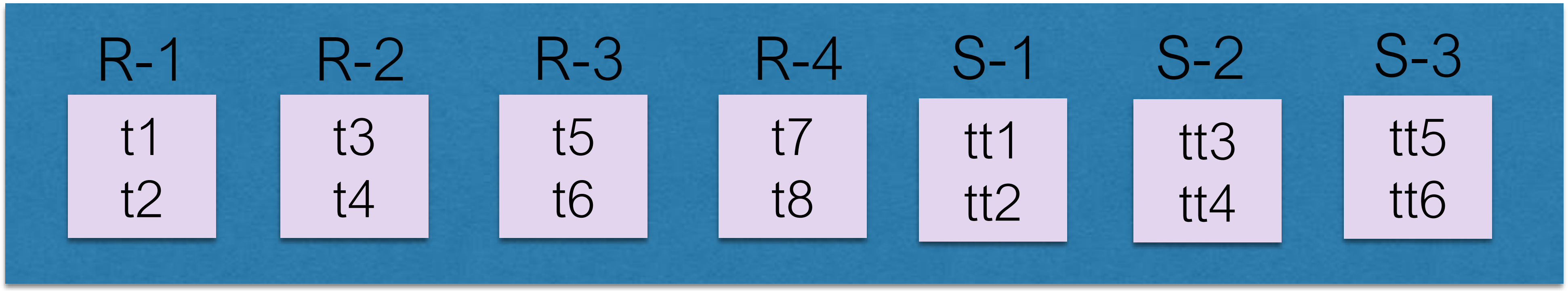
Páginas, disco y buffer

DB R.next()

Buffer



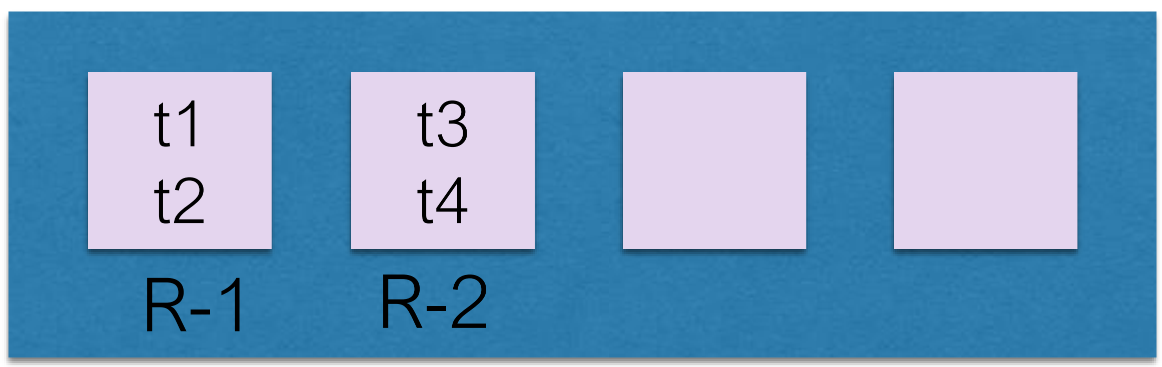
Disco



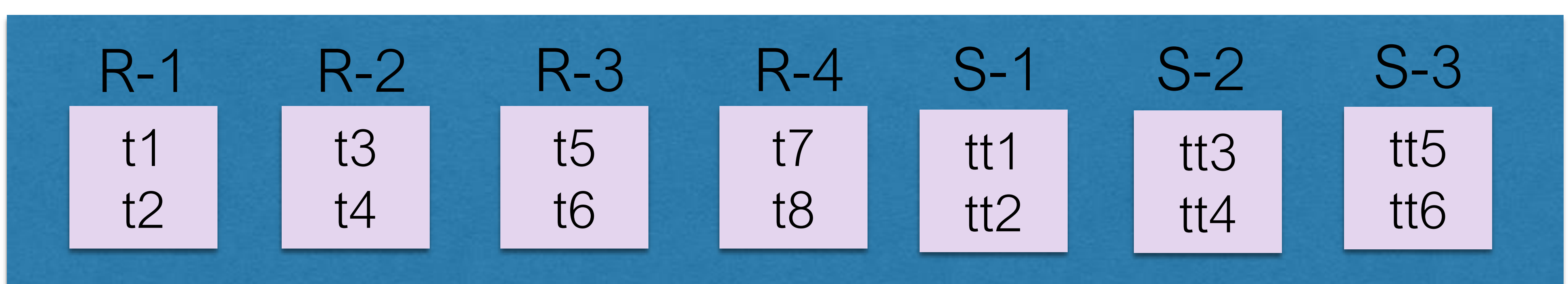
Páginas, disco y buffer

DB R.close()

Buffer



Disco



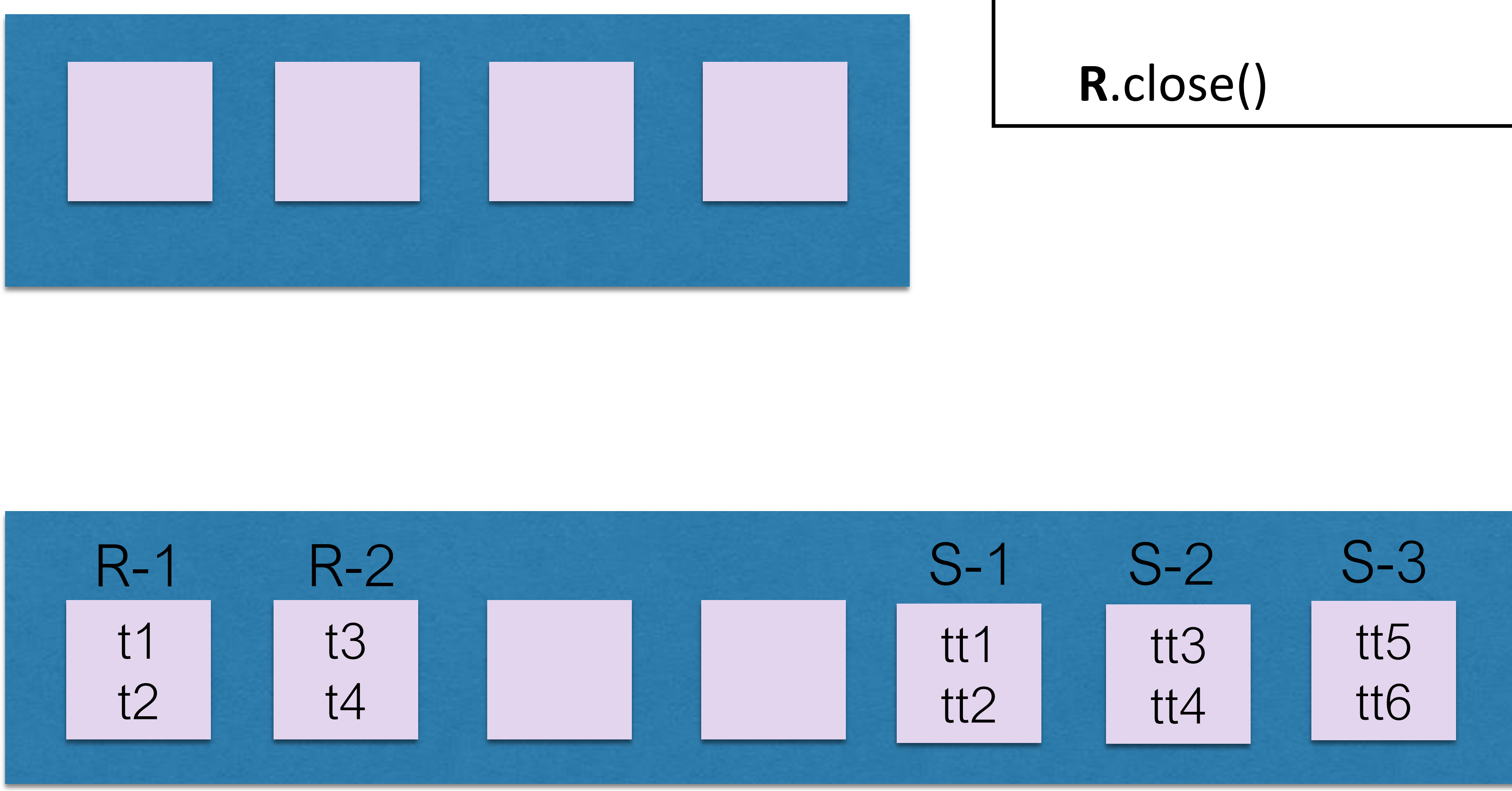
SELECT * FROM R

```
R.open()

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

R.close()
```

DB

Buffer

Disco



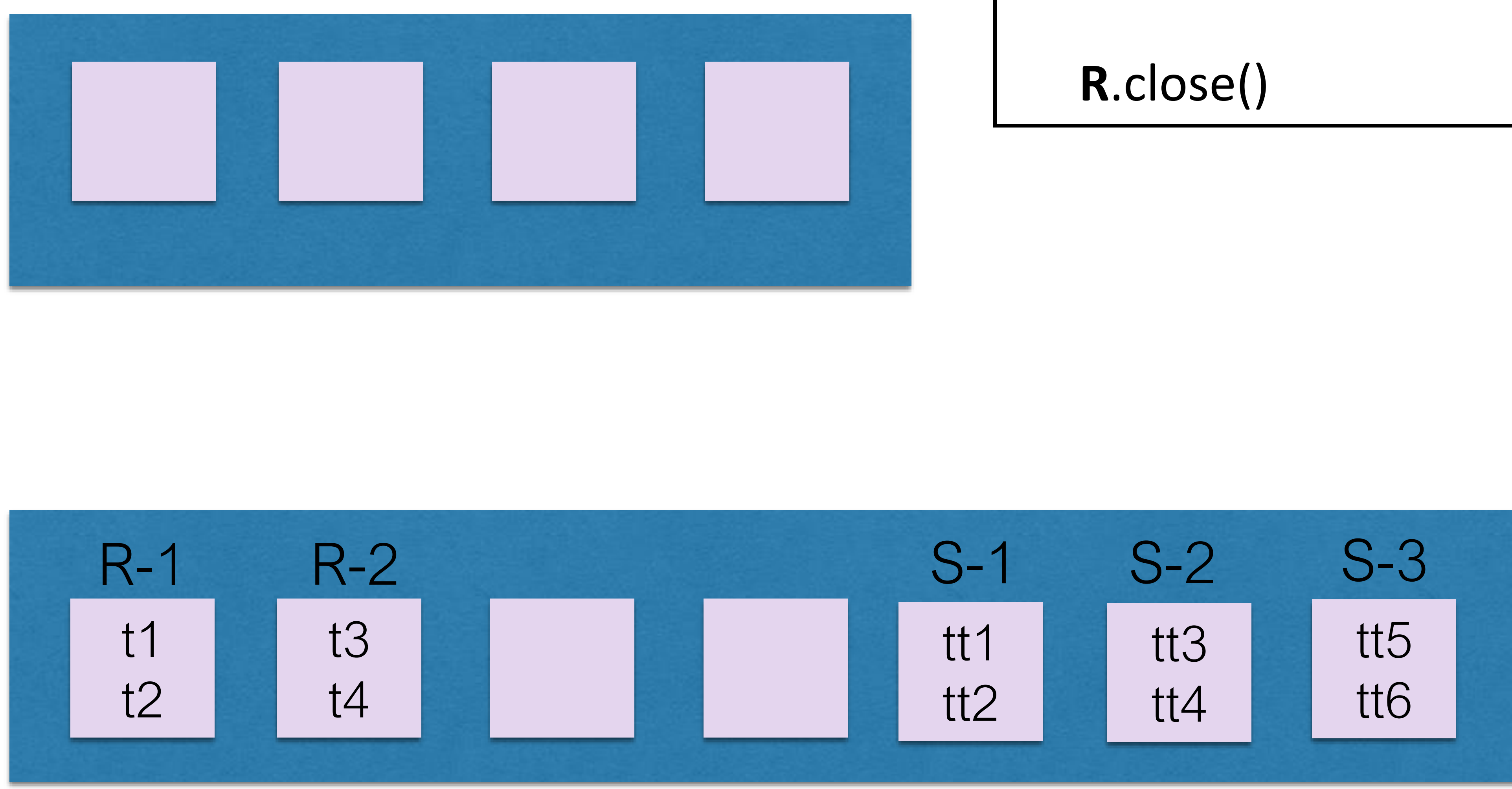
SELECT * FROM R

```
R.open()

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

R.close()
```

DB

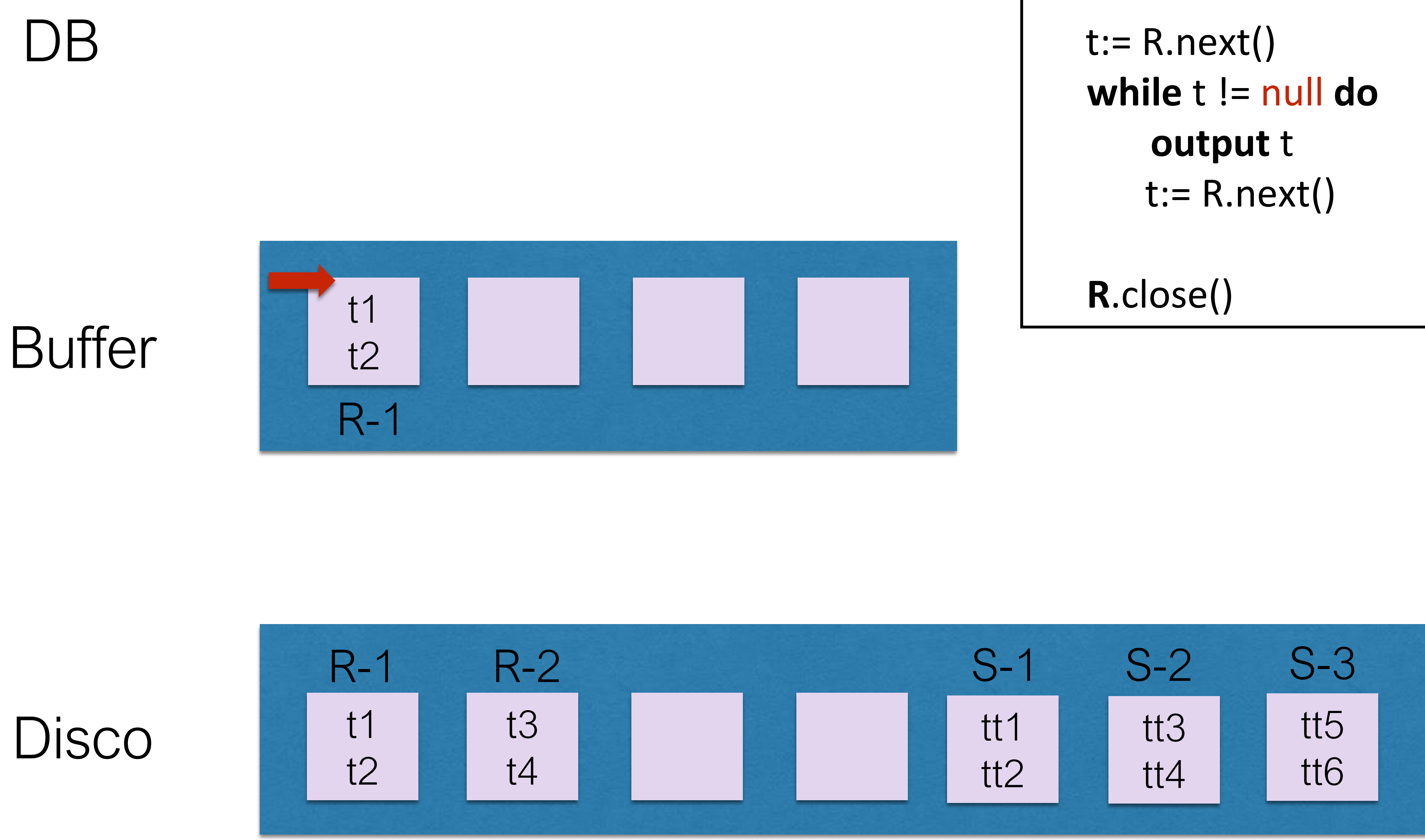
Buffer

Disco



SELECT * FROM R

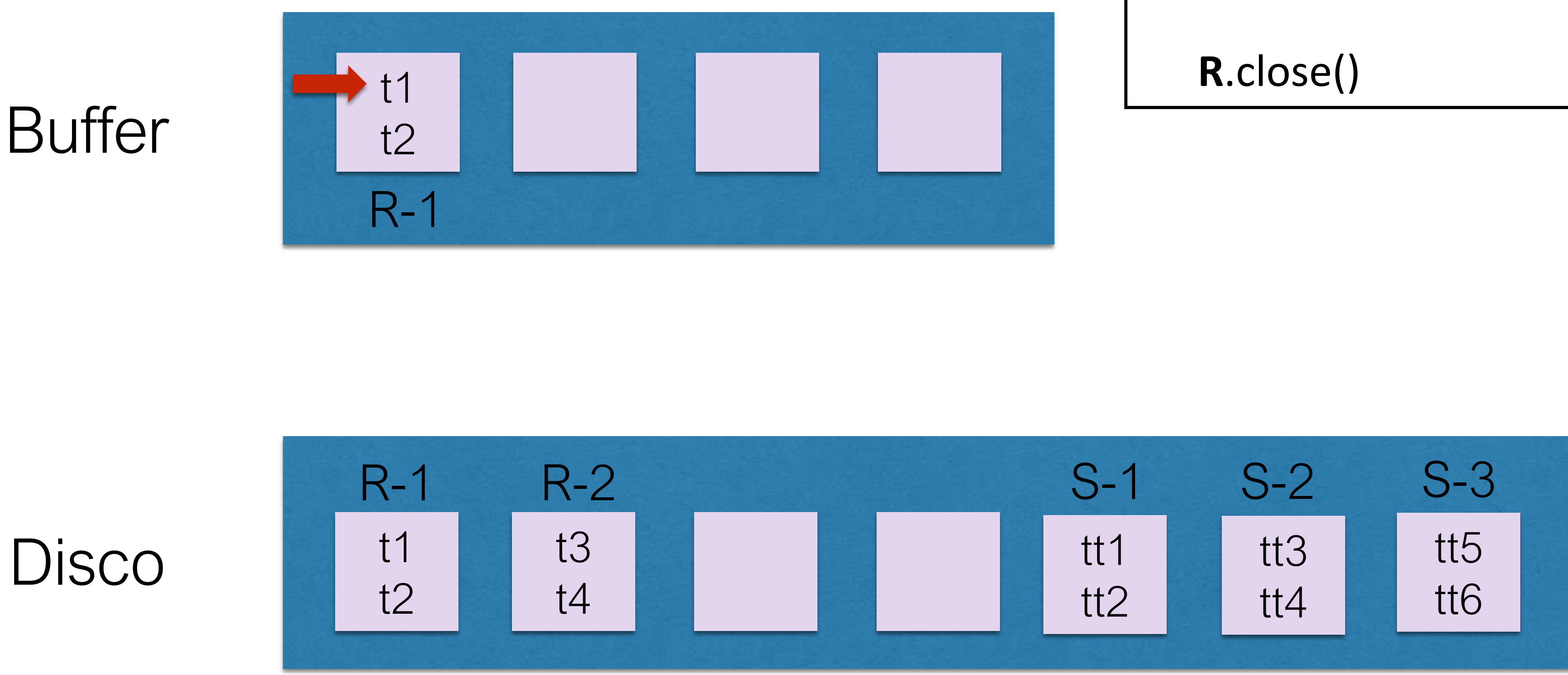
```
R.open()

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

R.close()
```



SELECT * FROM R

```
R.open()

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

R.close()
```

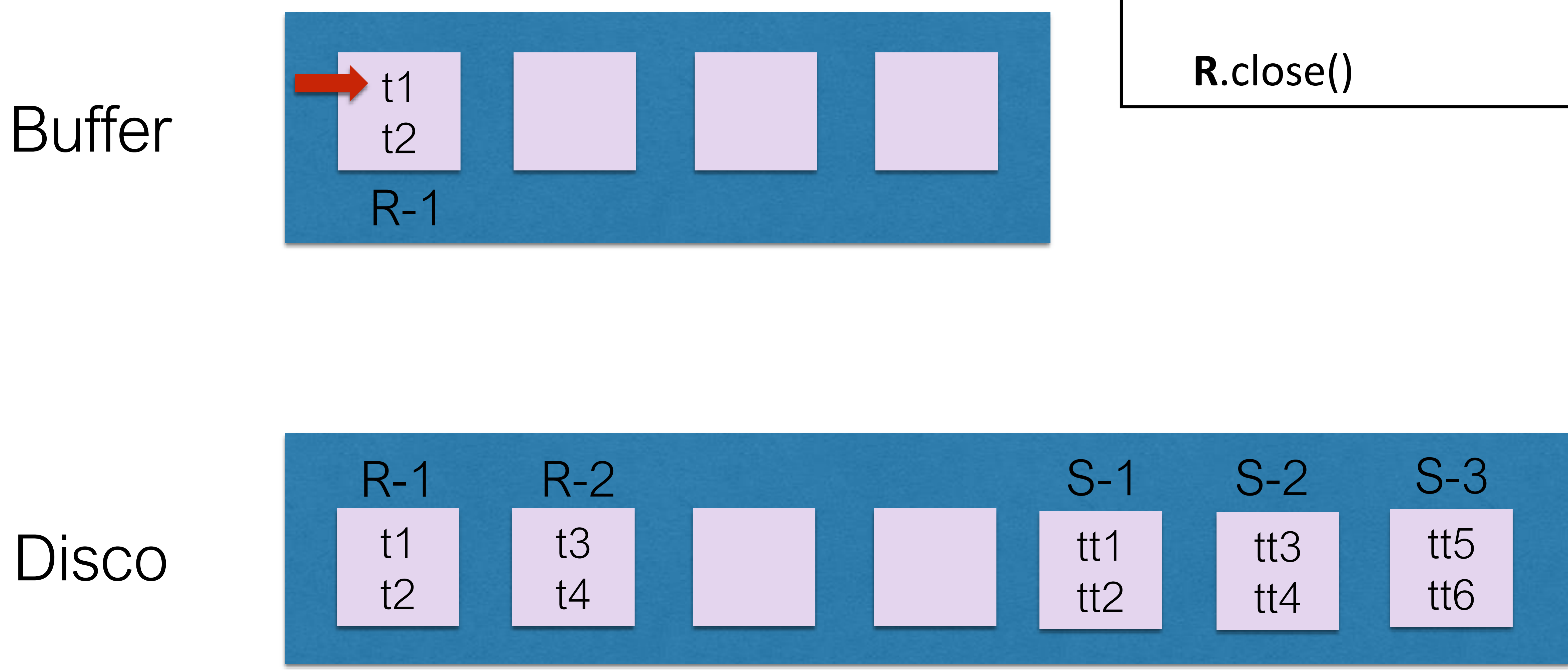
DB



SELECT * FROM R

```
R.open()

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

R.close()
```

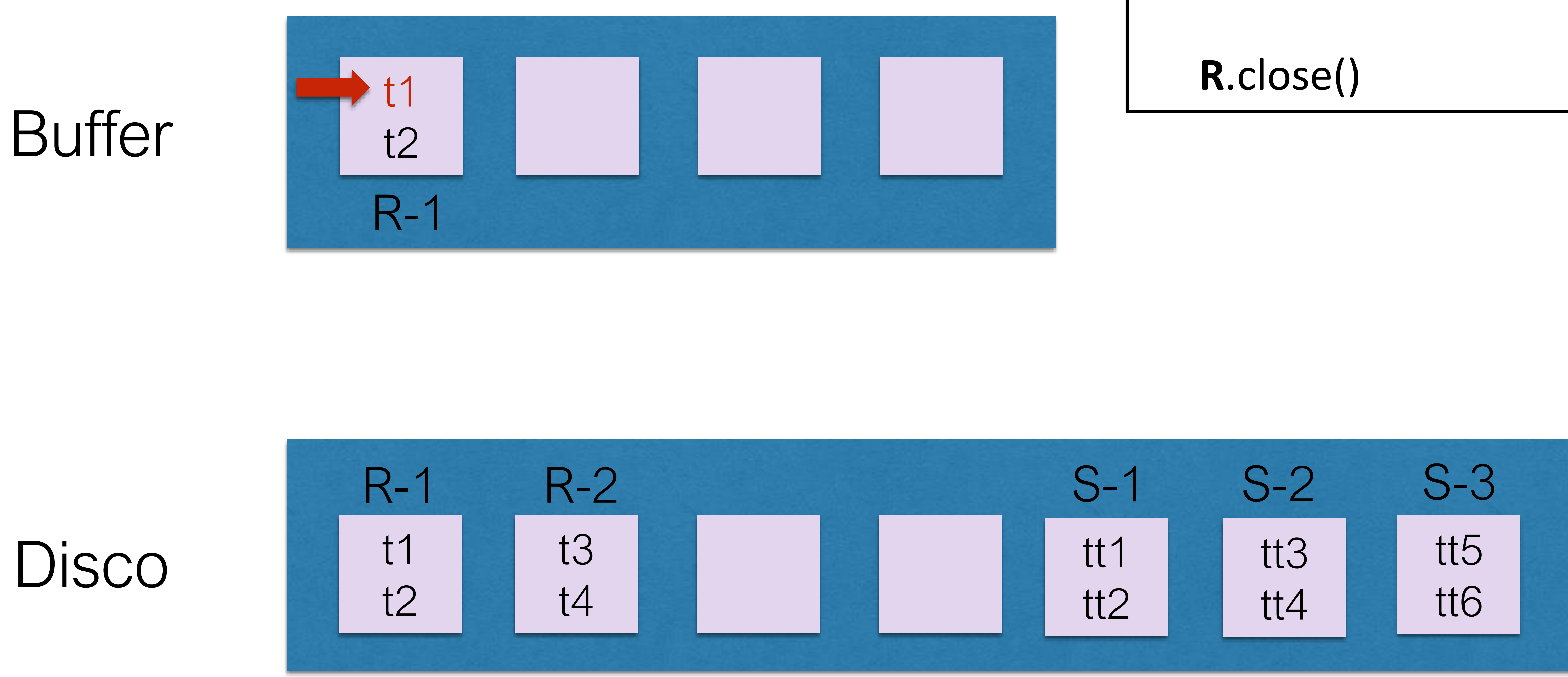
DB



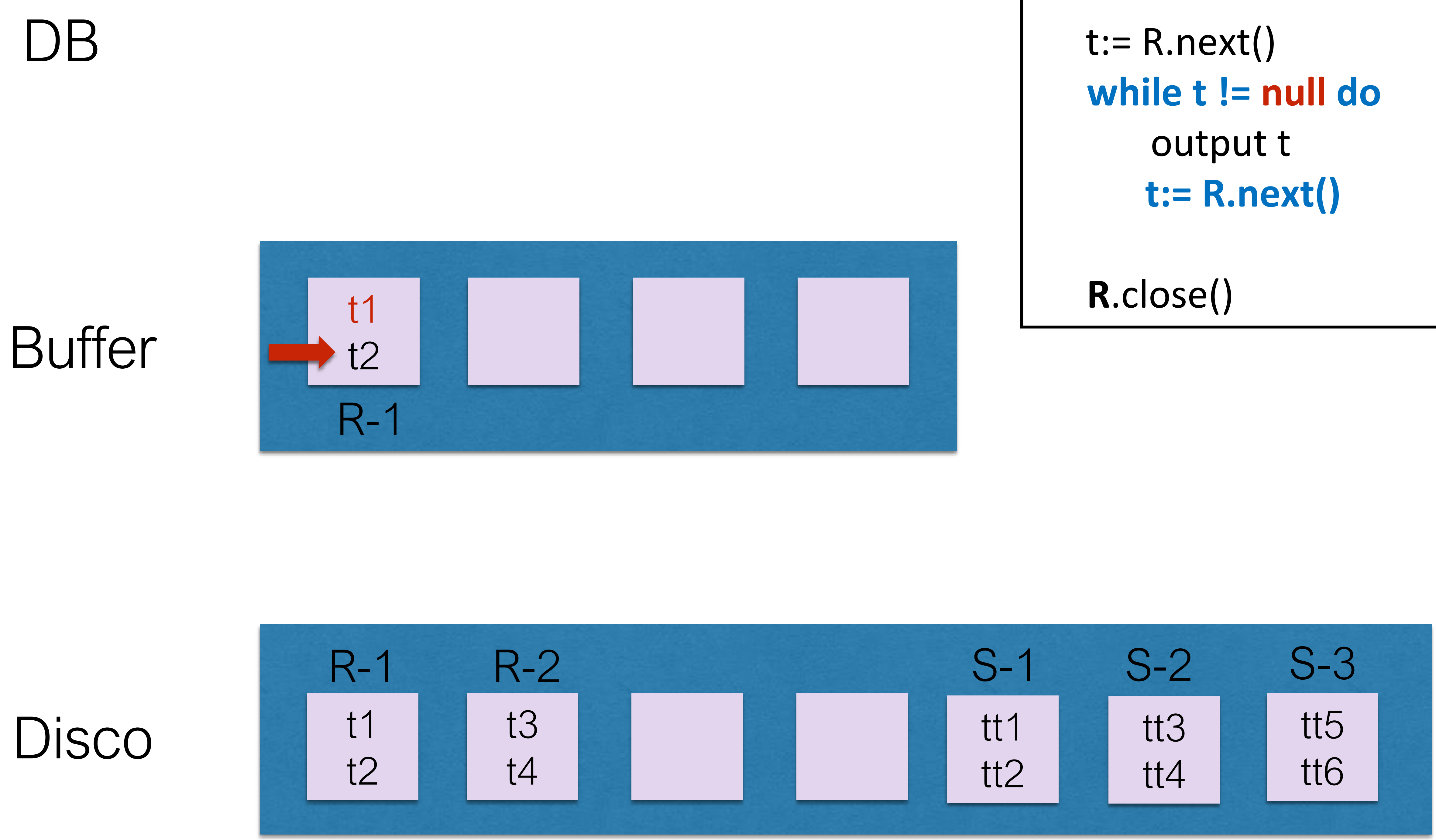
SELECT * FROM R

```
R.open()

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

R.close()
```

DB



SELECT * FROM R

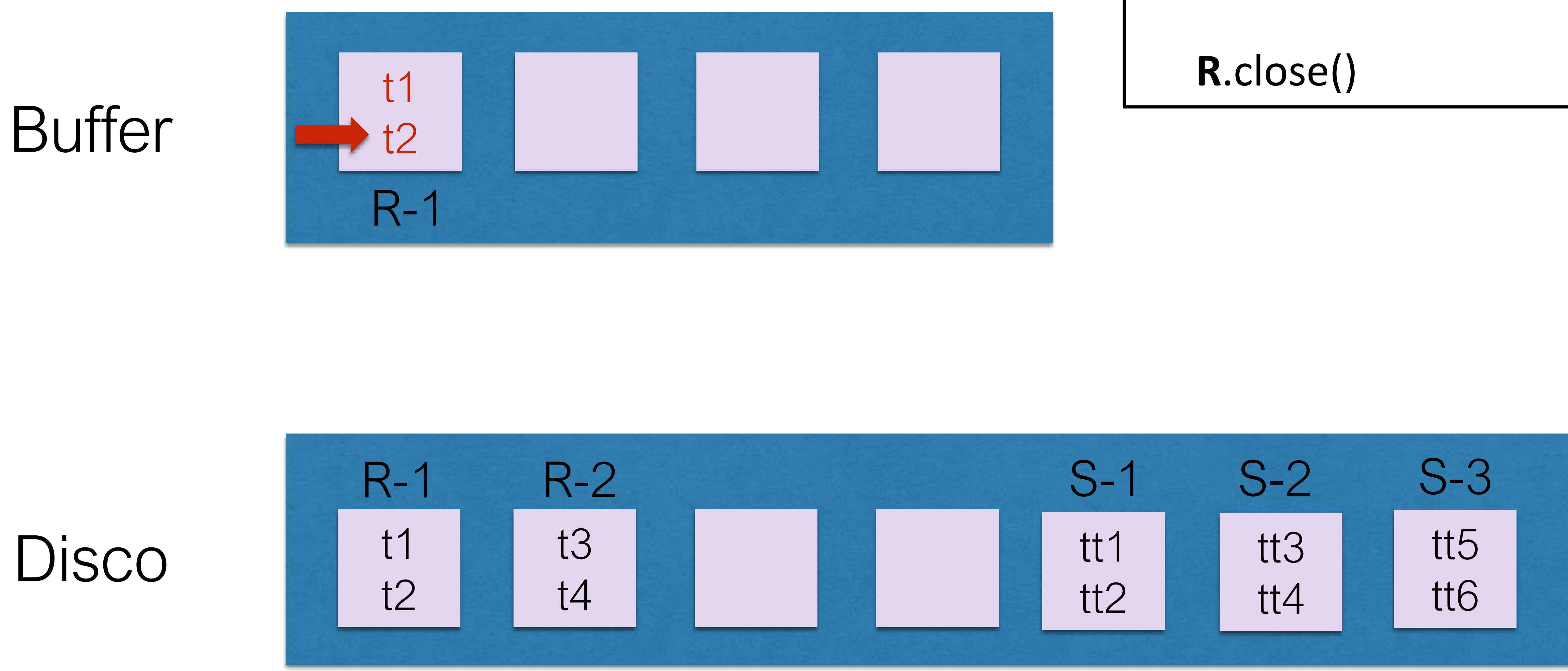



```
R.open()

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

R.close()
```

SELECT * FROM R

```
R.open()

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

R.close()
```

DB



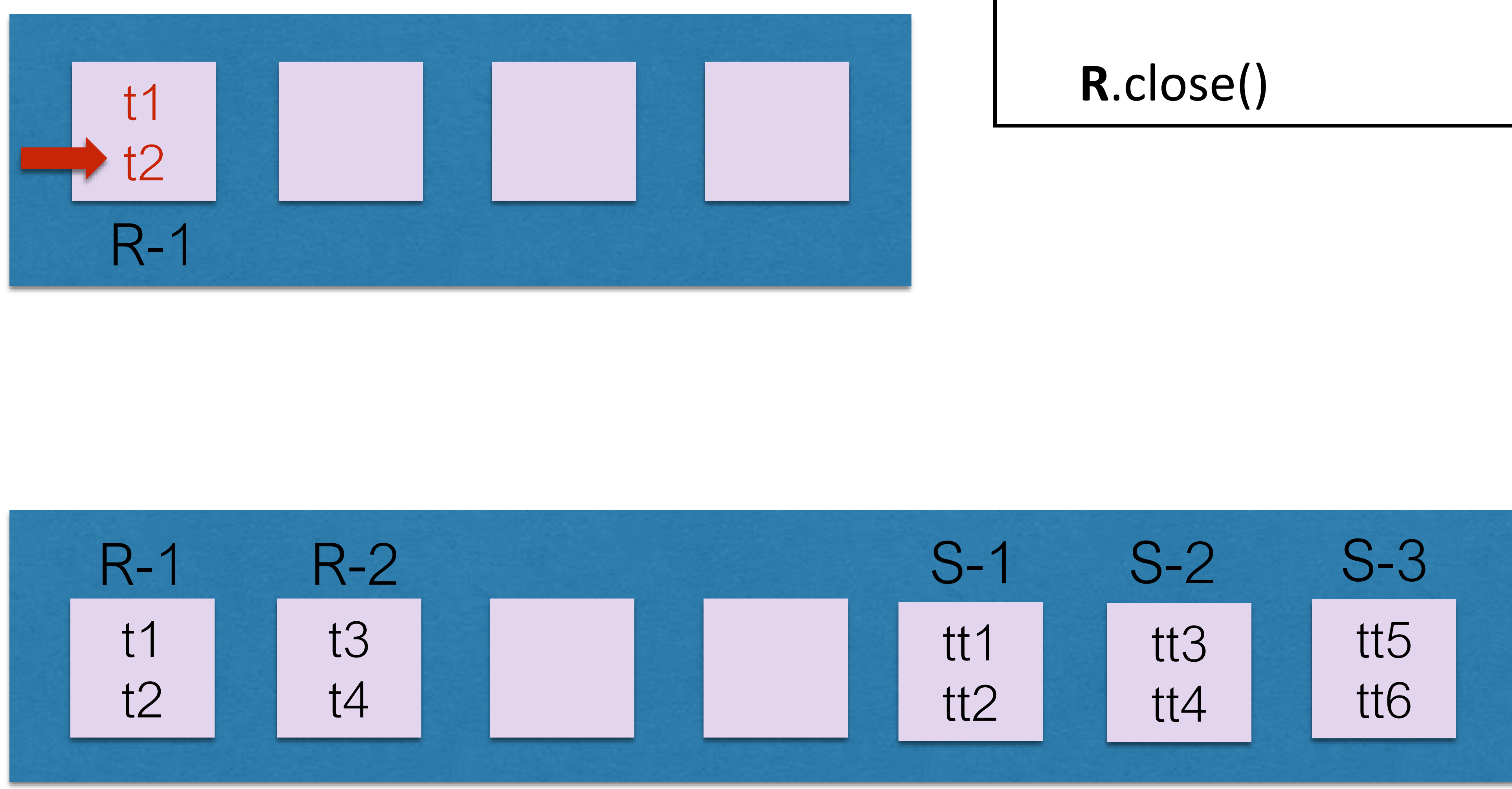
SELECT * FROM R

```
R.open()

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

R.close()
```

DB

Buffer

Disco

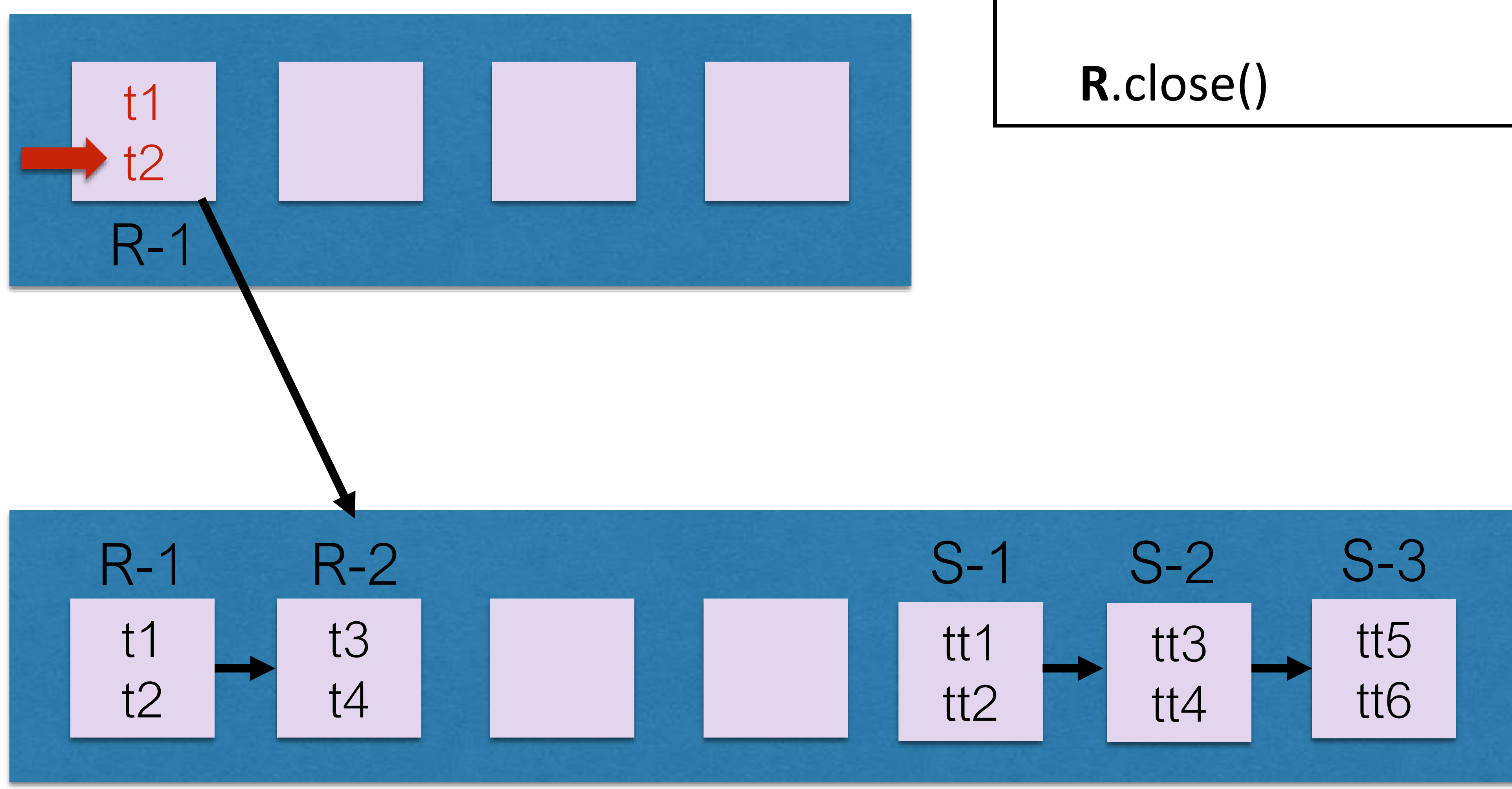


SELECT * FROM R

```
R.open()

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

R.close()
```

DB

Buffer

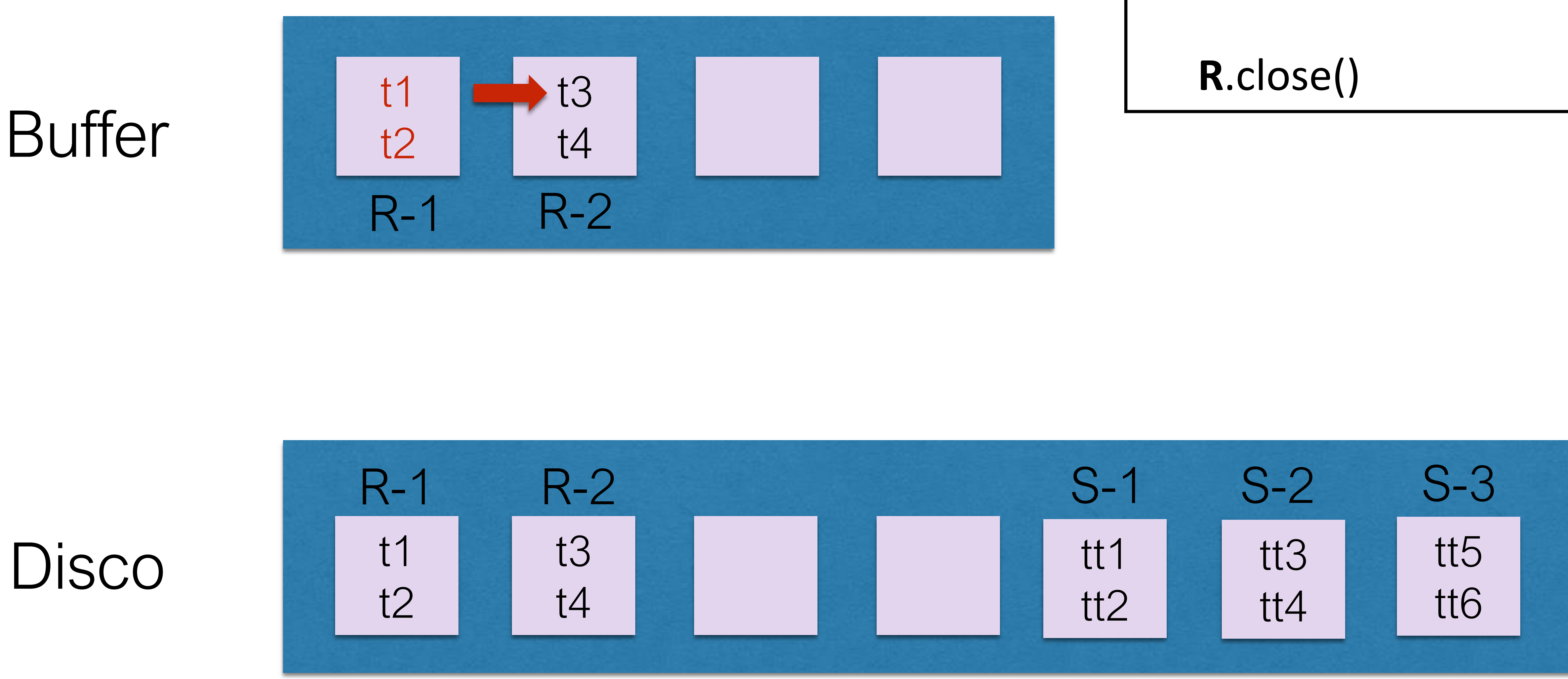


Disco

SELECT * FROM R

```
R.open()

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

R.close()
```

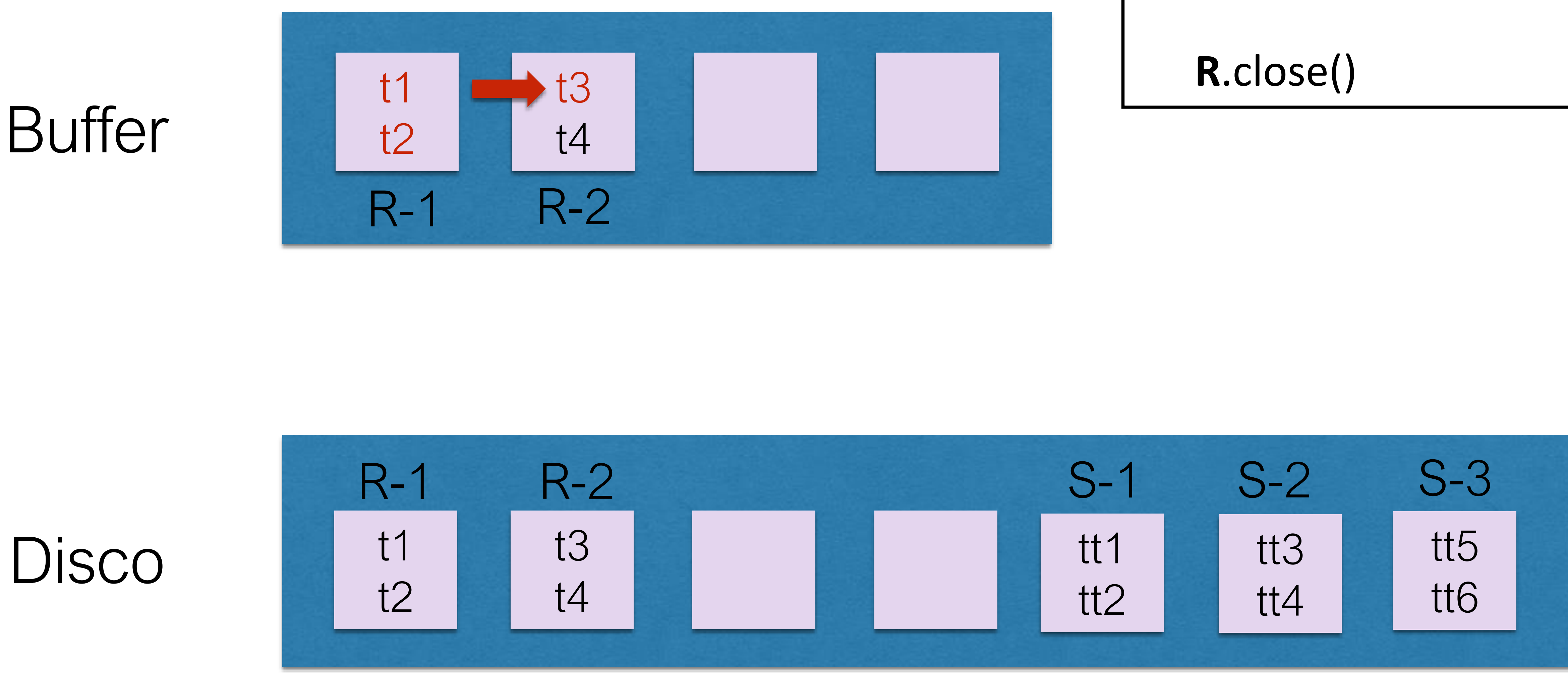
DB



SELECT * FROM R

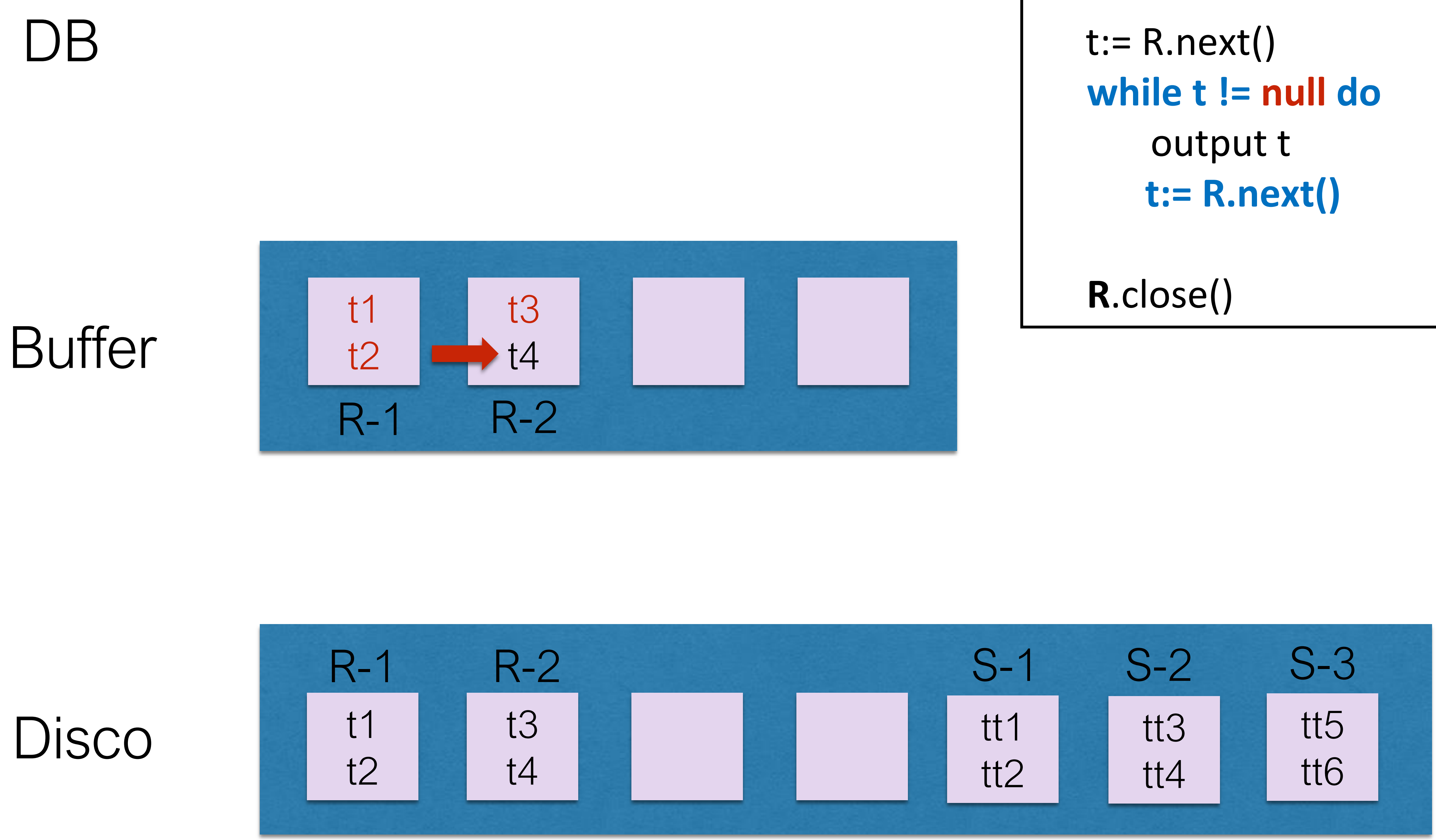
```
R.open()

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

R.close()
```

DB



SELECT * FROM R

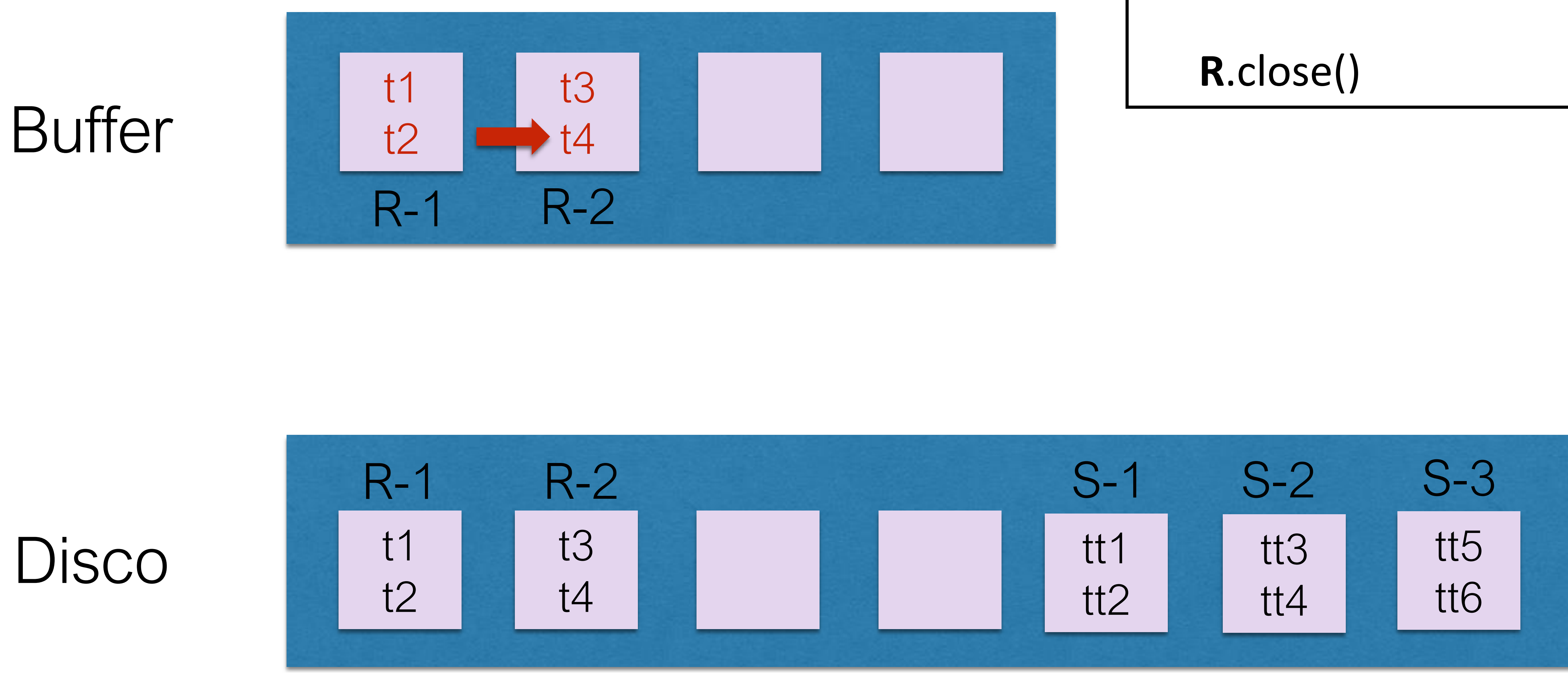
```
R.open()

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

R.close()
```



SELECT * FROM R

```
R.open()

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

R.close()
```

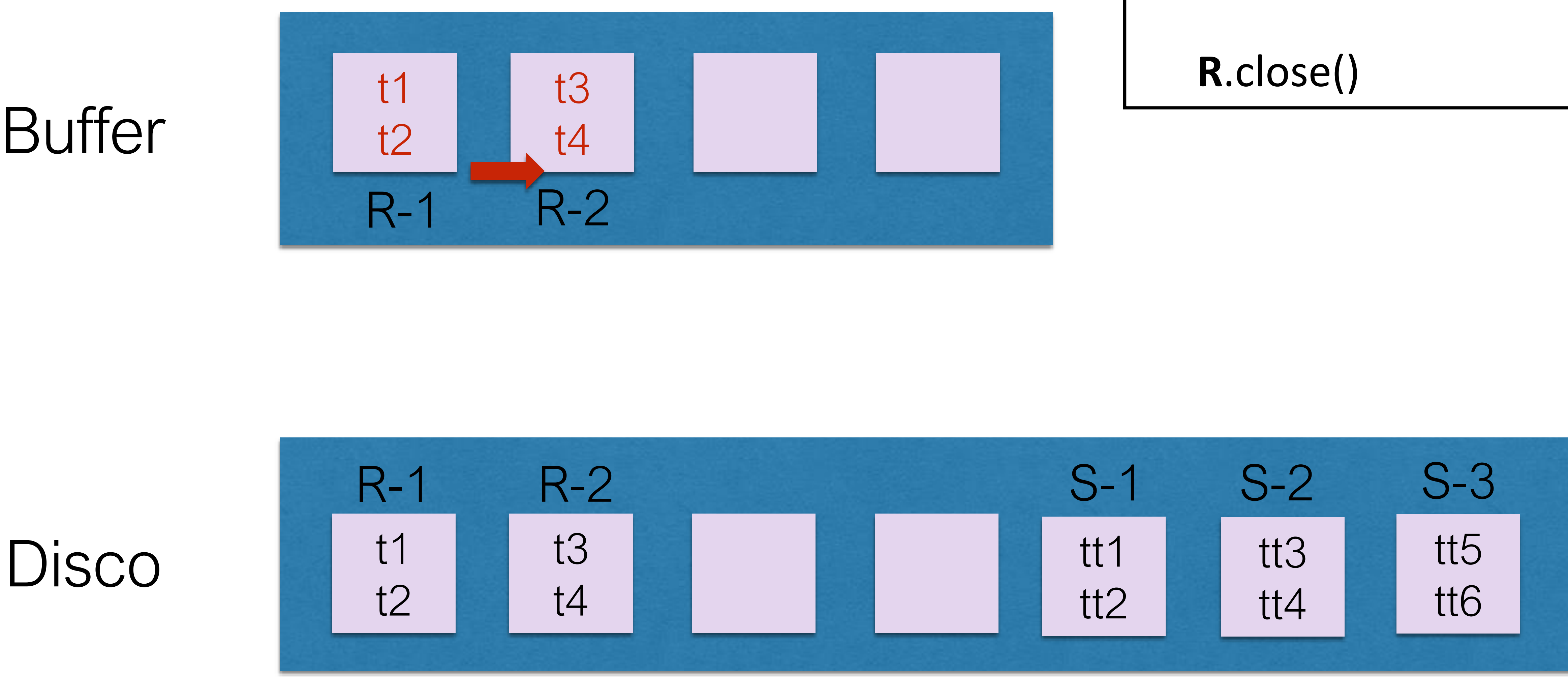
DB



SELECT * FROM R

```
R.open()

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

R.close()
```

DB



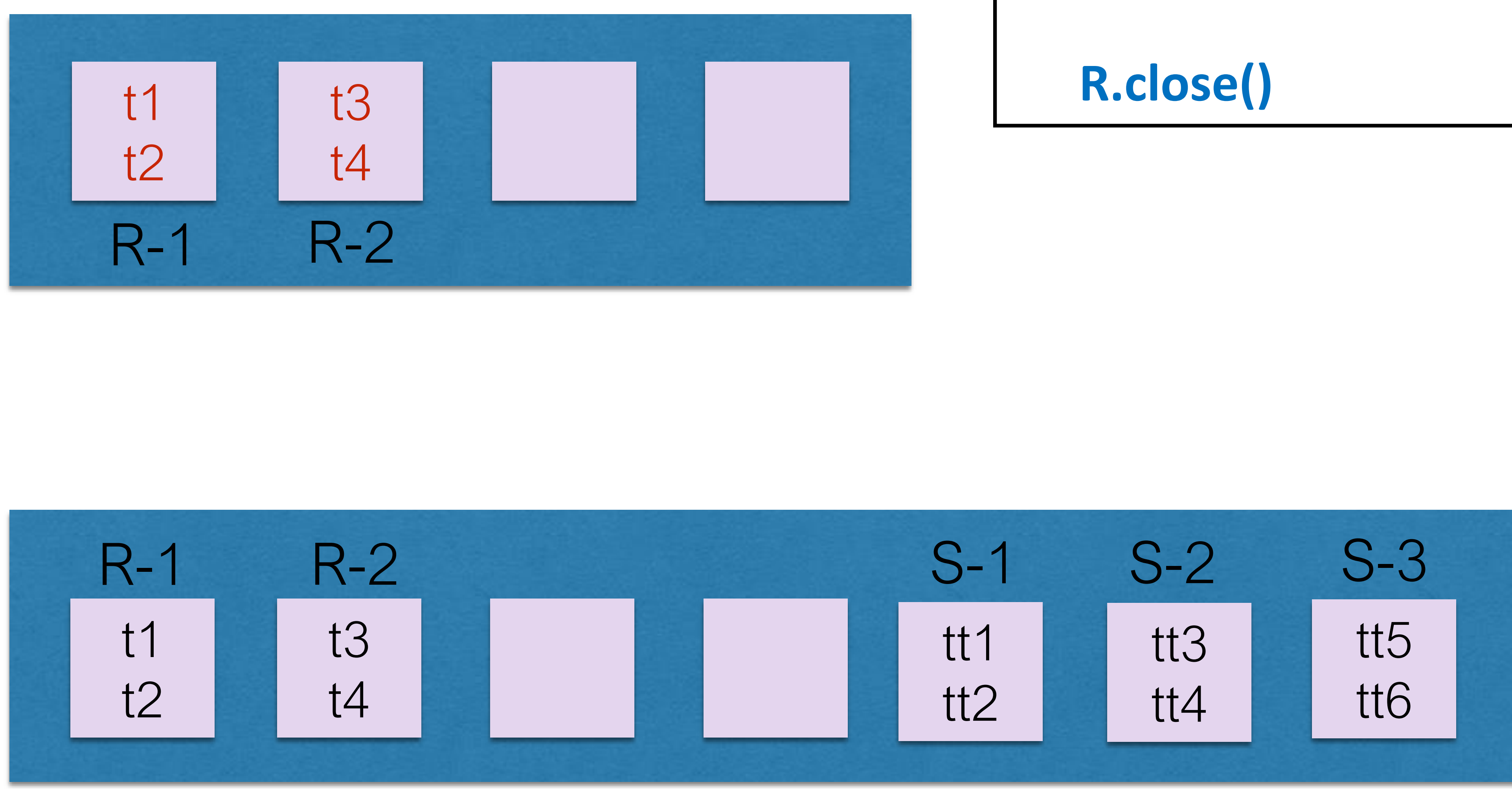
SELECT * FROM R

```
R.open()

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

R.close()
```

DB

Buffer

Disco



En realidad

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

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

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

Implementa 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

close()
    R.close()
```

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 índice

Necesariamente tenemos que recorrer todo **R**

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

Con índice y consulta de rango

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

Hint Existe un índice especial para hacer esto

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 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

Join = R⋈$_p$S

Join.open()

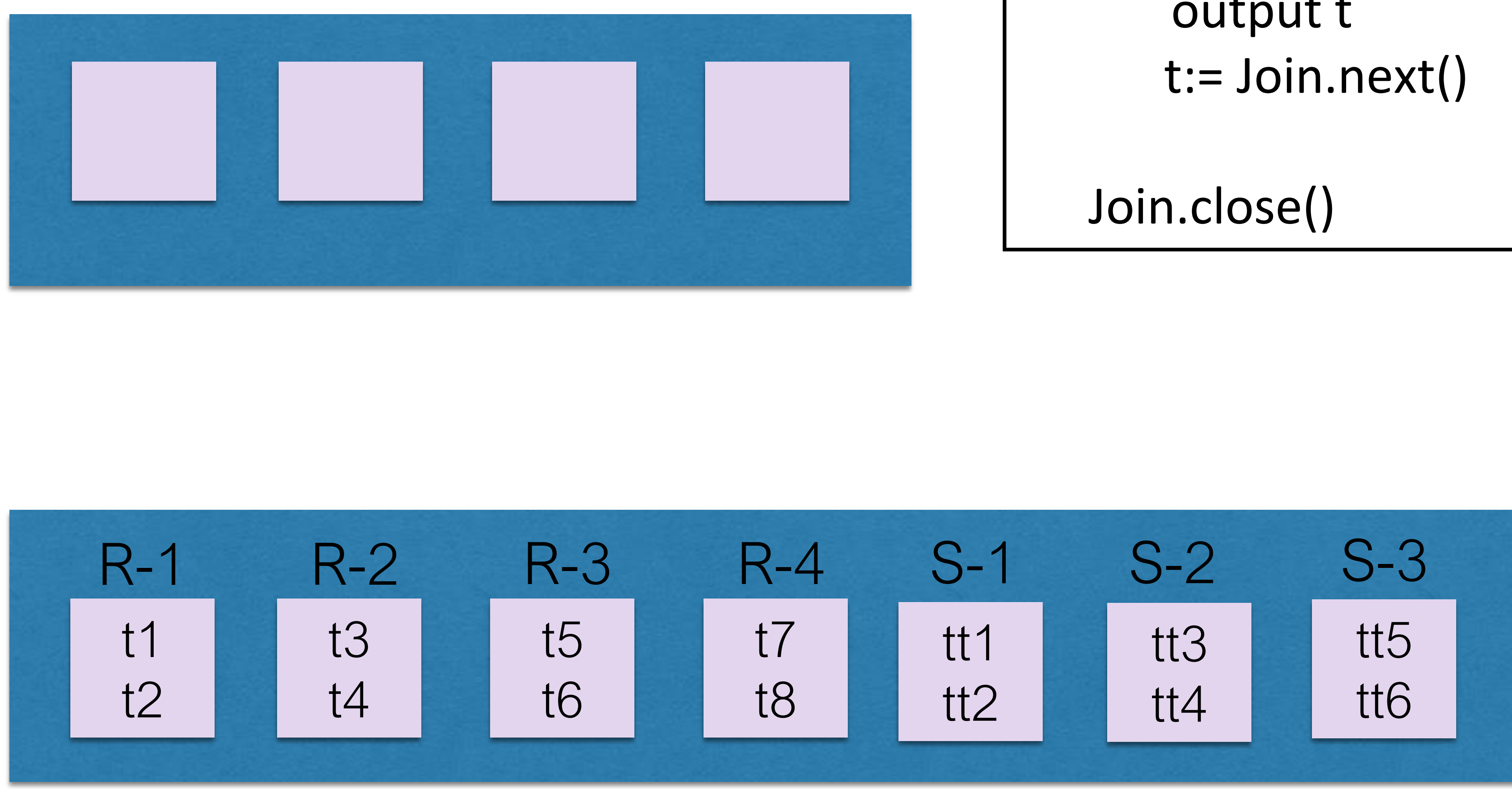
t:= Join.next()
while t != null do
    output t
    t:= Join.next()

Join.close()

DB

Buffer

Disco



Nested Loop Join

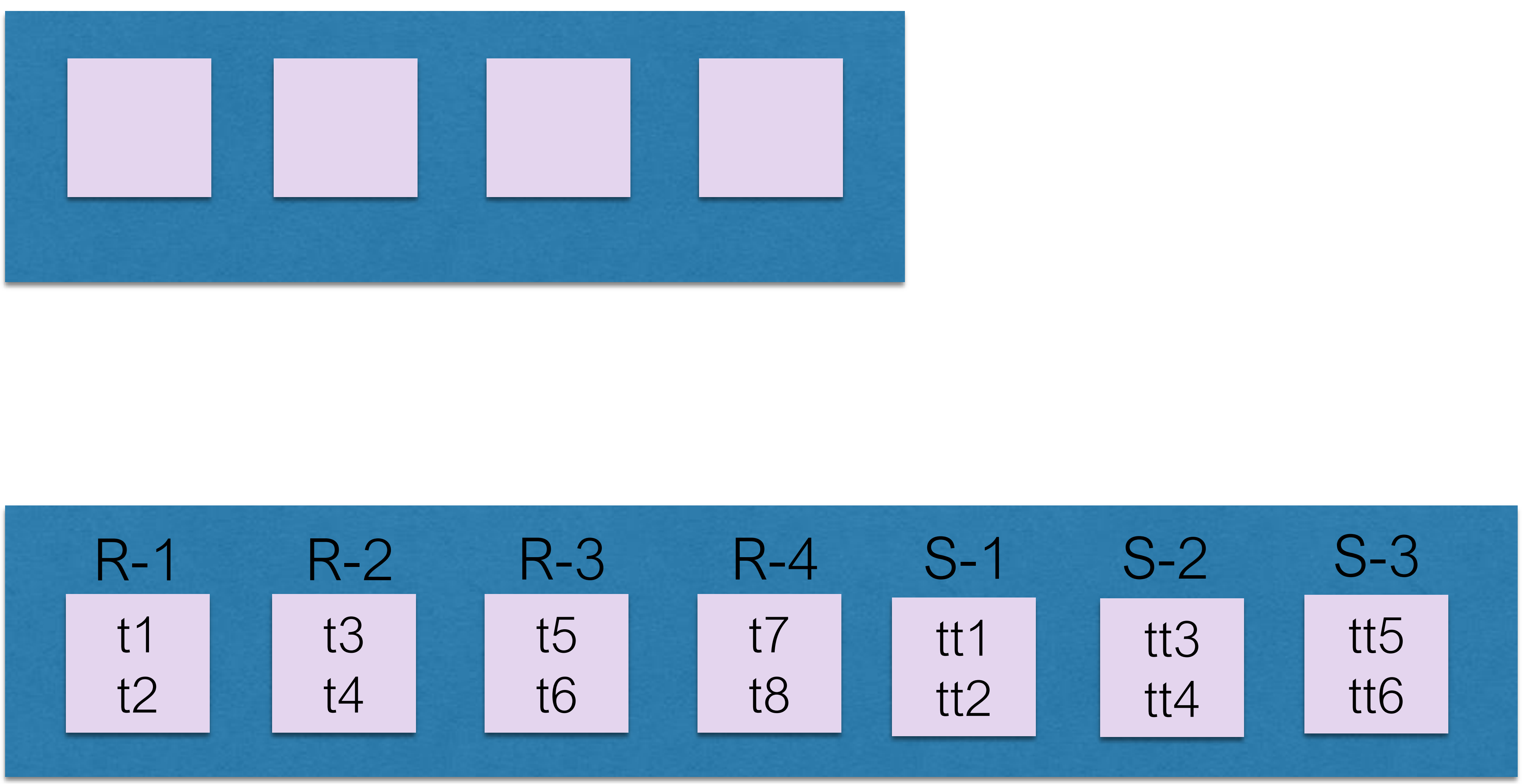
DB

$R \bowtie_p S$ …Join.open()

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

Buffer

Disco

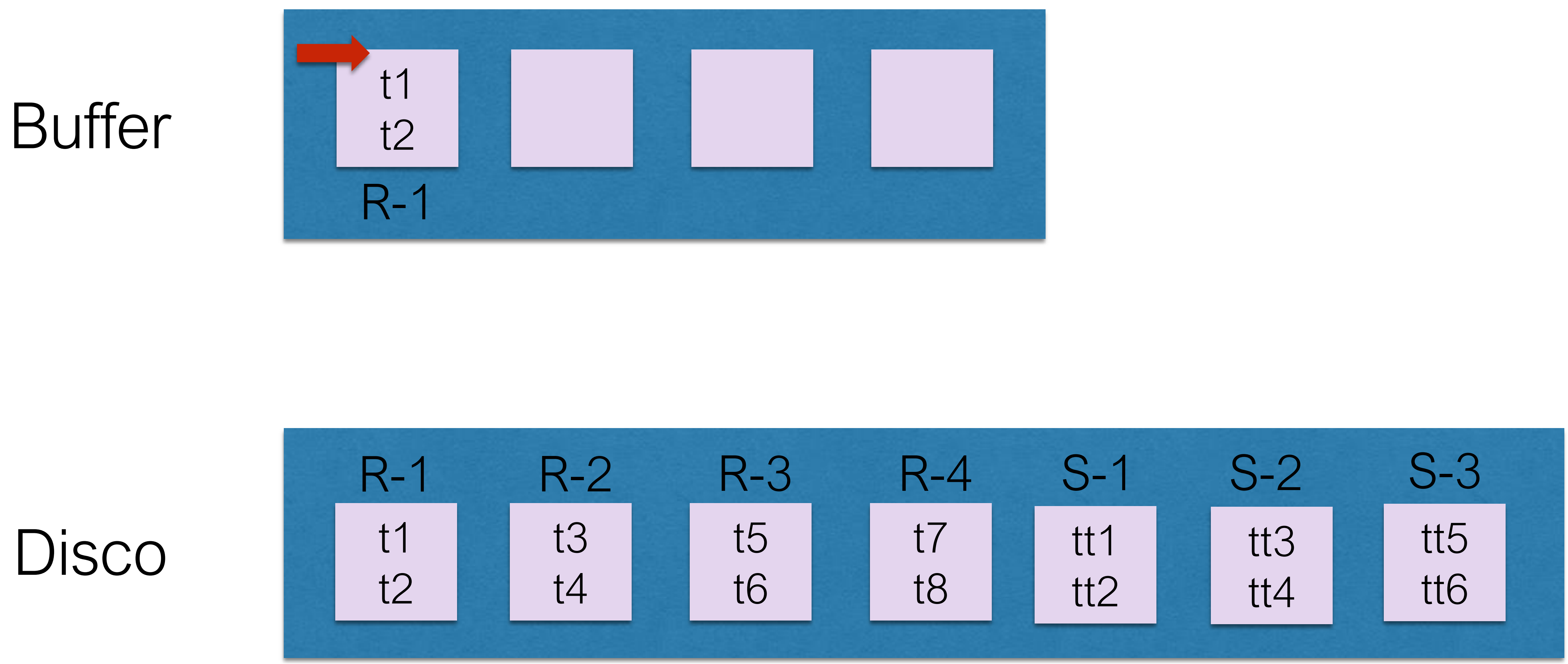


Nested Loop Join

DB

R⋈$_p$S …Join.open()

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



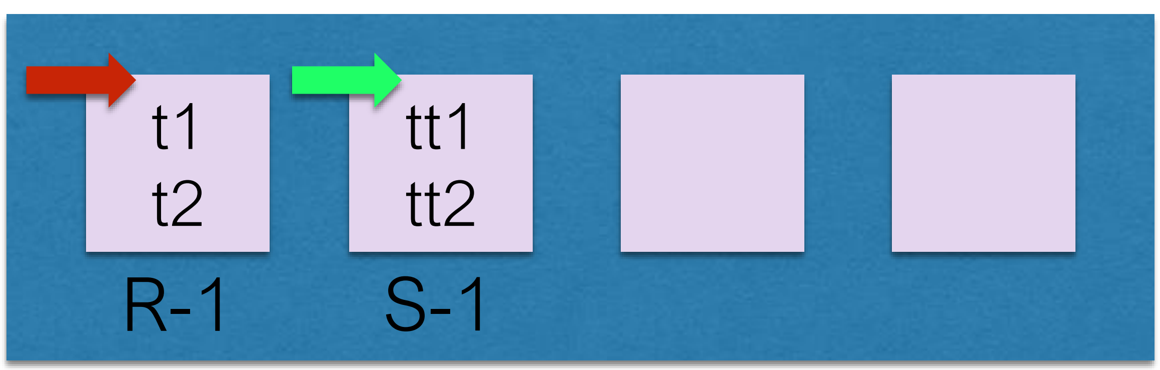
Nested Loop Join

DB

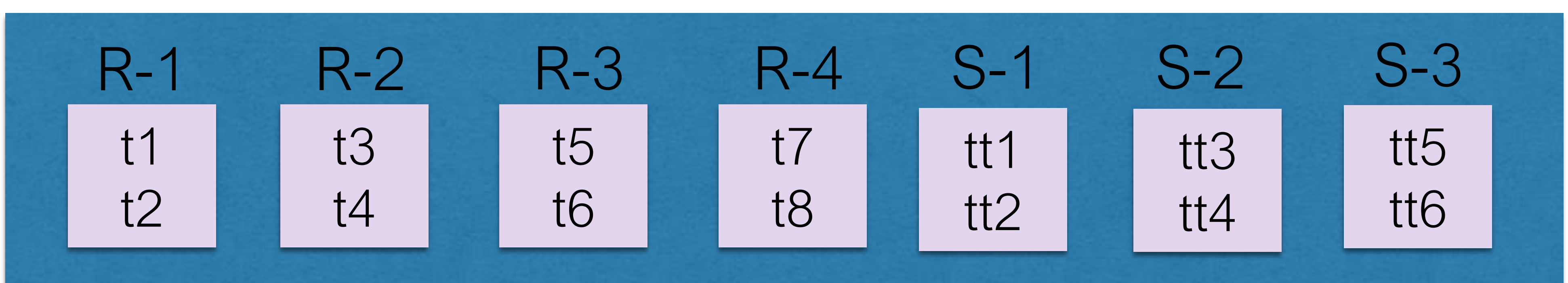
$R \bowtie_p S$ …Join.open()

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

Buffer



Disco



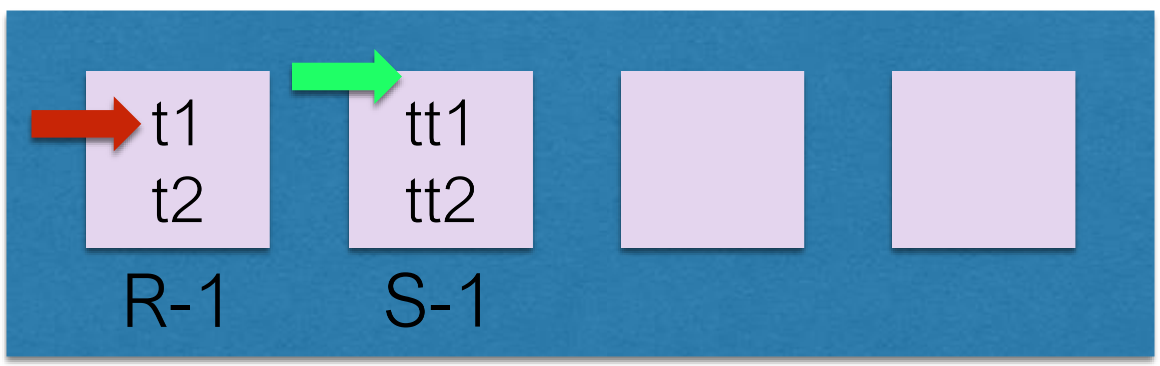
Nested Loop Join

DB

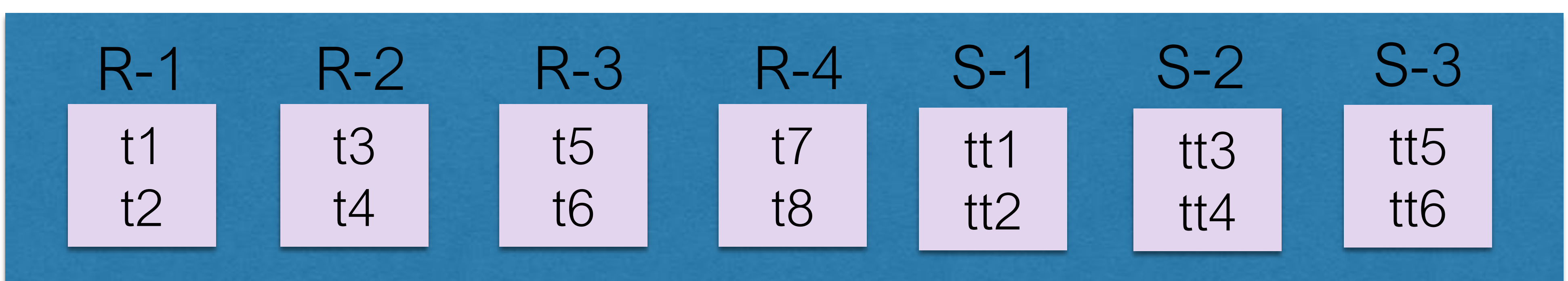
$R \bowtie_p S$ …Join.open()

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

Buffer



Disco

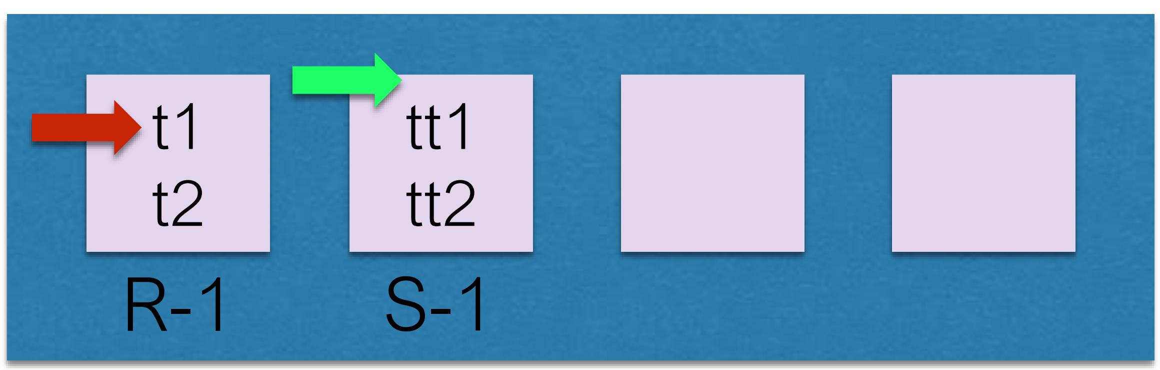


Nested Loop Join

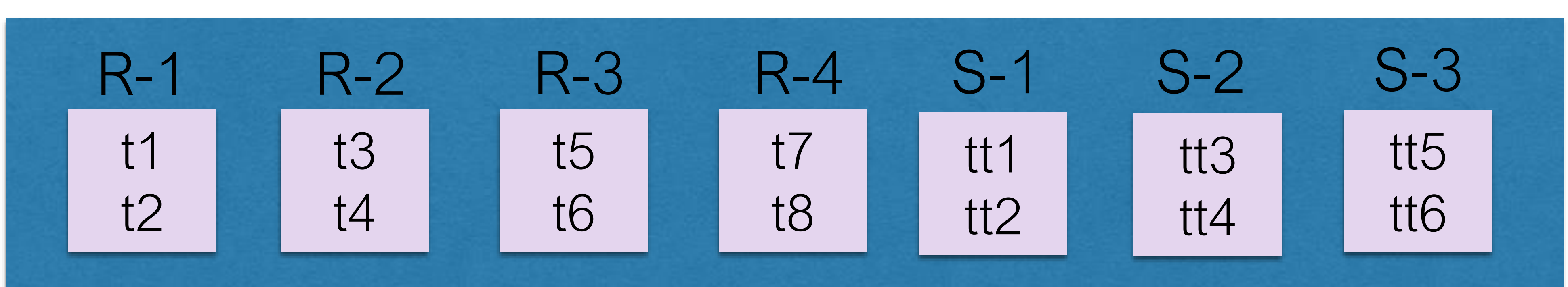
DB

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

Buffer



Disco



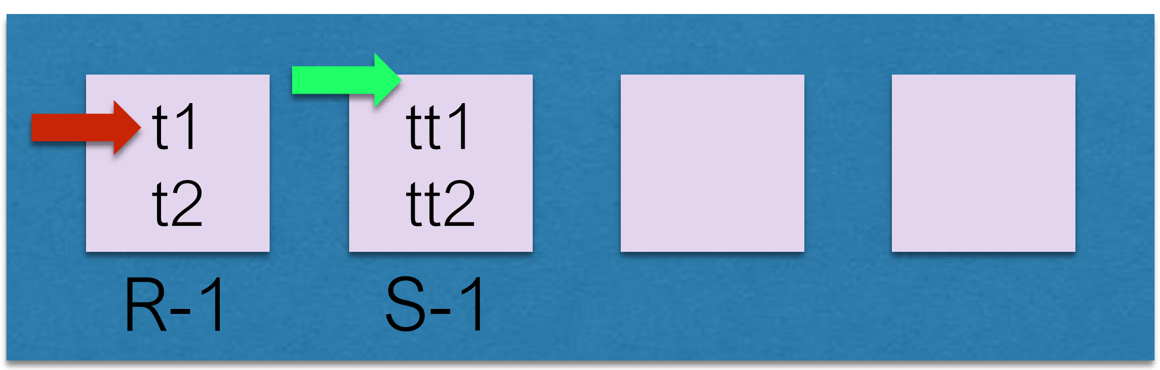
Nested Loop Join

DB

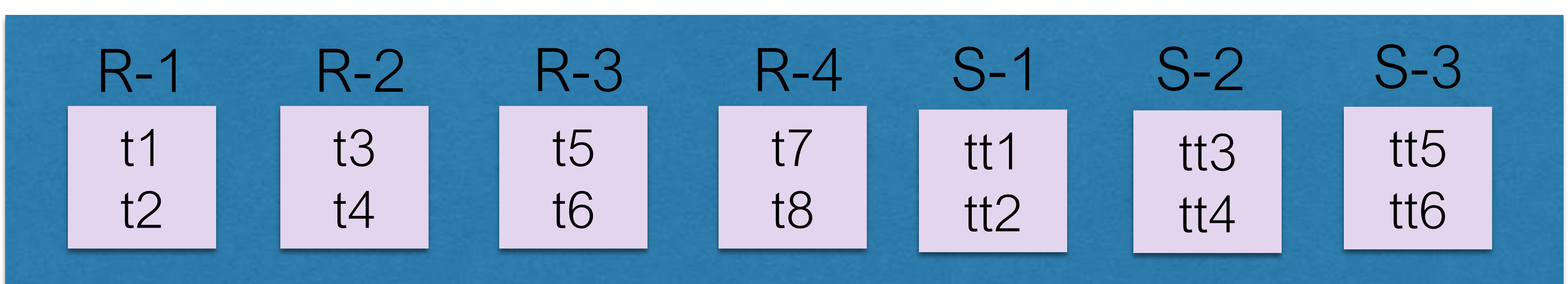
```
t:= Join.next()
while t != null do
    output t
    t:= Join.next()
```

```
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



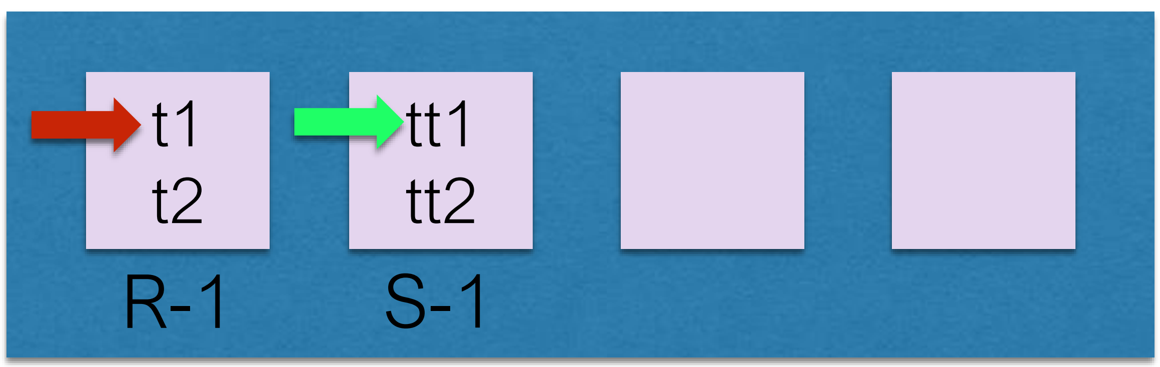
Nested Loop Join

DB

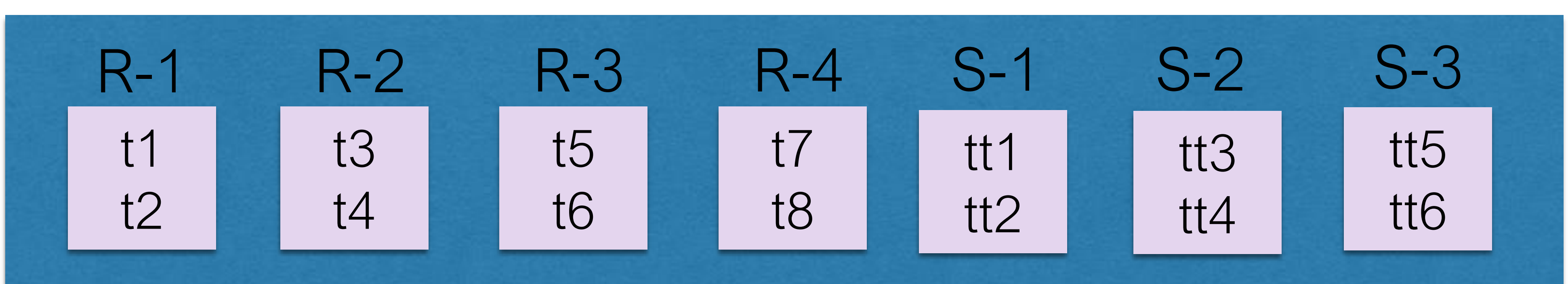
```
t:= Join.next()
while t != null do
    output t
    t:= Join.next()
```

```
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



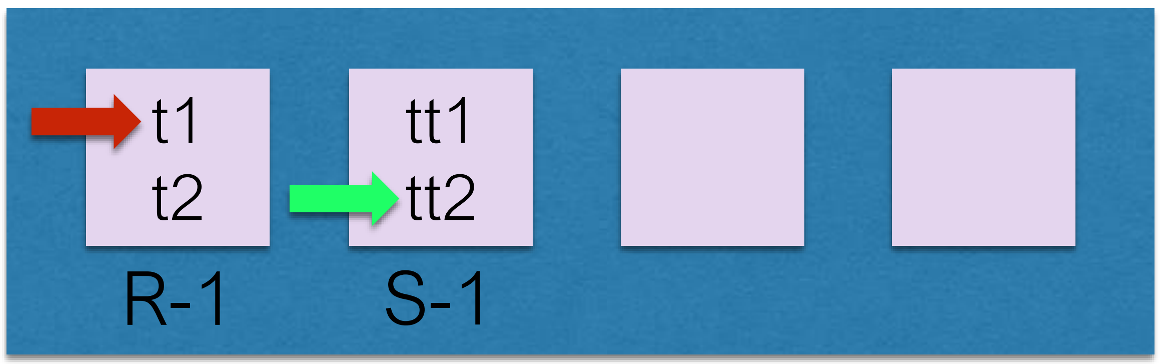
Nested Loop Join

DB

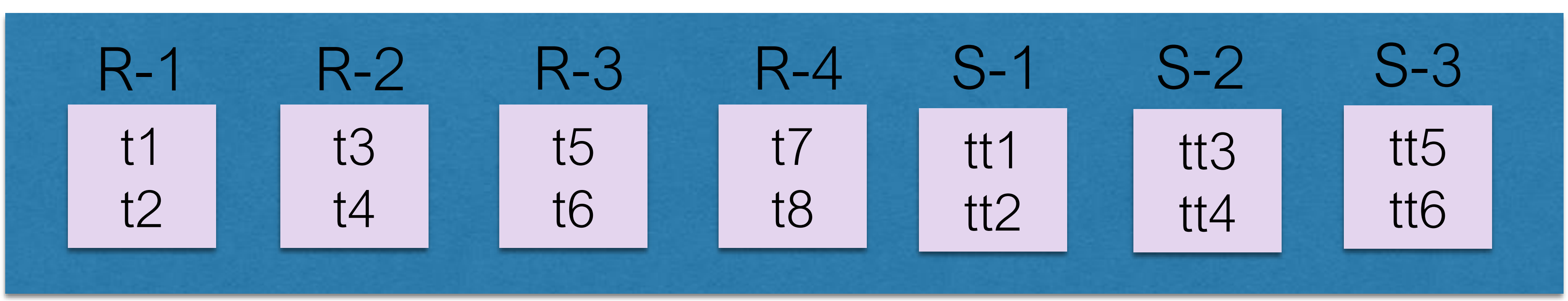
```
t:= Join.next()
while t != null do
    output t
    t:= Join.next()
```

```
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



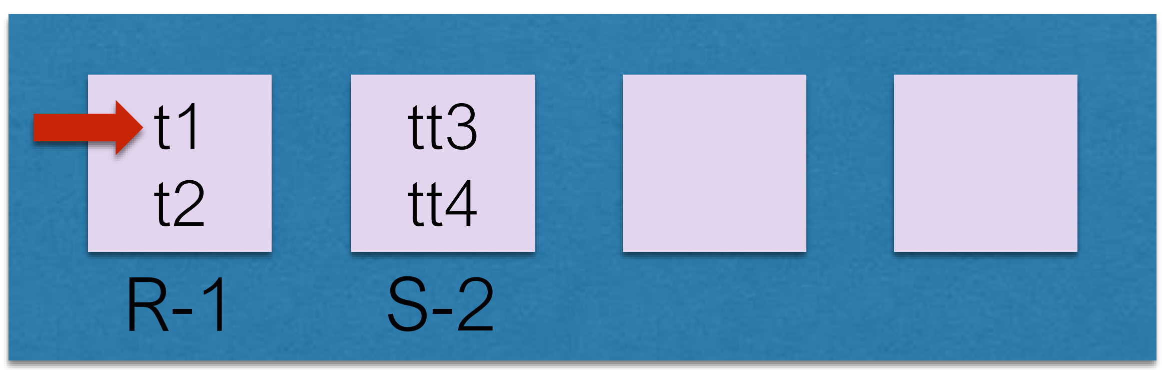
Nested Loop Join

DB

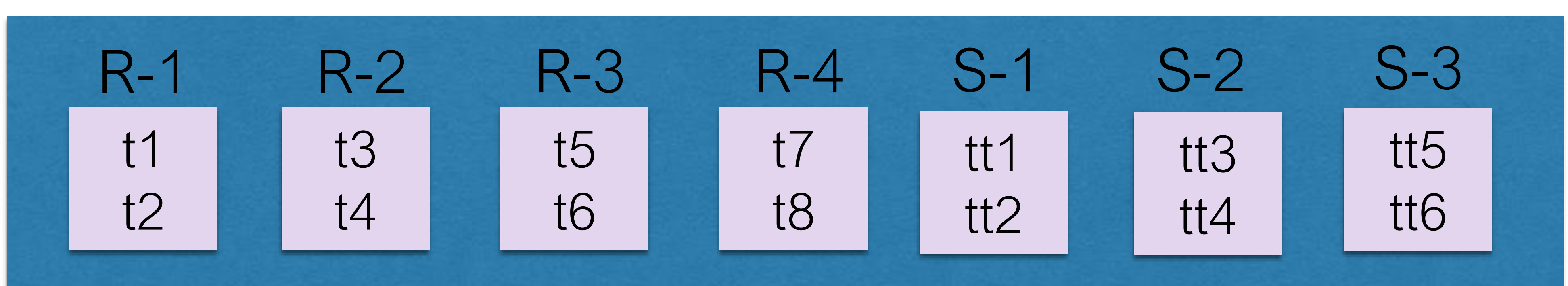
```
t:= Join.next()
while t != null do
    output t
    t:= Join.next()
```

```
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

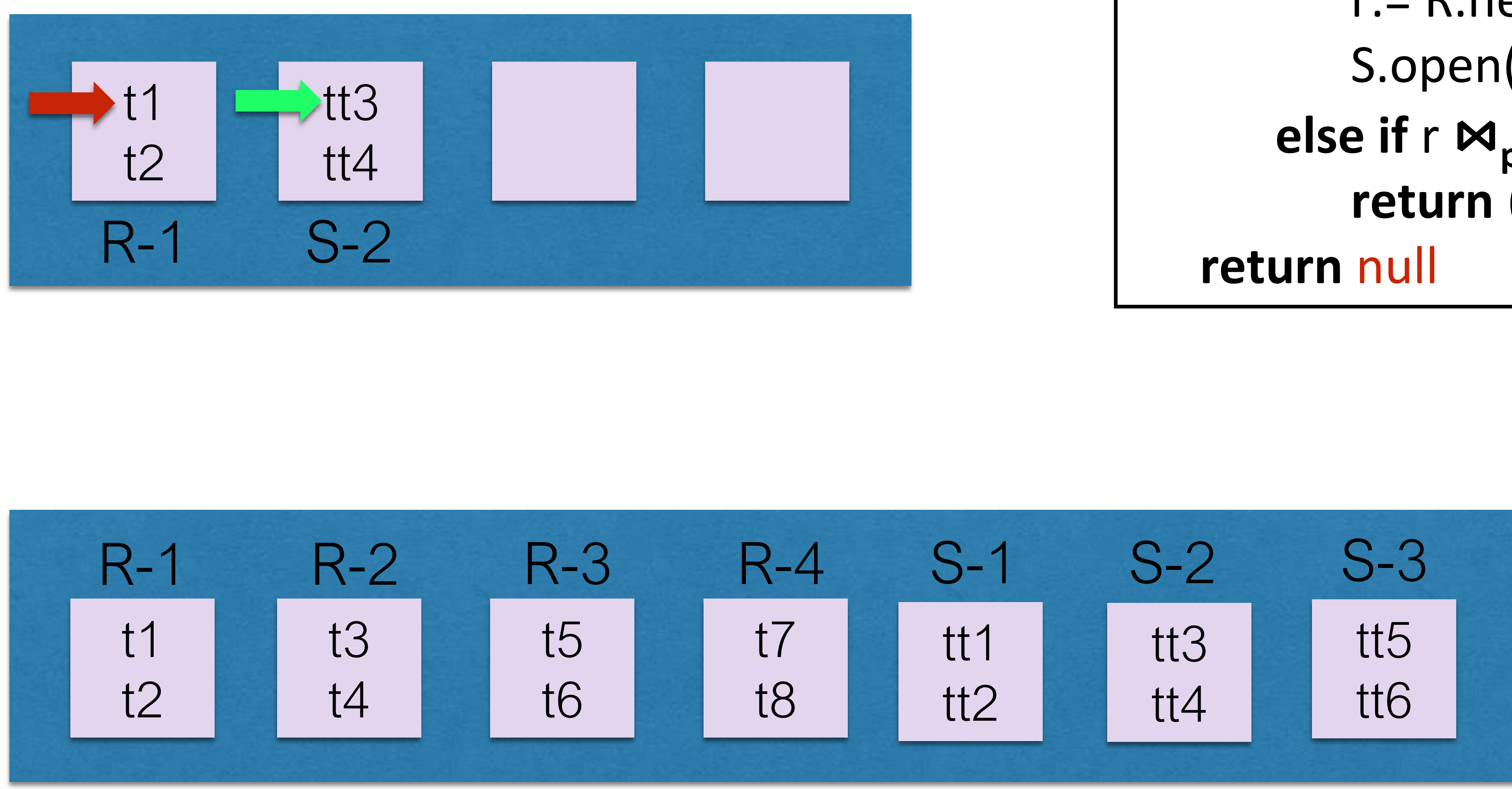
DB

```
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



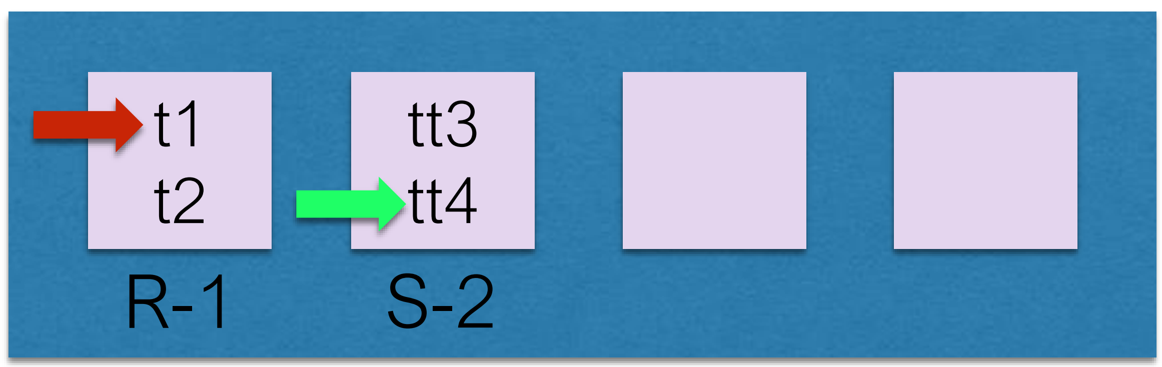
Nested Loop Join

DB

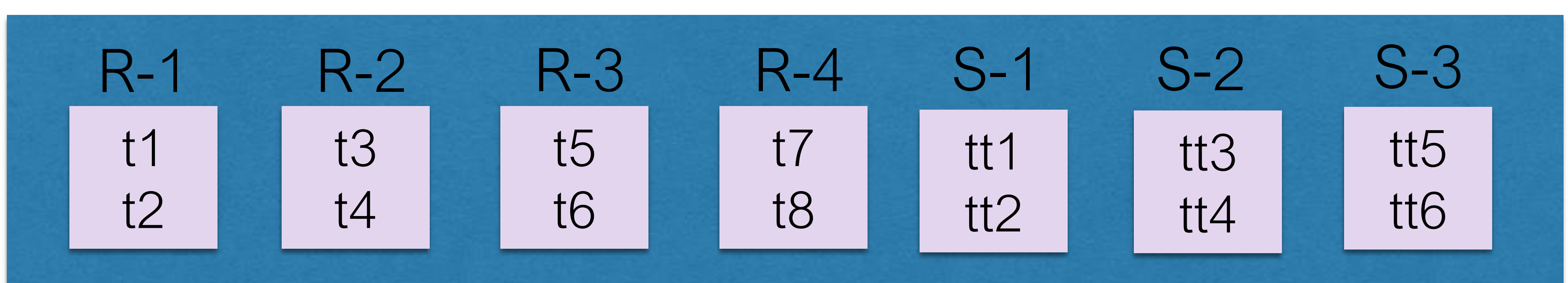
```
t:= Join.next()
while t != null do
    output t
    t:= Join.next()
```

```
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



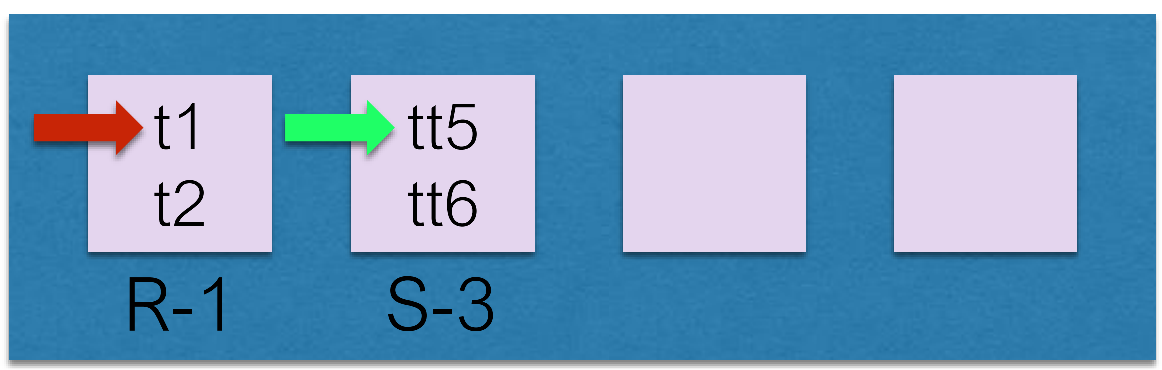
Nested Loop Join

DB

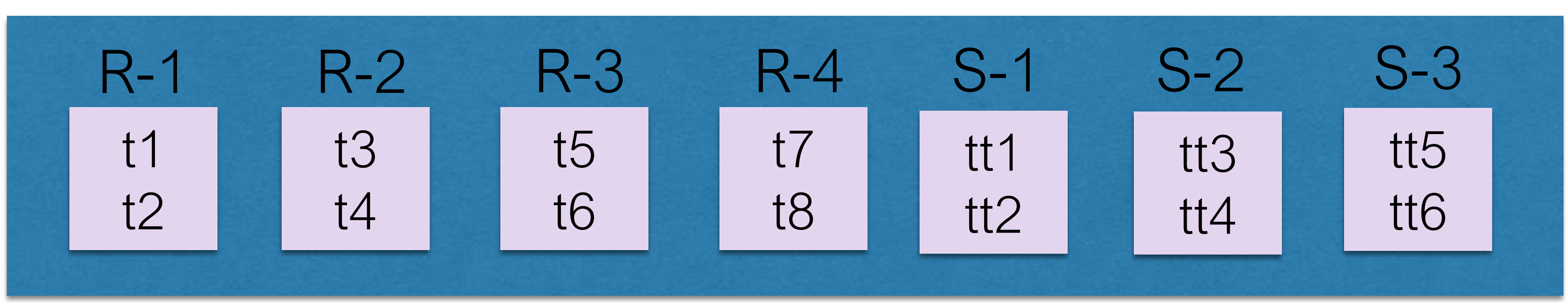
```
t:= Join.next()
while t != null do
    output t
    t:= Join.next()
```

```
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

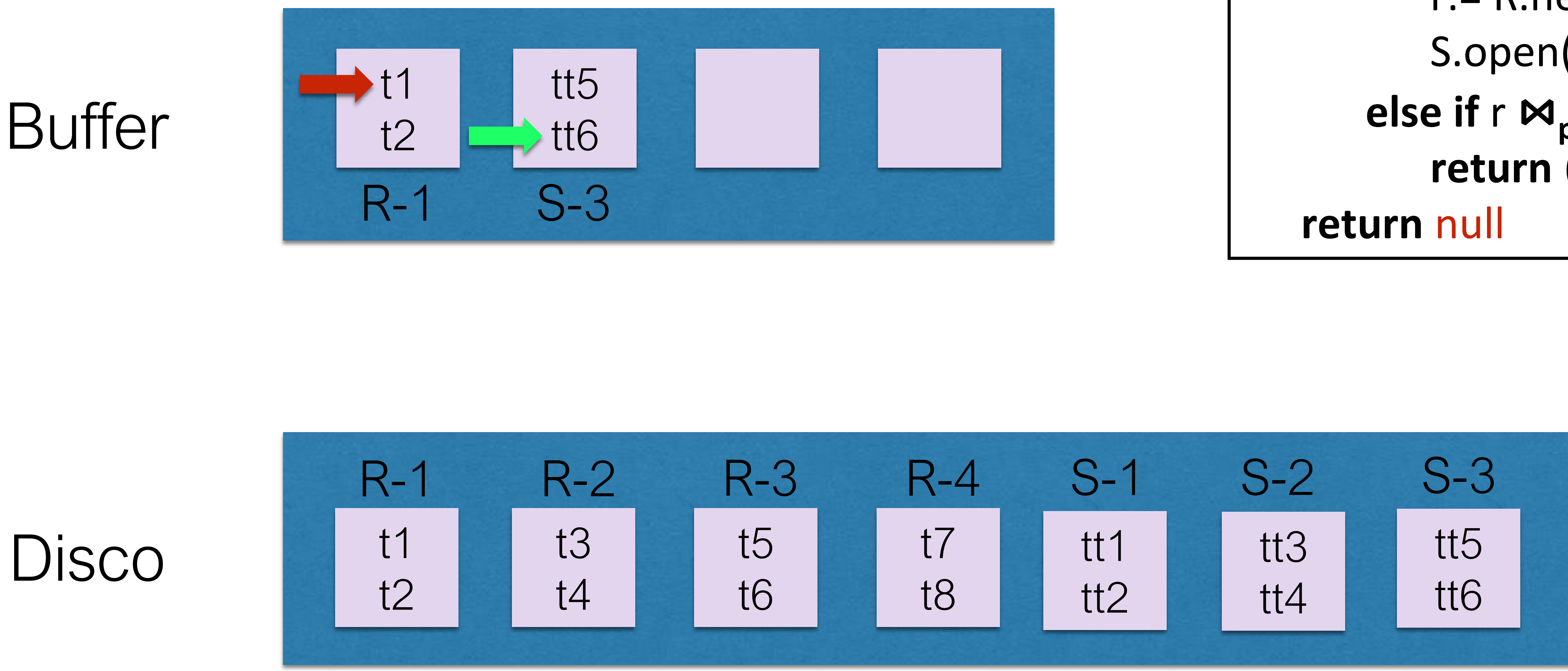


Nested Loop Join

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
```

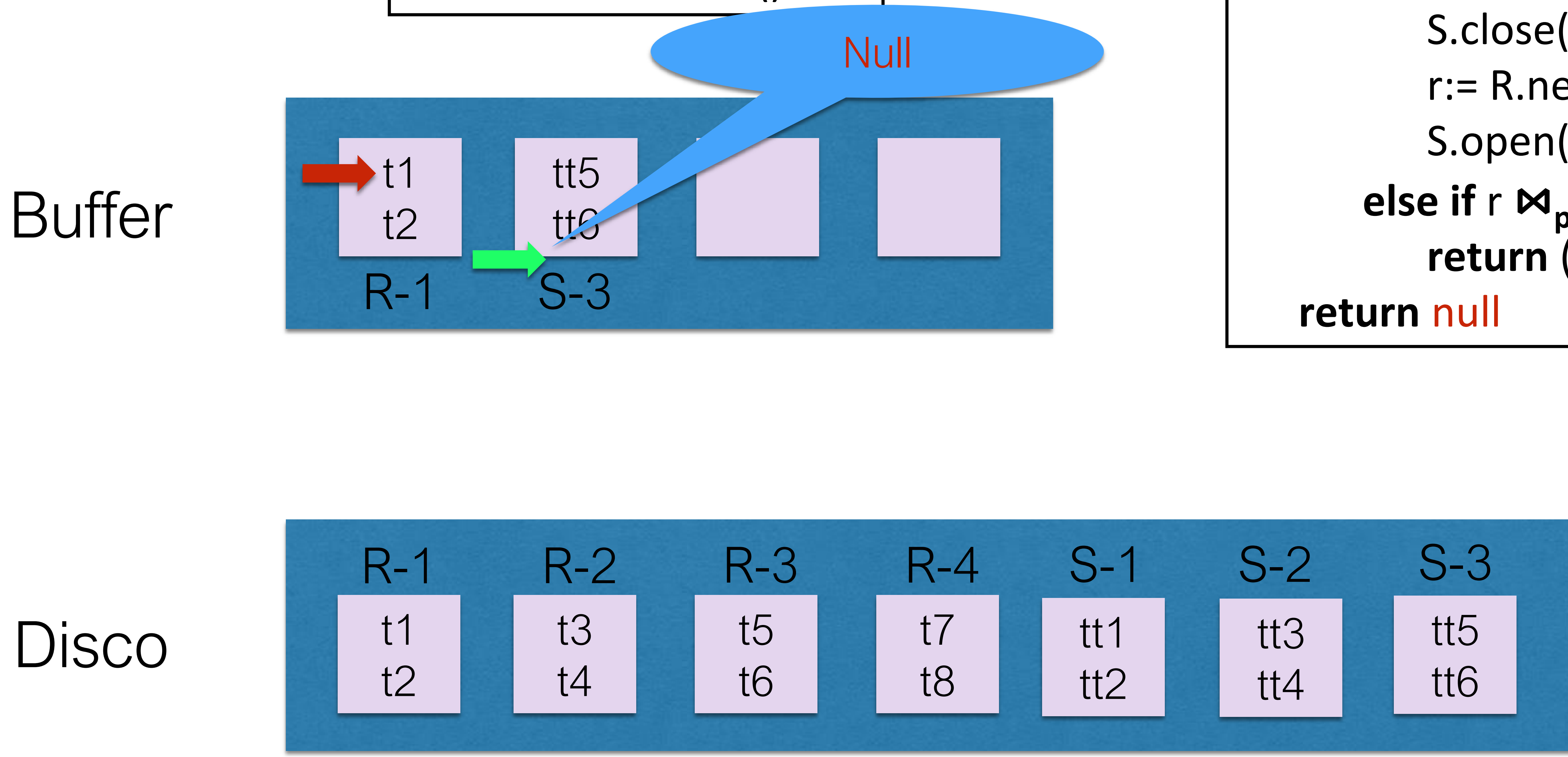


Nested Loop Join

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
```

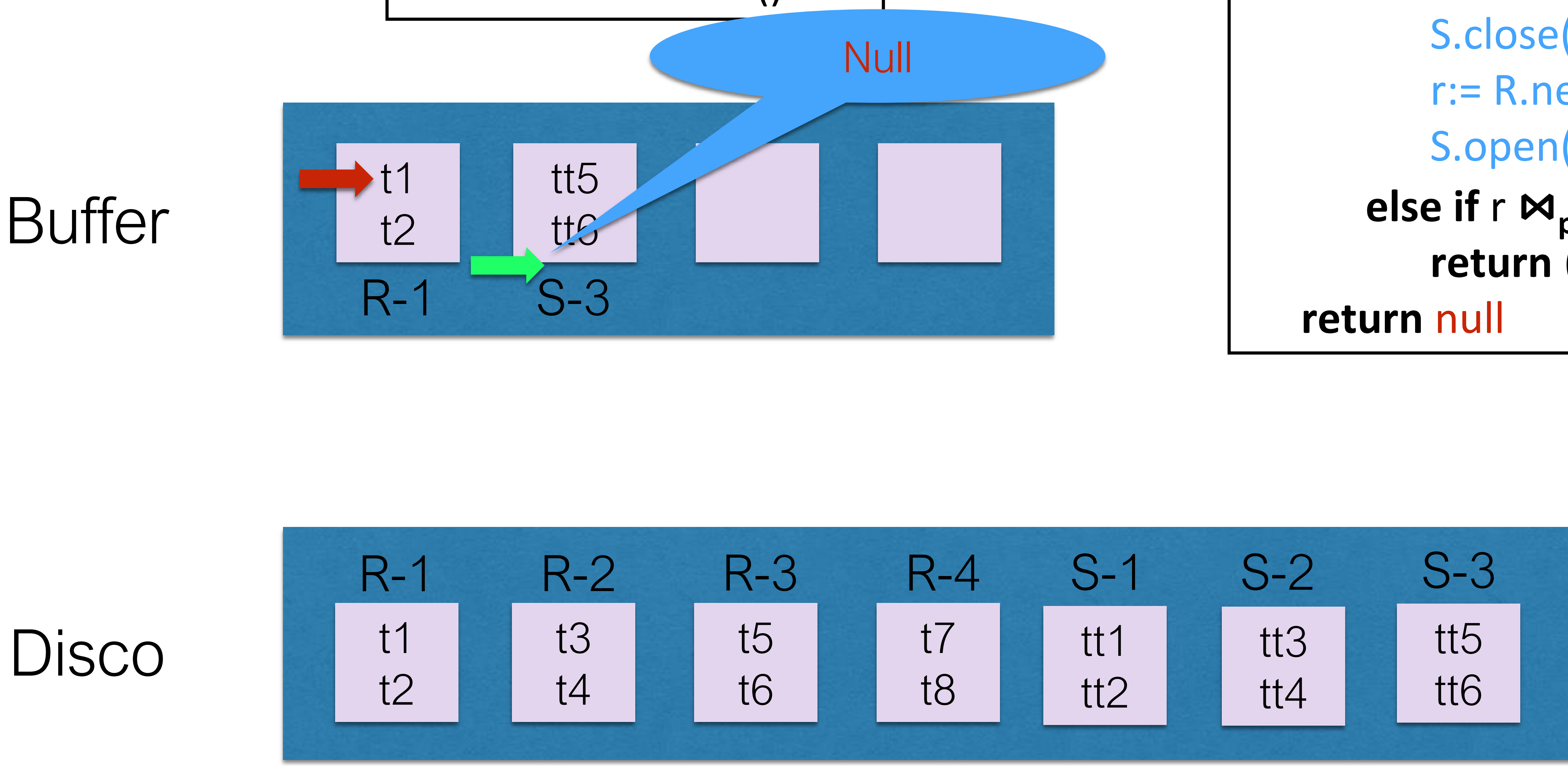


Nested Loop Join

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
```



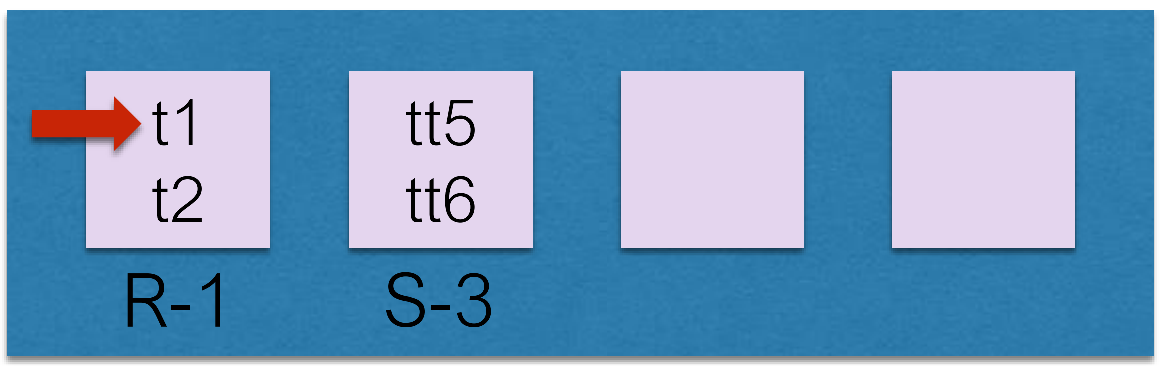
Nested Loop Join

DB

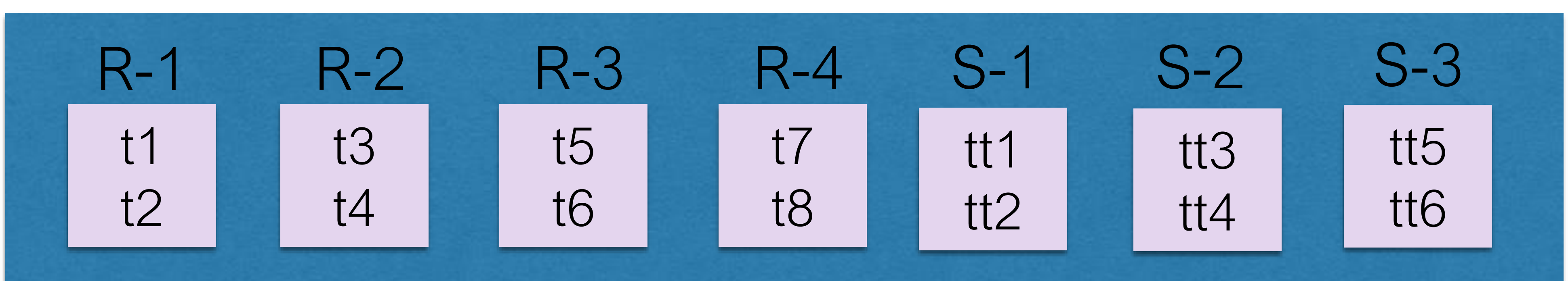
```
t:= Join.next()
while t != null do
      output t
      t:= Join.next()
```

```
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

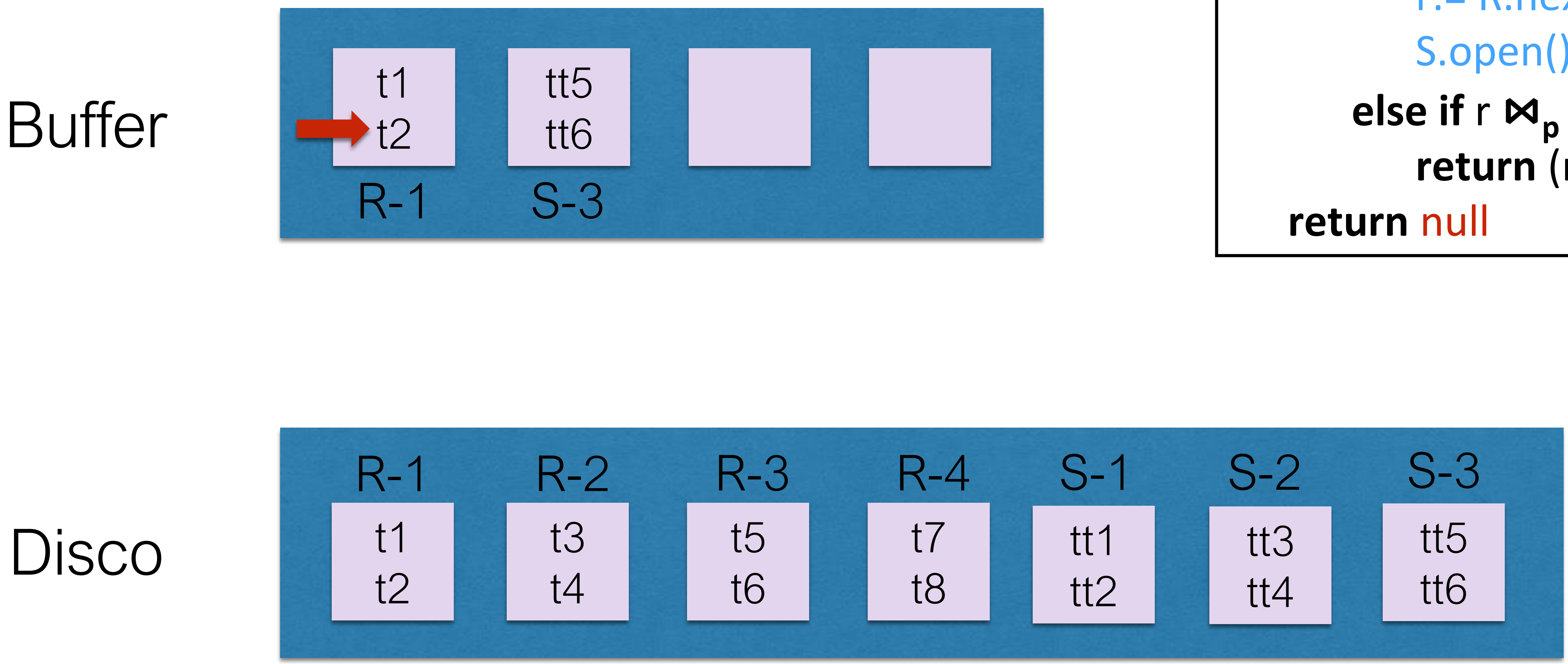


Nested Loop Join

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
```

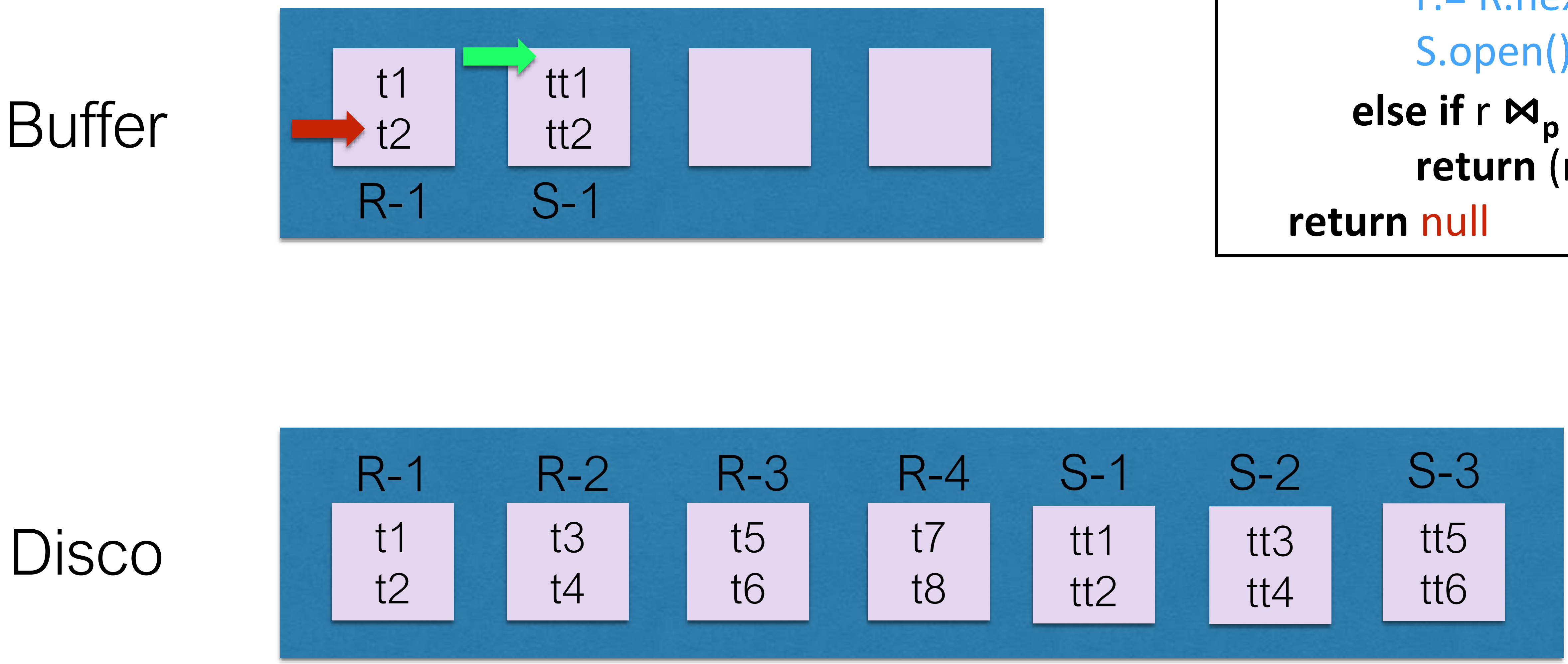


Nested Loop Join

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
```

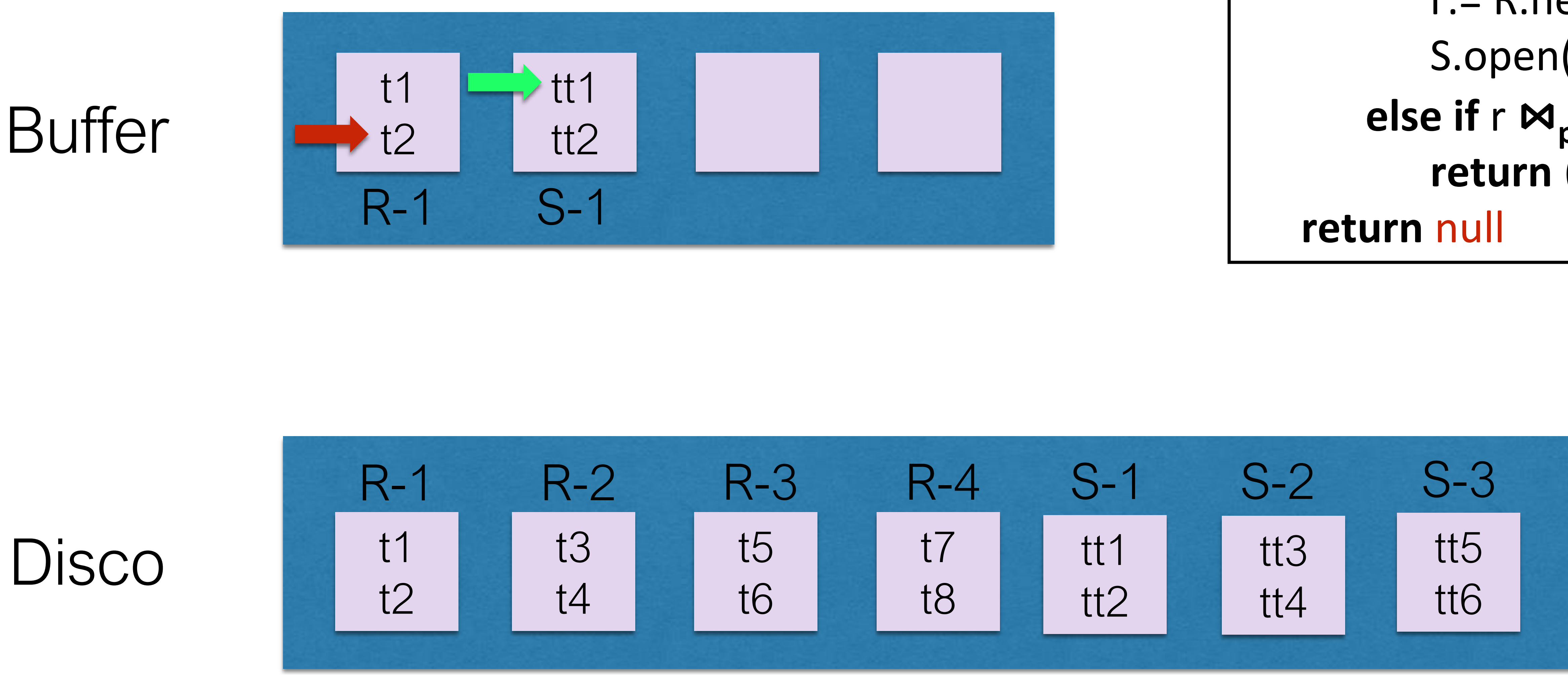


Nested Loop Join

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
```



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

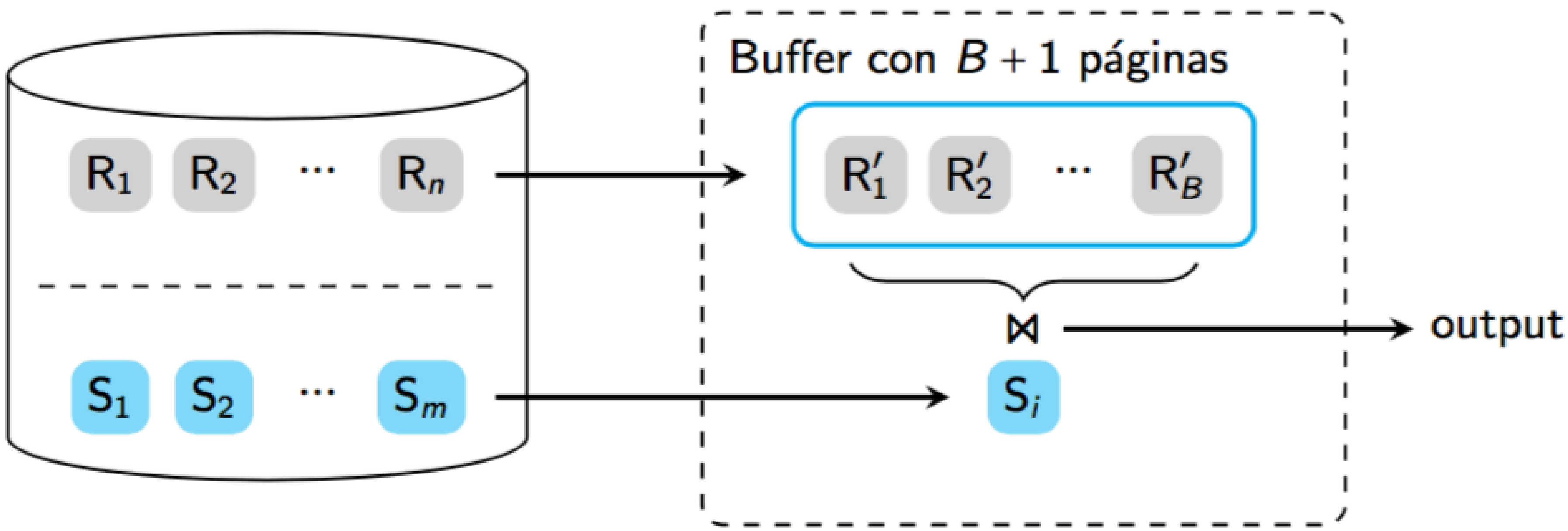
Cada relación tiene 2048 páginas y 55.000 tuplas aproximadamente

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

3.1 horas

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**

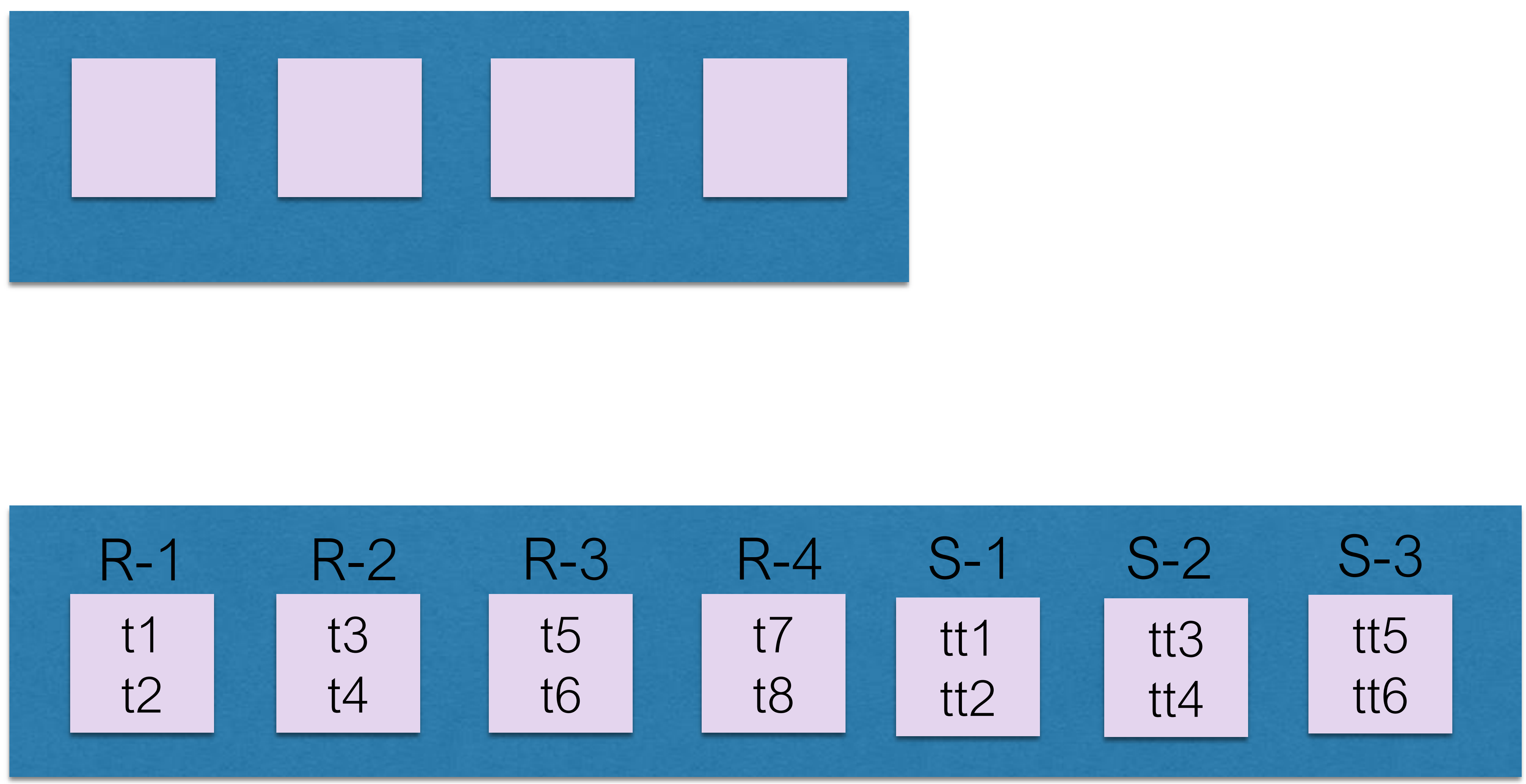
```
next()
    while Buff != empty do
        while s != null do
            r:= Buffer.next()
            if r == null then
                Buffer.reset()
                s:= S.next()
            else if (r,s) satisfacen p then
                return (r,s)
        fillBuffer()
    return null
```

Block Nested Loop Join

DB $R \bowtie_p S$

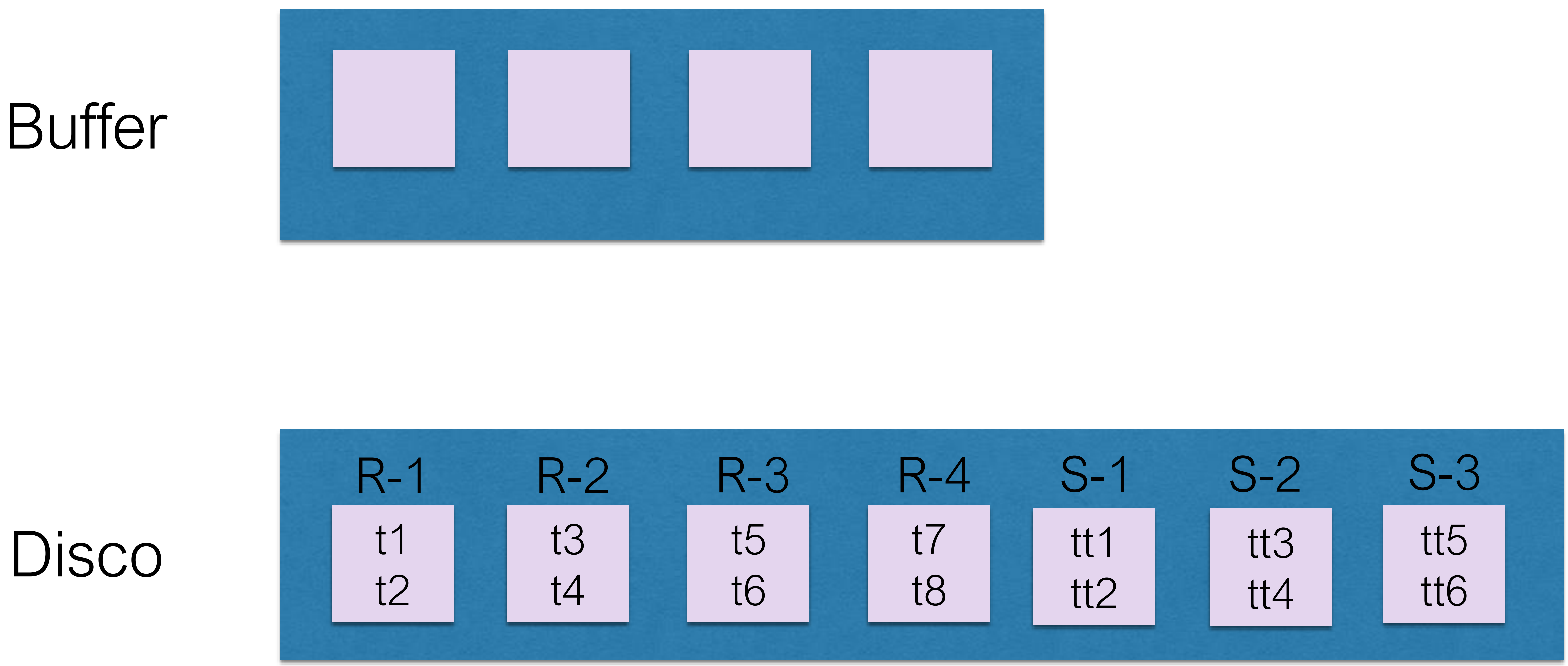
Buffer

Disco



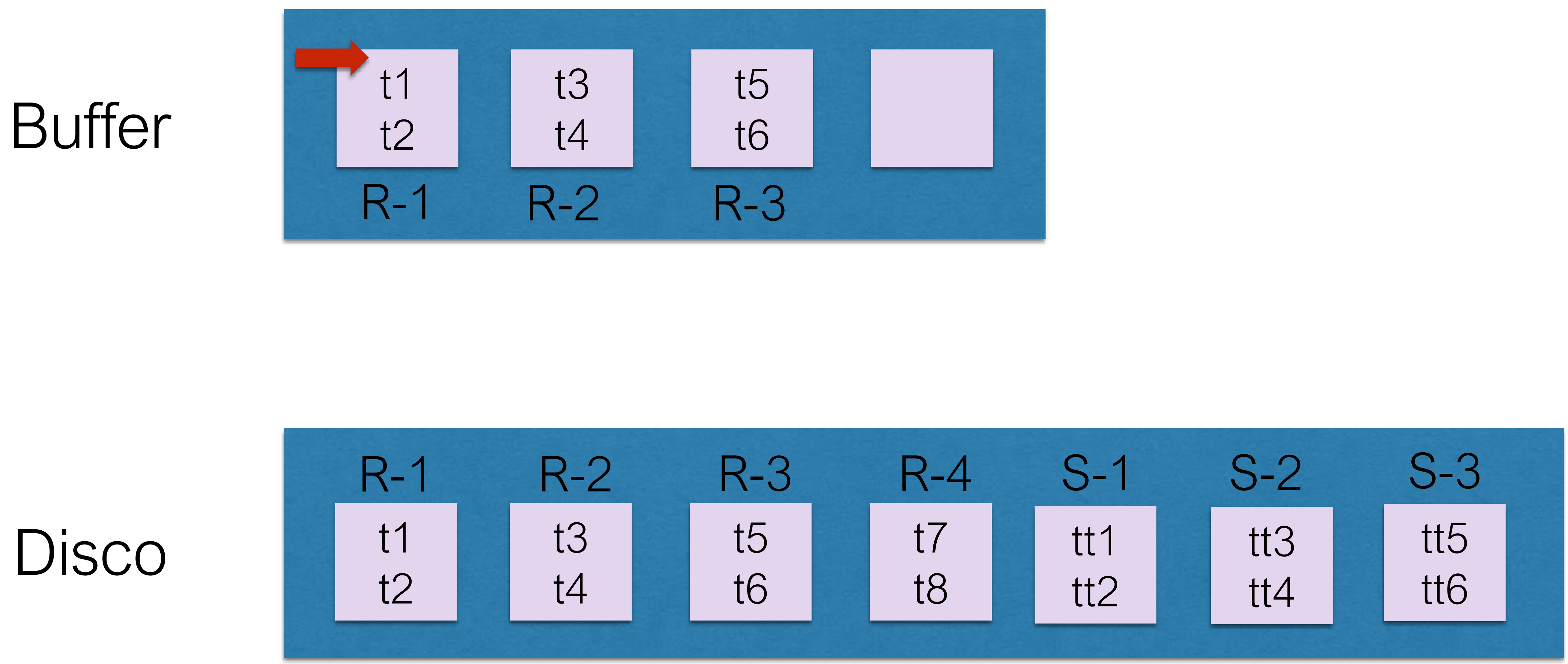
Block Nested Loop Join

DB $R \bowtie_p S$.open()



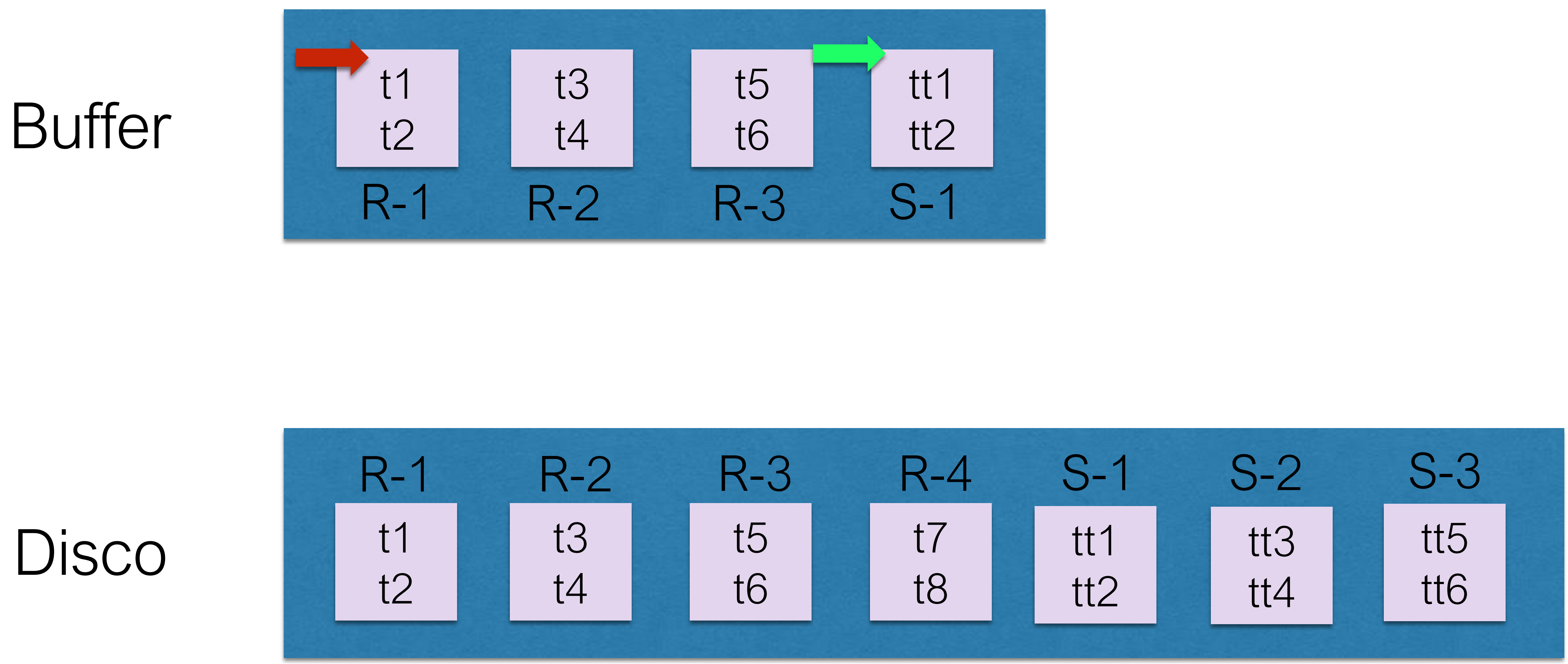
Block Nested Loop Join

DB $R \bowtie_p S$.open()



Block Nested Loop Join

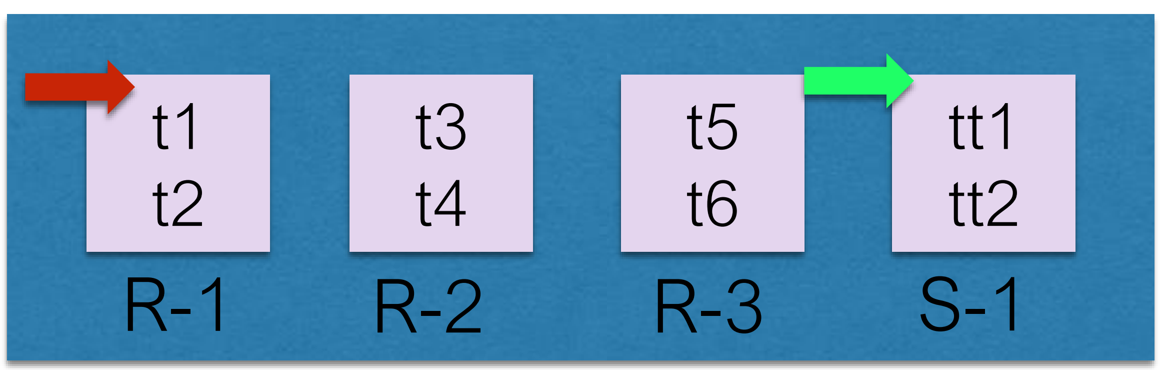
DB $R \bowtie_p S.open()$



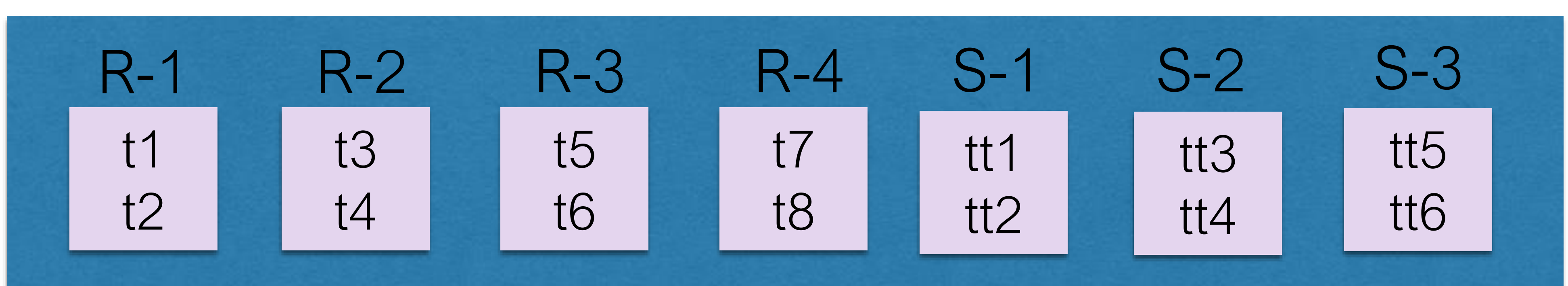
Block Nested Loop Join

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

Buffer



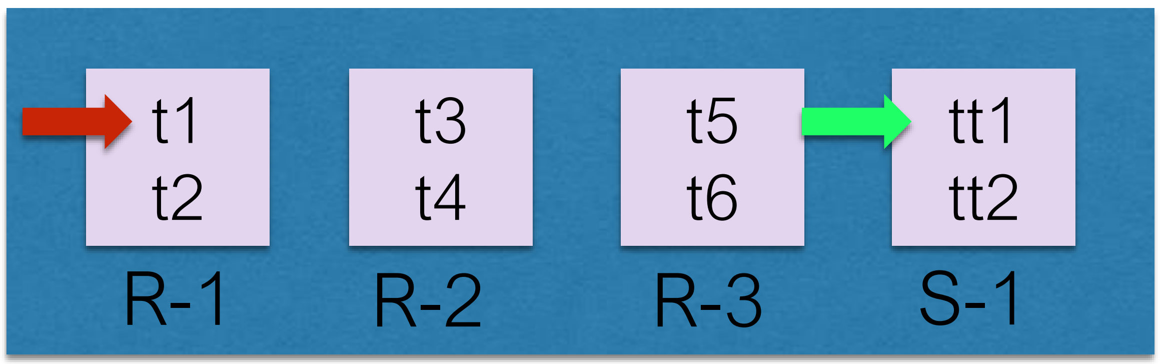
Disco



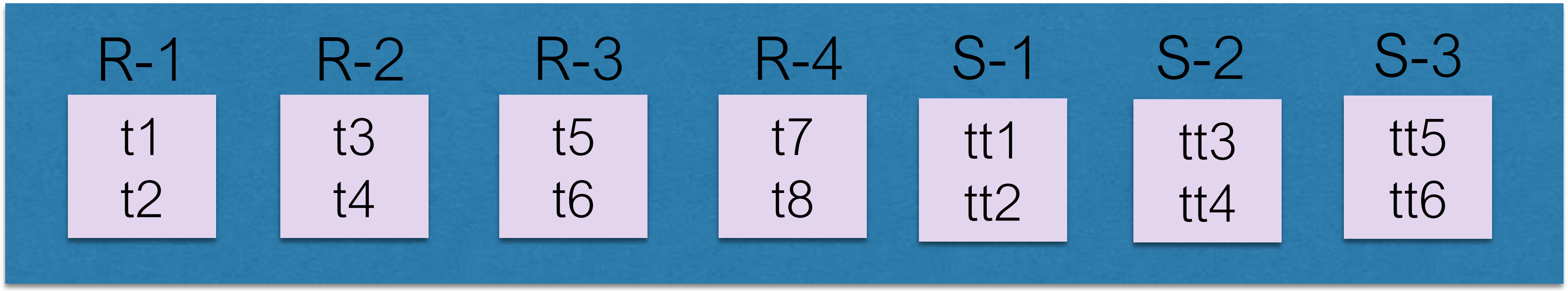
Block Nested Loop Join

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

Buffer



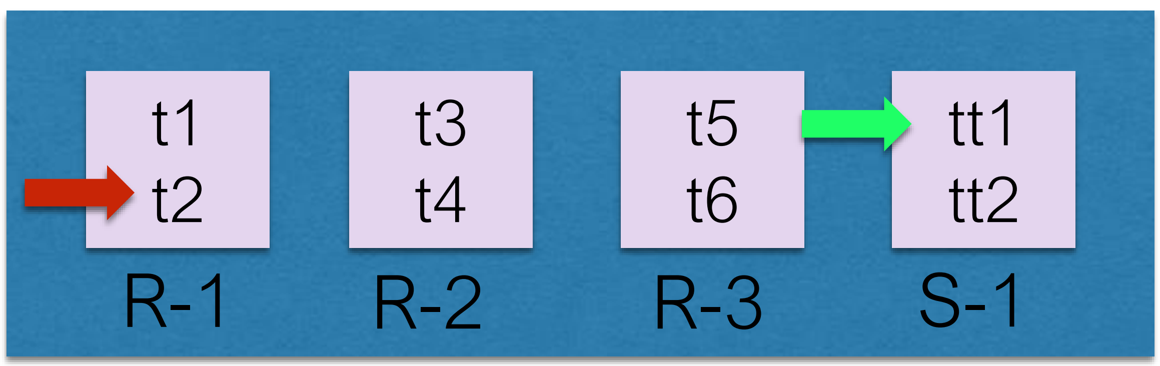
Disco



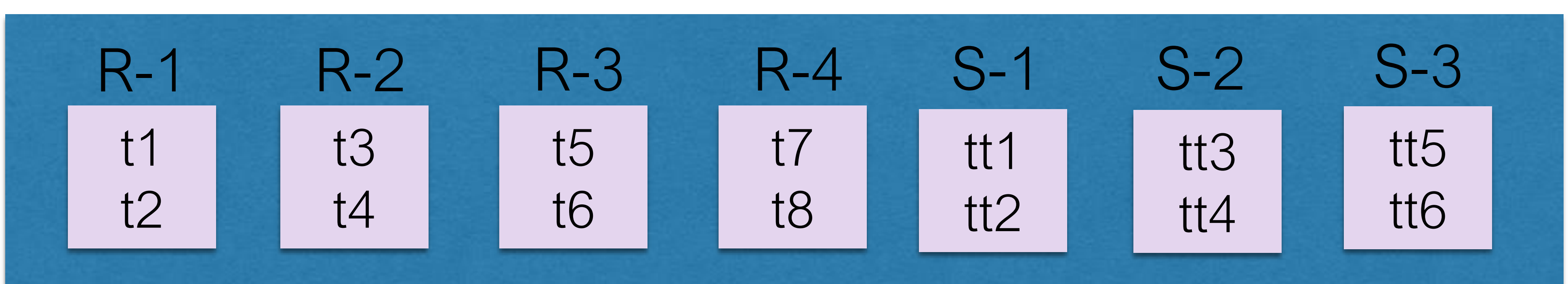
Block Nested Loop Join

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

Buffer

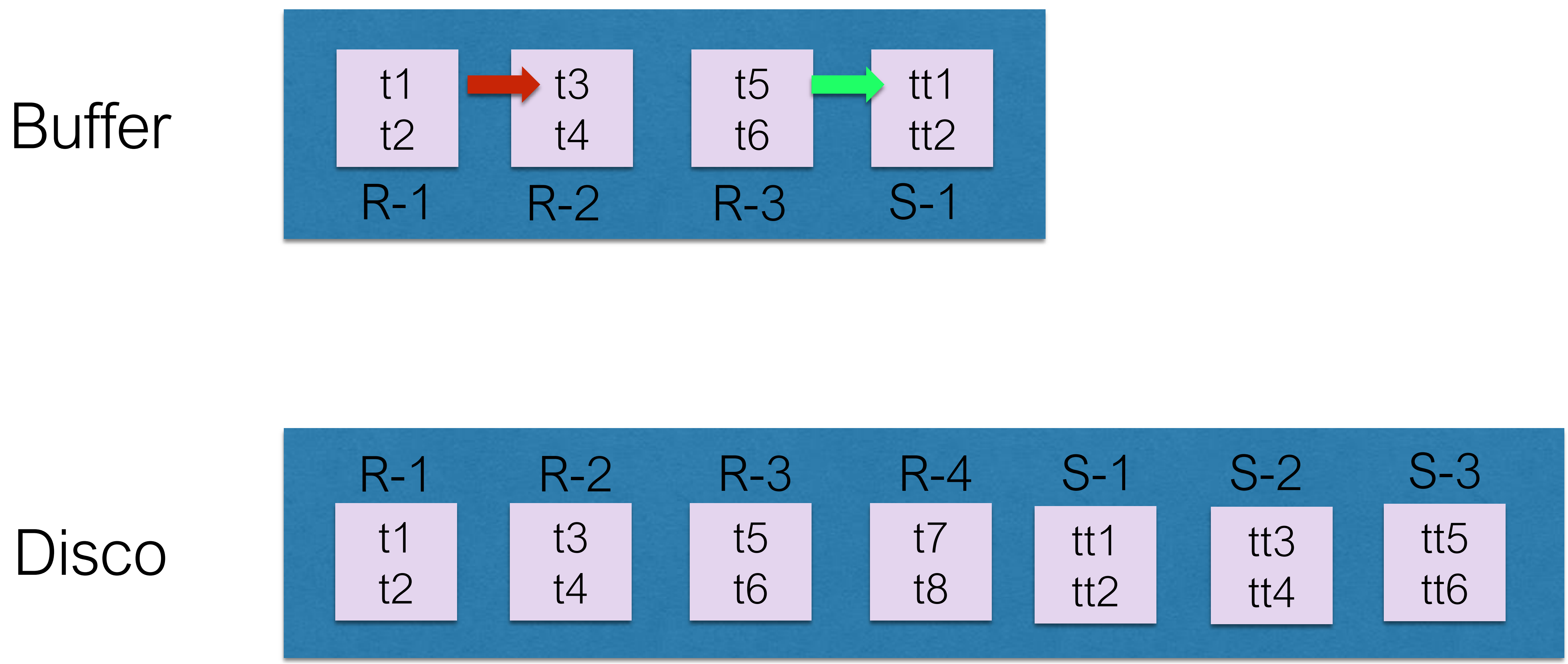


Disco



Block Nested Loop Join

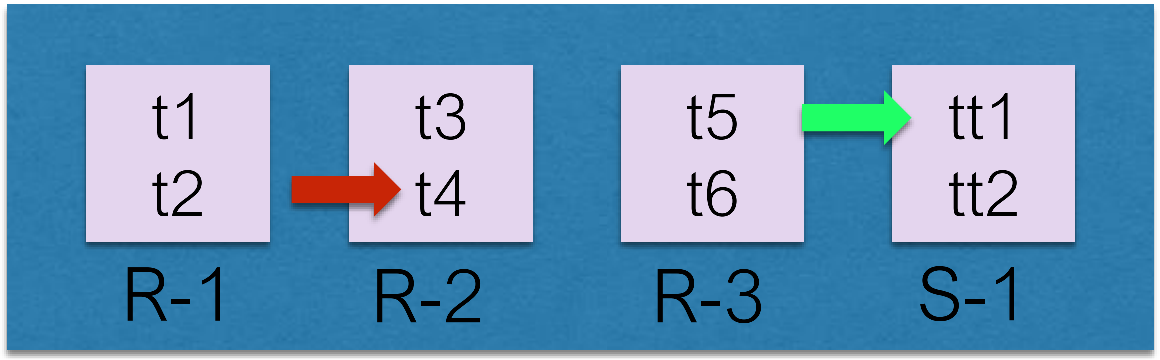
DB while ($R \bowtie_p S$.next())



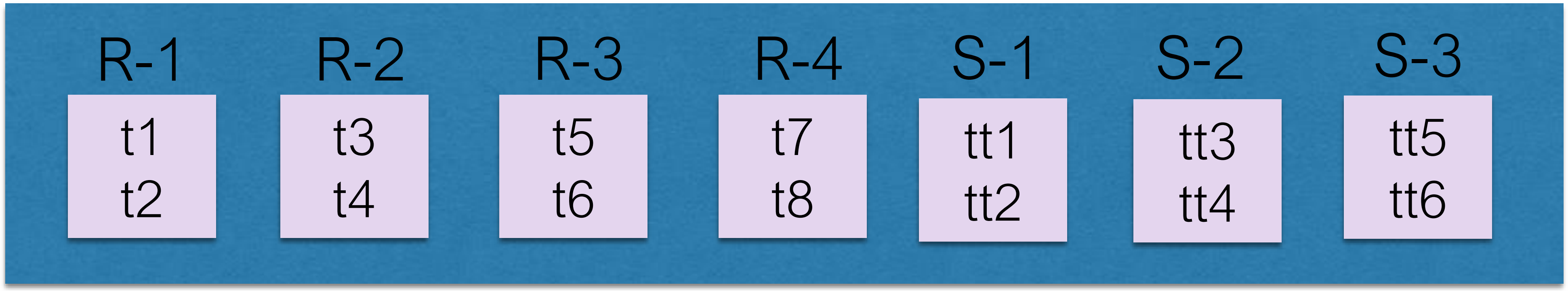
Block Nested Loop Join

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

Buffer

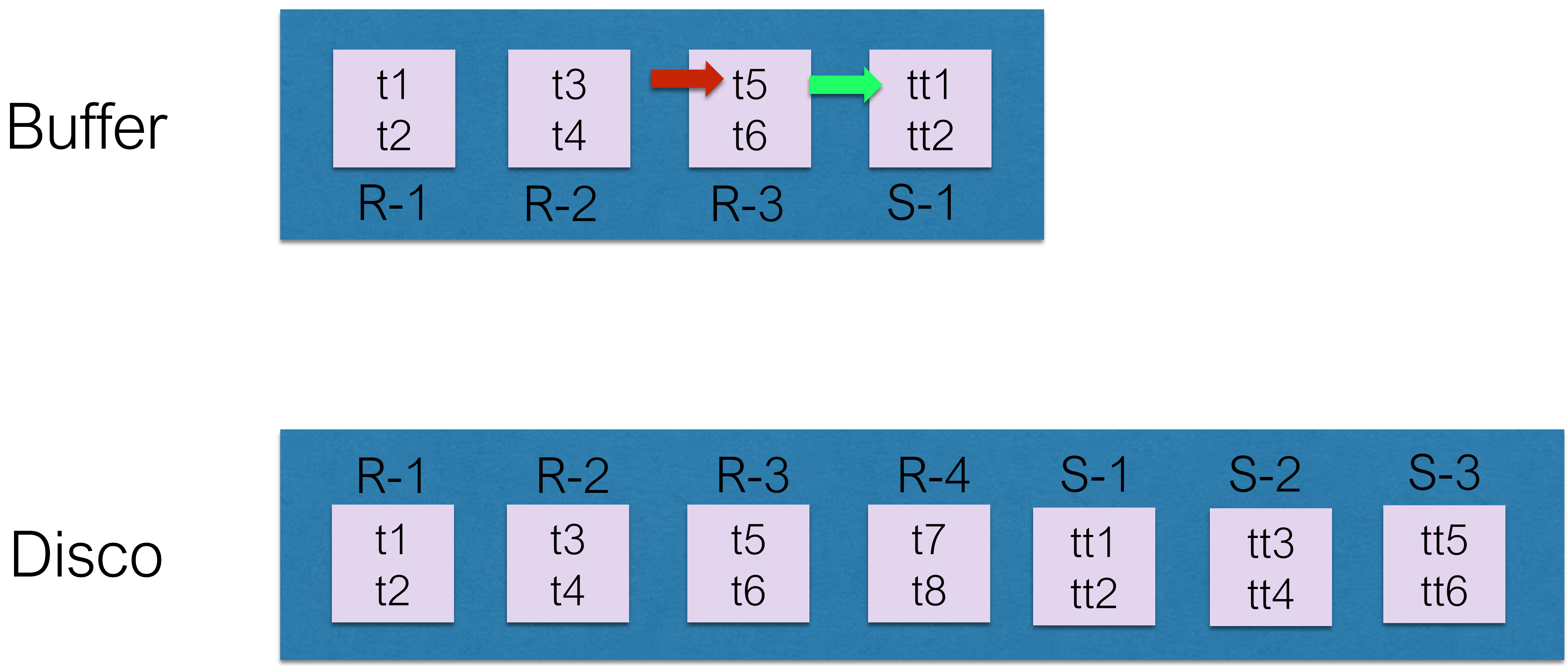


Disco



Block Nested Loop Join

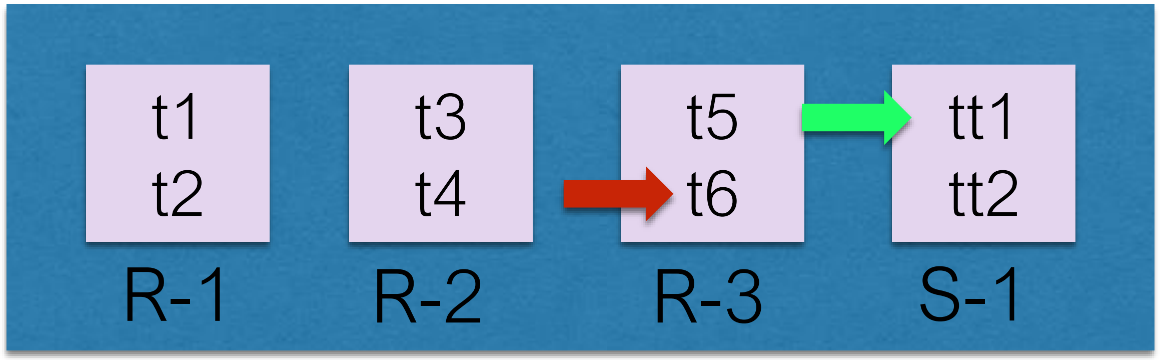
DB while ($R \bowtie_p S$.next())



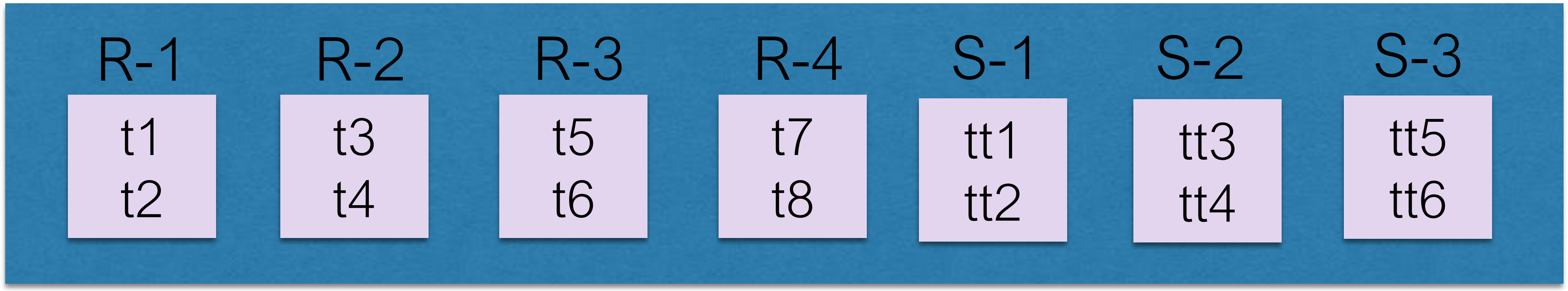
Block Nested Loop Join

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

Buffer

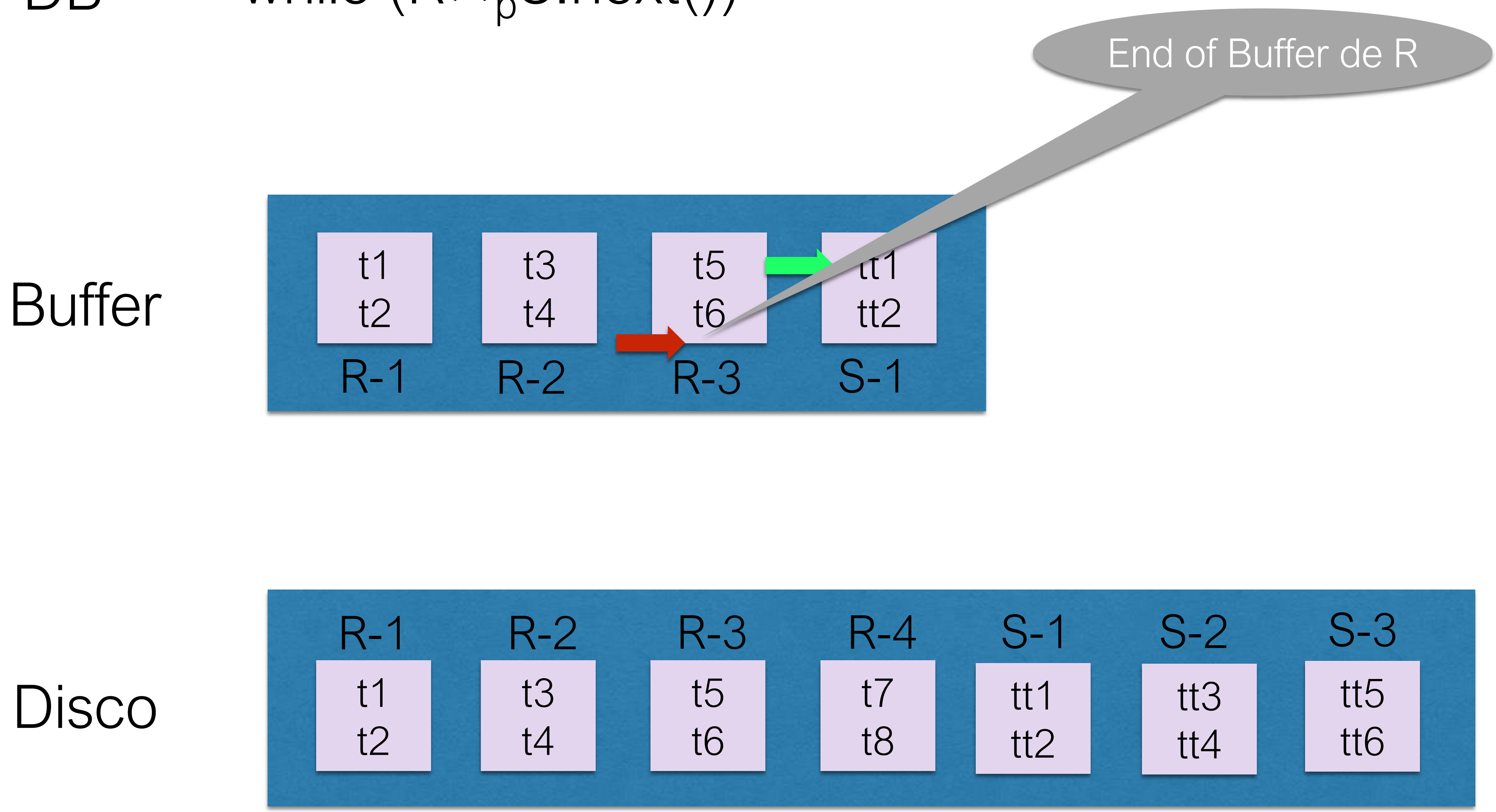


Disco



Block Nested Loop Join

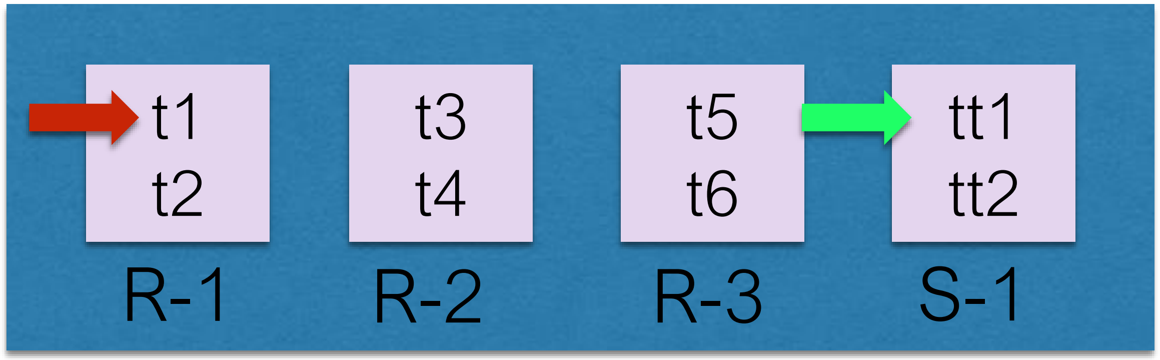
DB while $(R \bowtie_p S.next())$



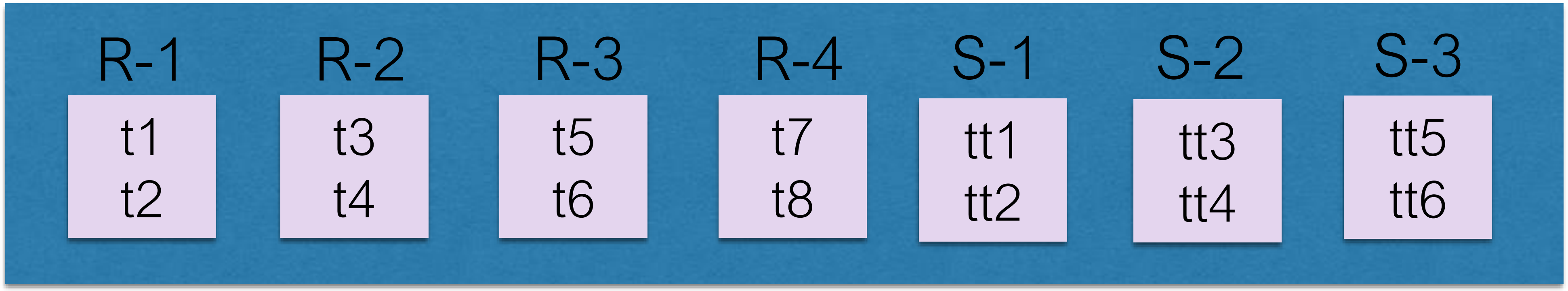
Block Nested Loop Join

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

Buffer



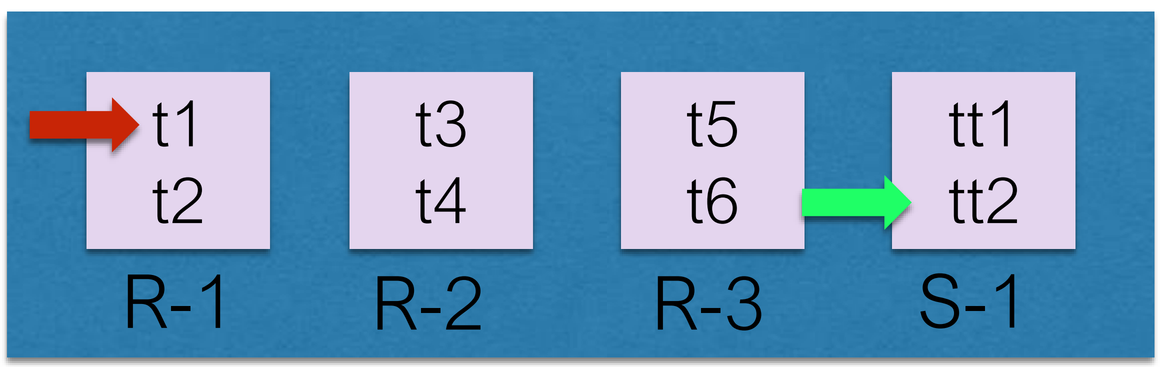
Disco



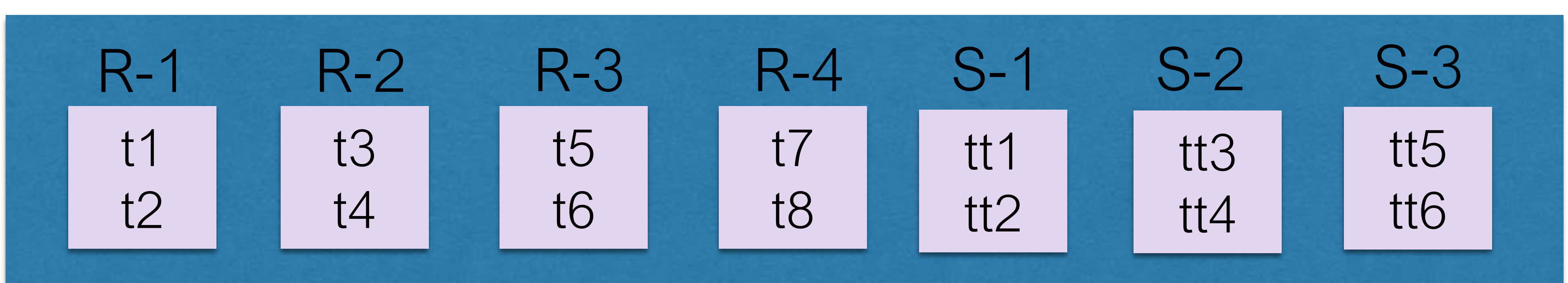
Block Nested Loop Join

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

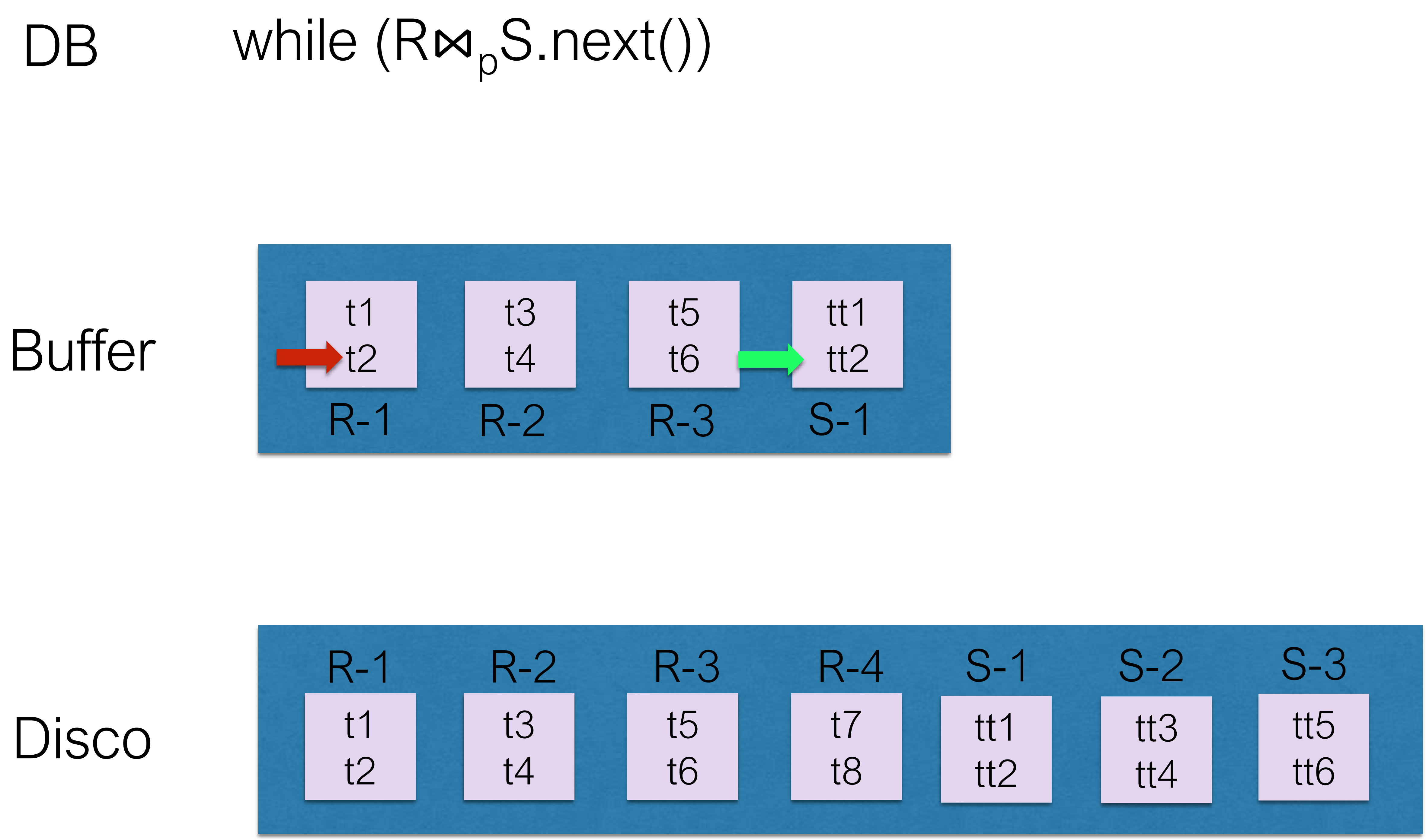
Buffer



Disco



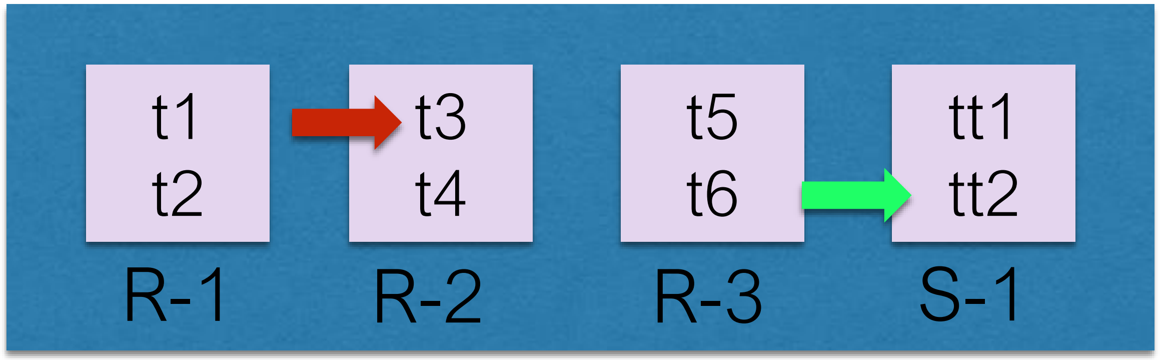
Block Nested Loop Join



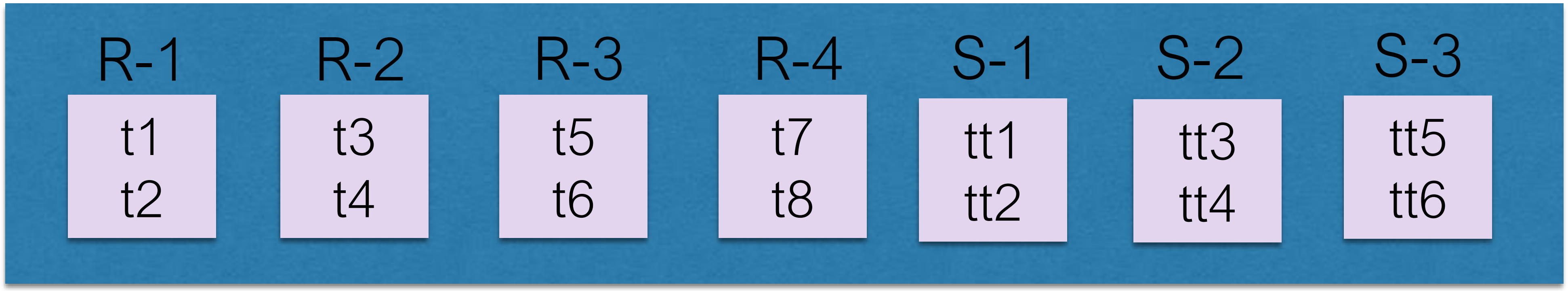
Block Nested Loop Join

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

Buffer



Disco

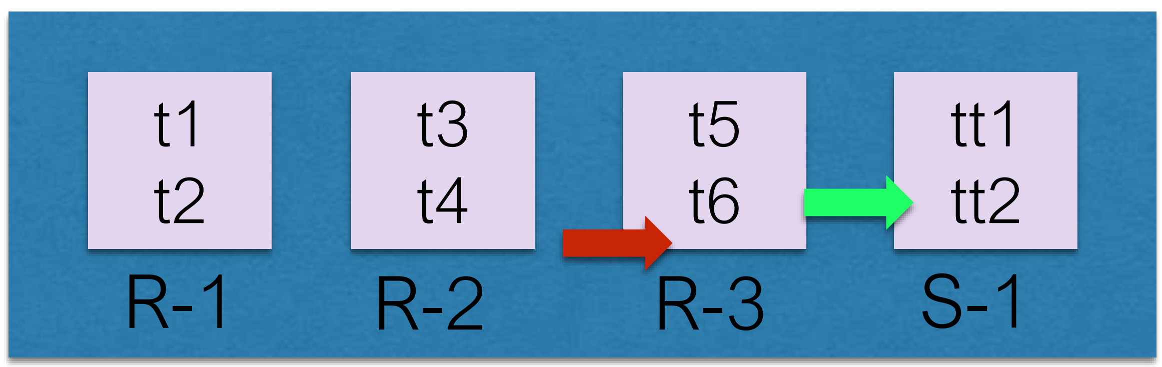


Block Nested Loop Join

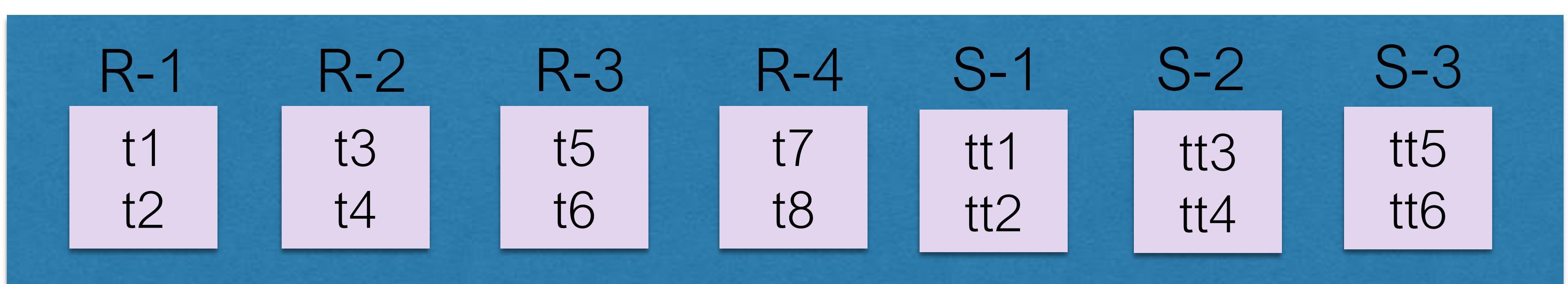
DB

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

Buffer

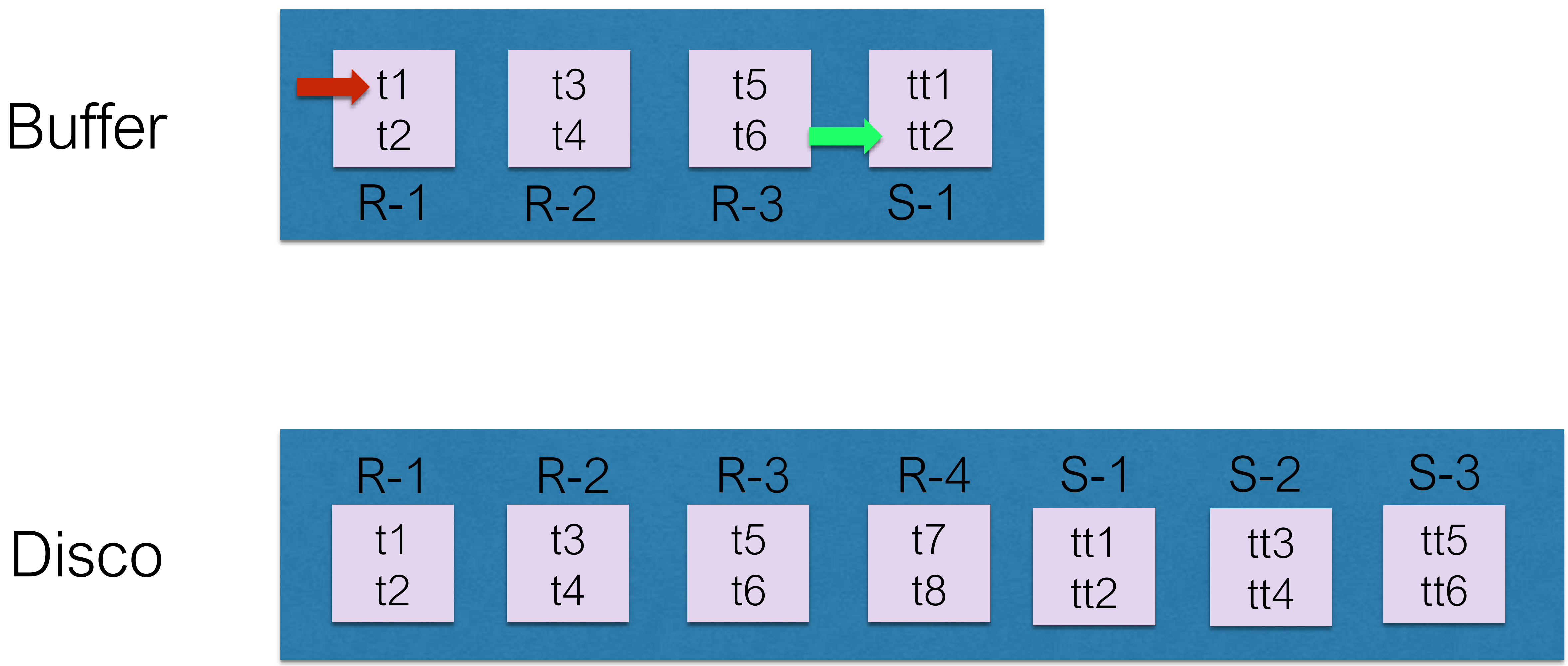


Disco



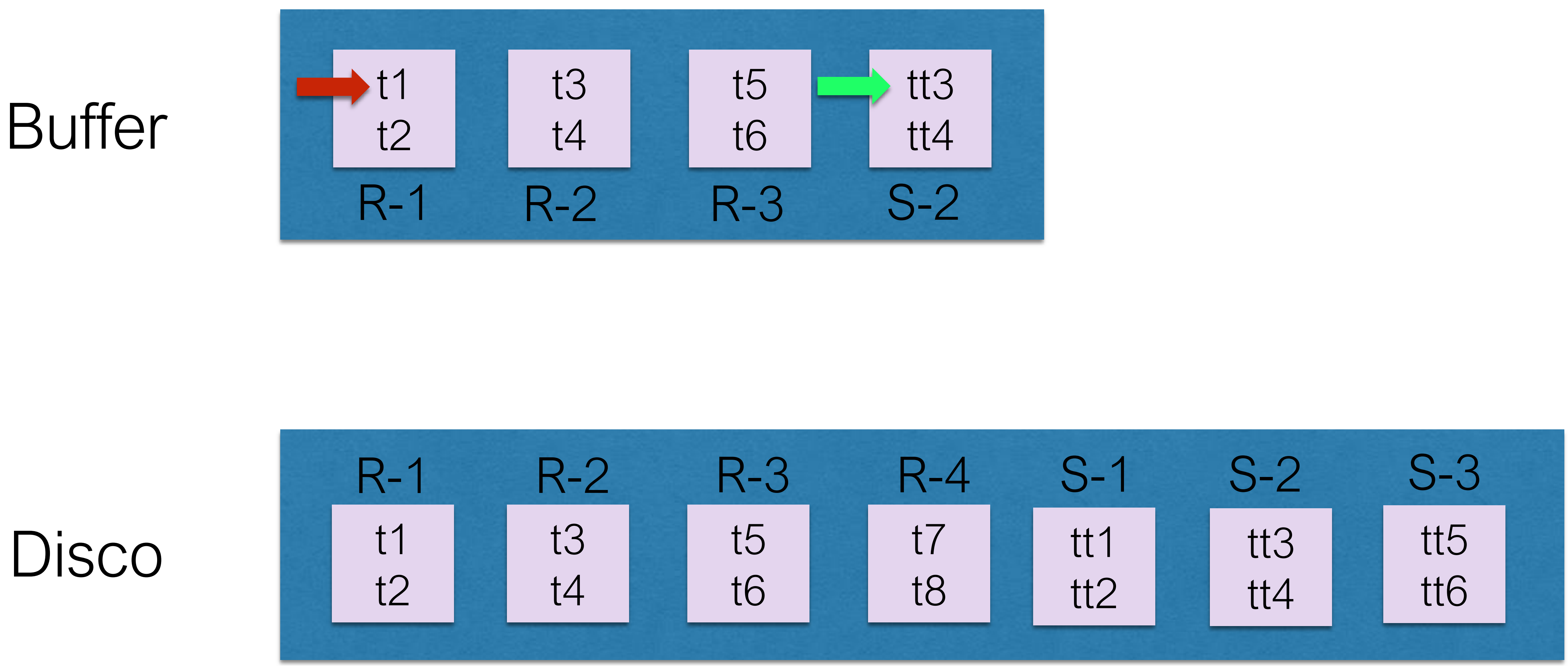
Block Nested Loop Join

DB while (R⋈$_p$S.next())



Block Nested Loop Join

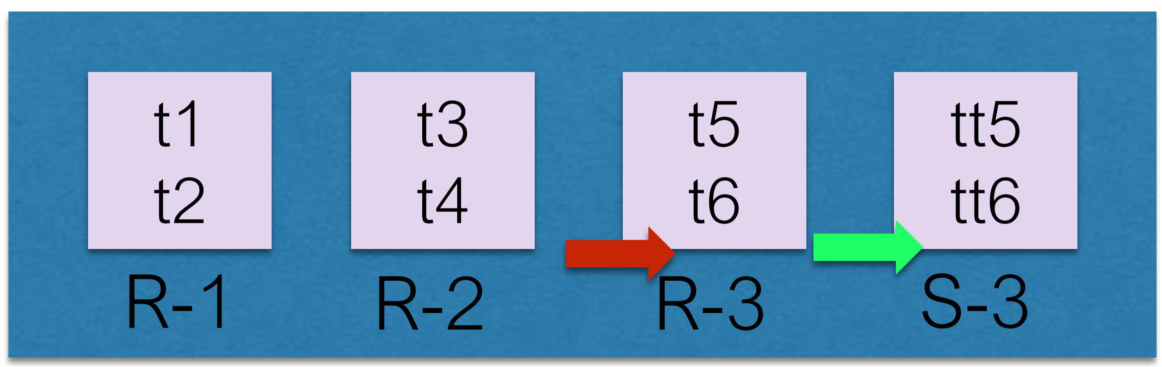
DB $\quad$ while ($R \bowtie_p S$.next())



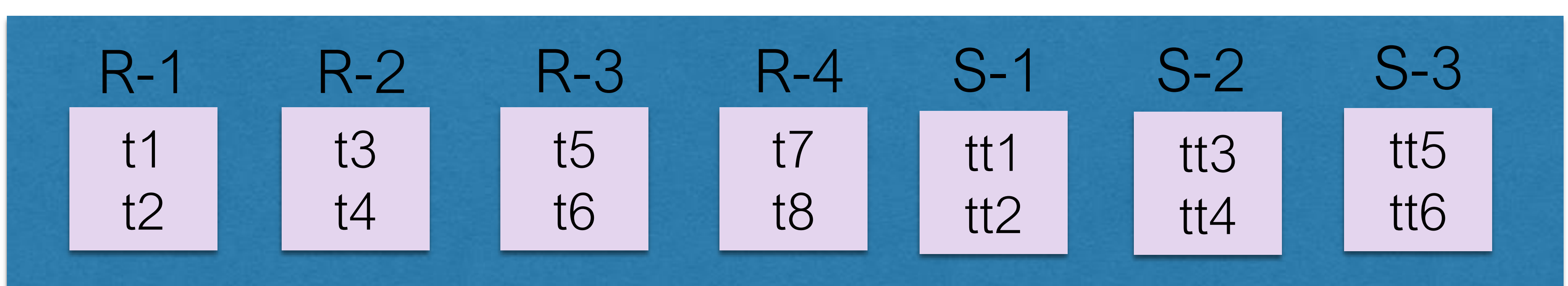
Block Nested Loop Join

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

Buffer



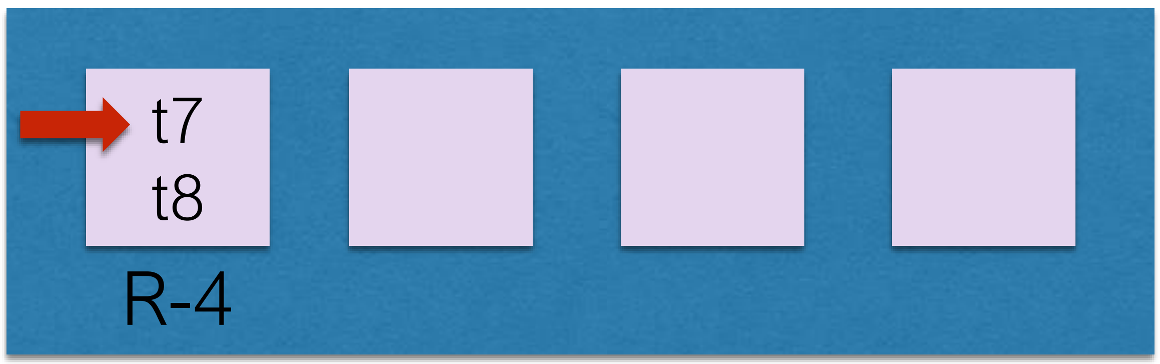
Disco



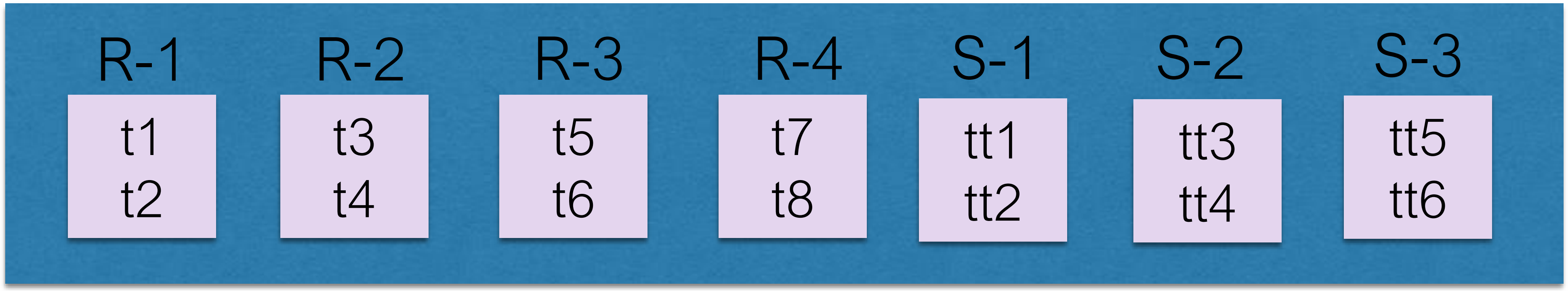
Block Nested Loop Join

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

Buffer



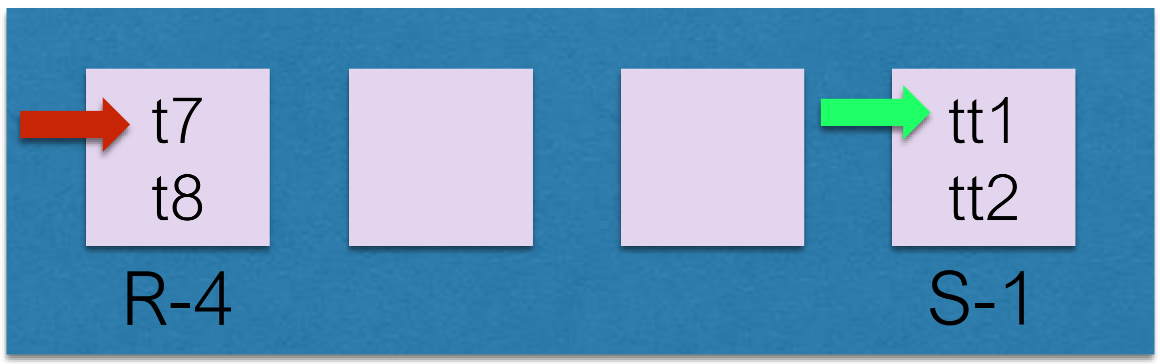
Disco



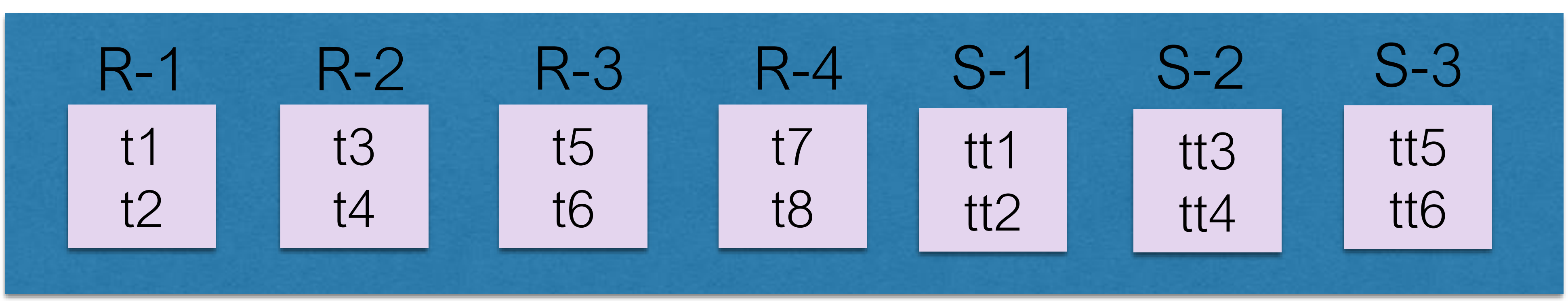
Block Nested Loop Join

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

Buffer



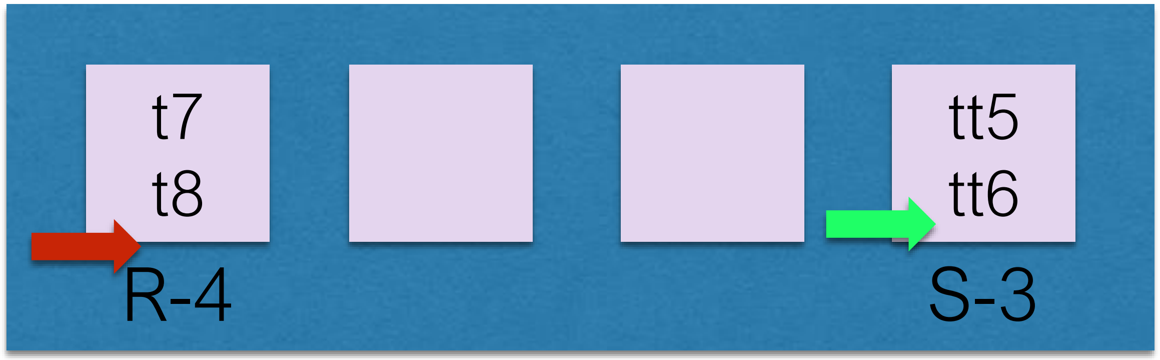
Disco



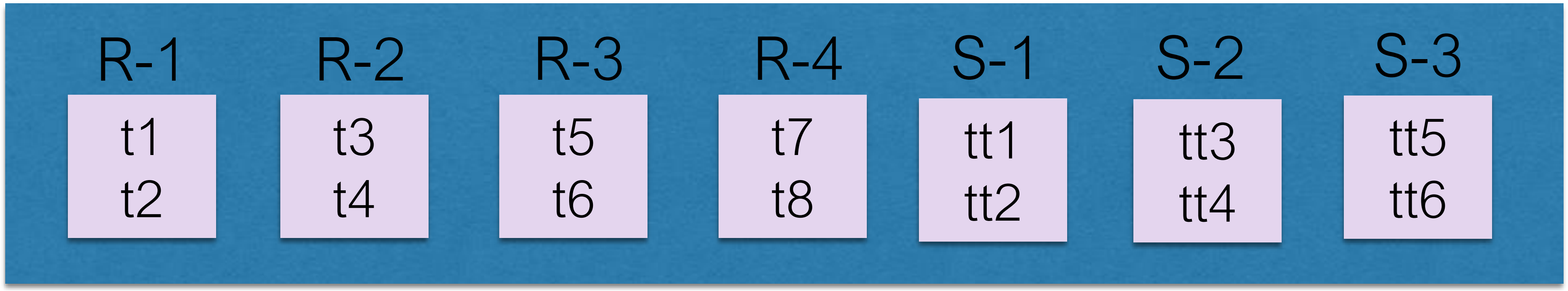
Block Nested Loop Join

DB while (R⋈$_p$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:

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

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

¿Por qué estudiarlos otra vez?

Sorting

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

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**

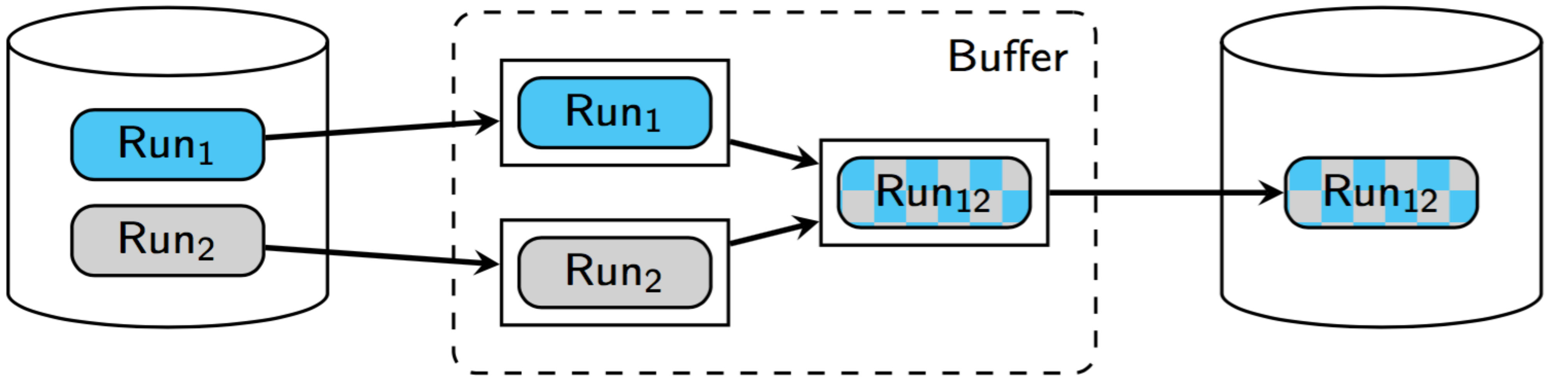
External Merge Sort

Fase 0: creamos los runs iniciales

Fase i:

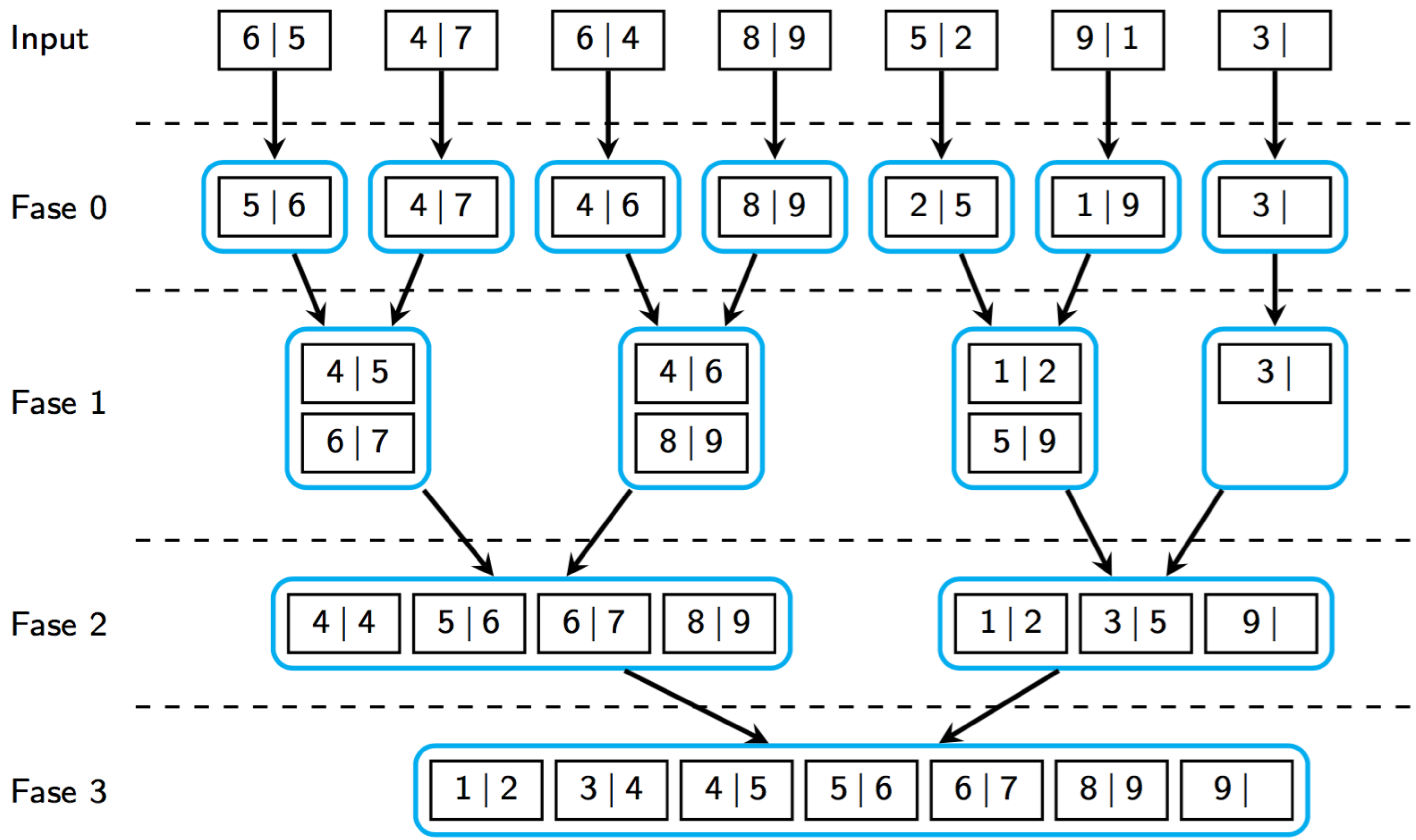
- Traemos los runs a memoria
- Hacemos el merge de cada par de runs
- Almacenamos el nuevo run a disco (i.e. materializamos resultados intermedios)

External Merge Sort



Ojo! cada run se compone de varias páginas, por lo que en cada fase hay un subconjunto de ambos runs en buffer

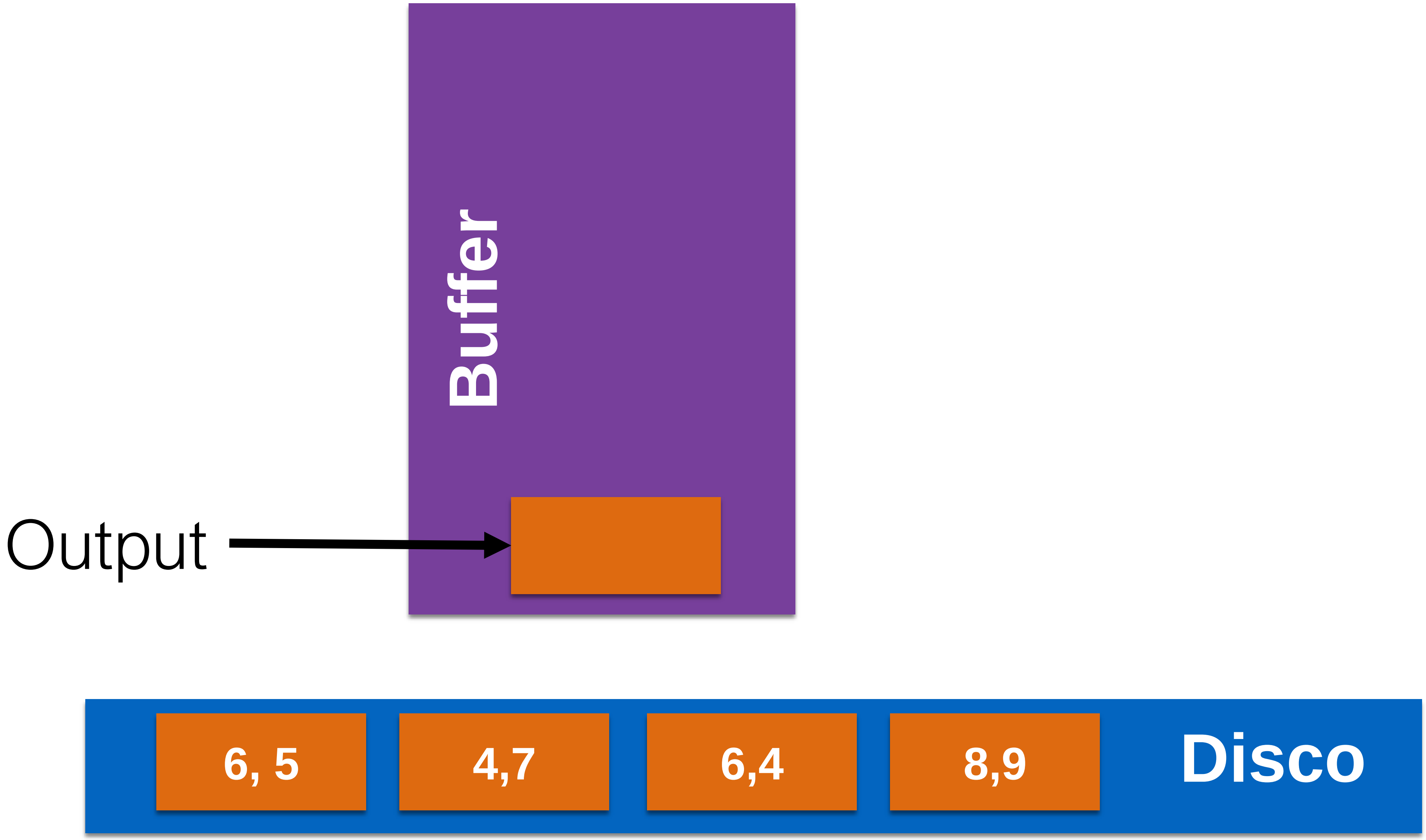
External Merge Sort



External Merge Sort

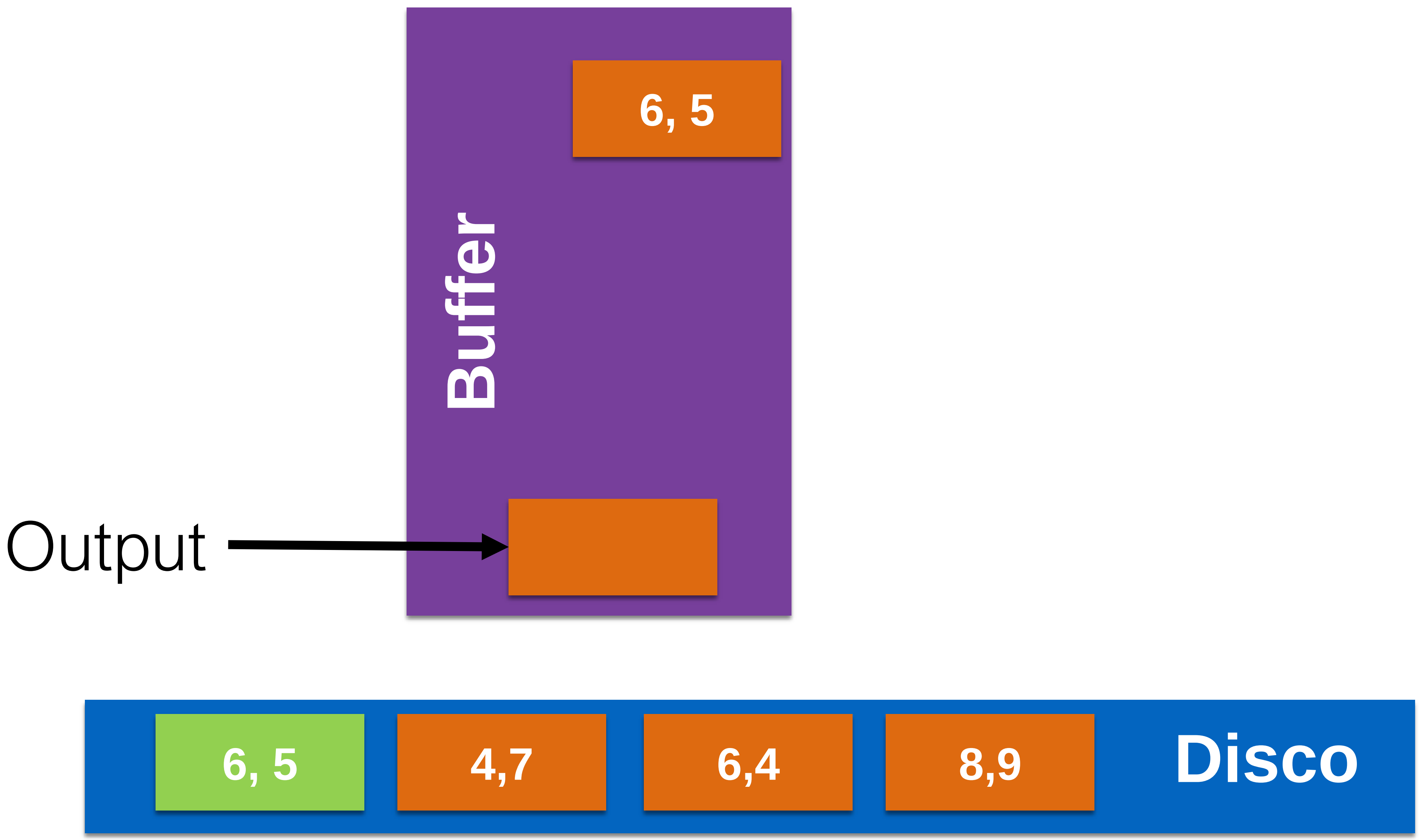
Fase 0

Se van escribiendo las páginas ordenadas:



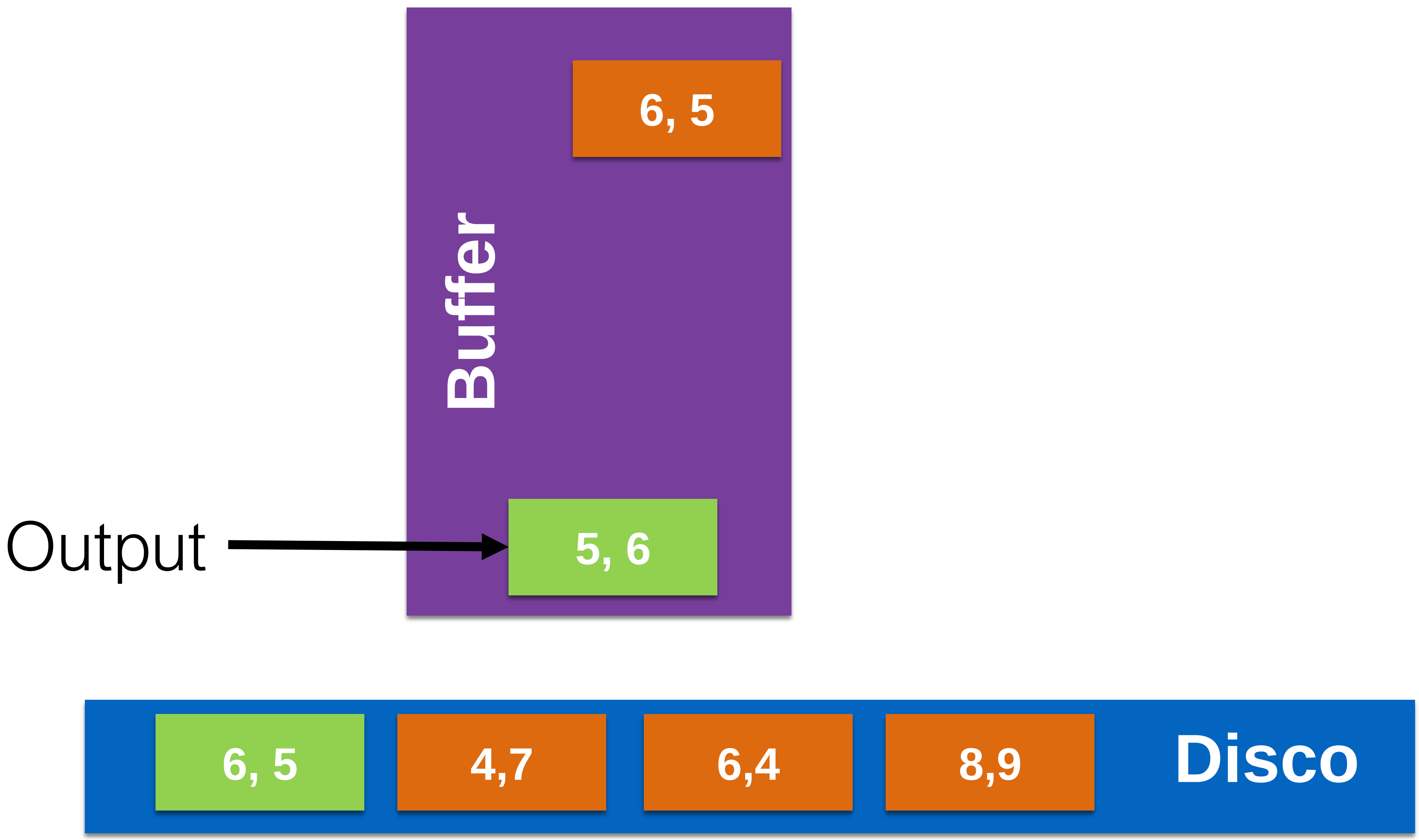
External Merge Sort

Fase 0



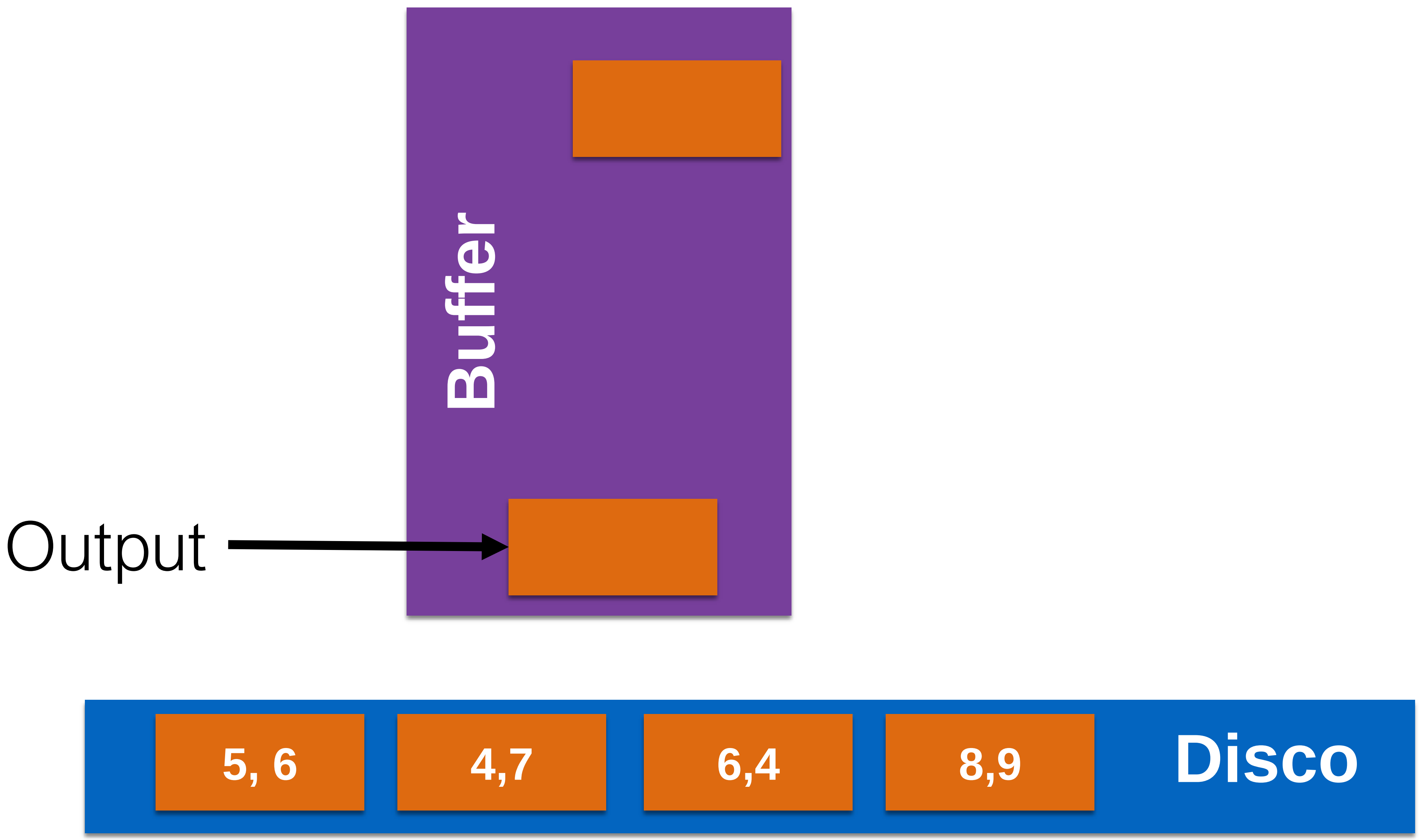
External Merge Sort

Fase 0



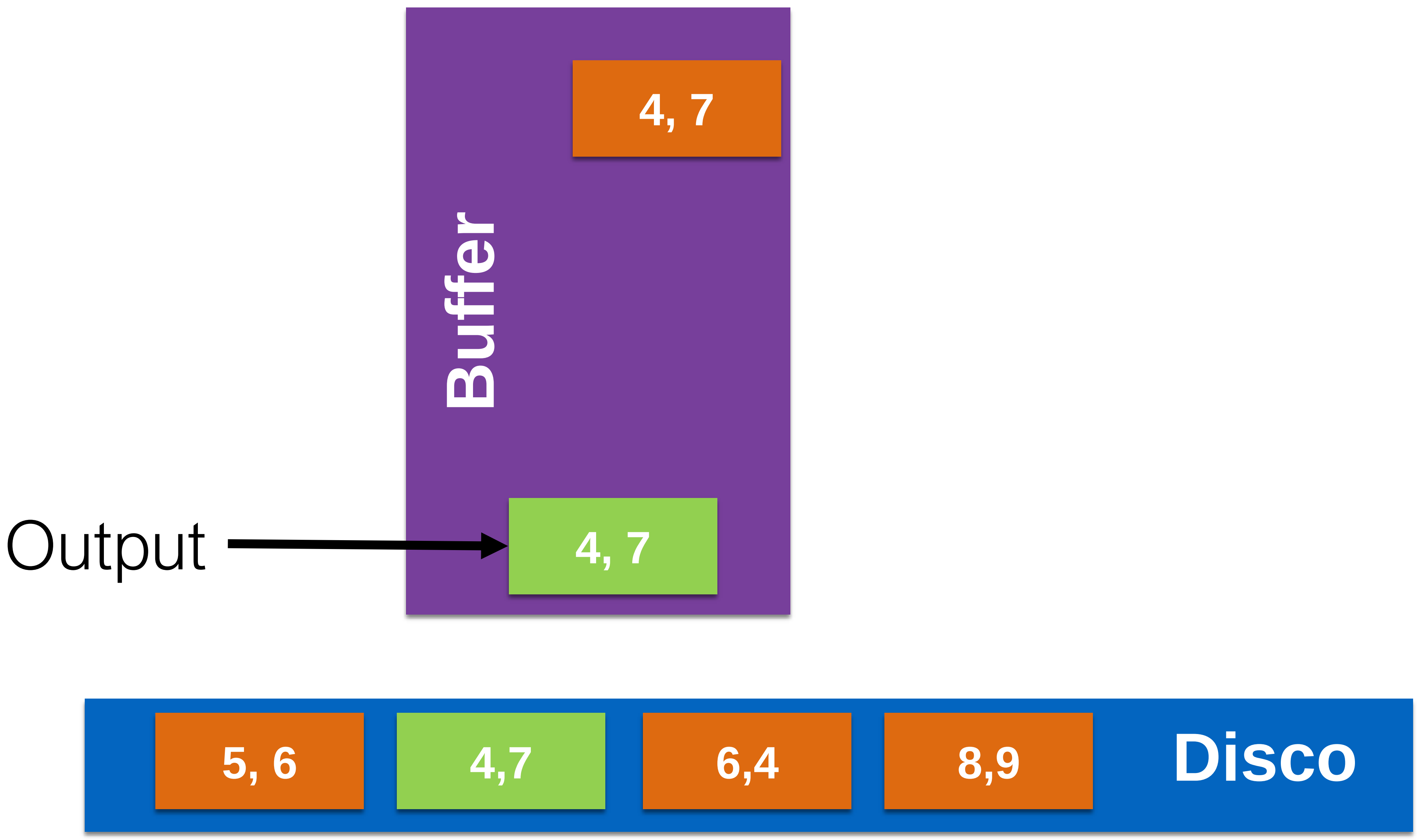
External Merge Sort

Fase 0



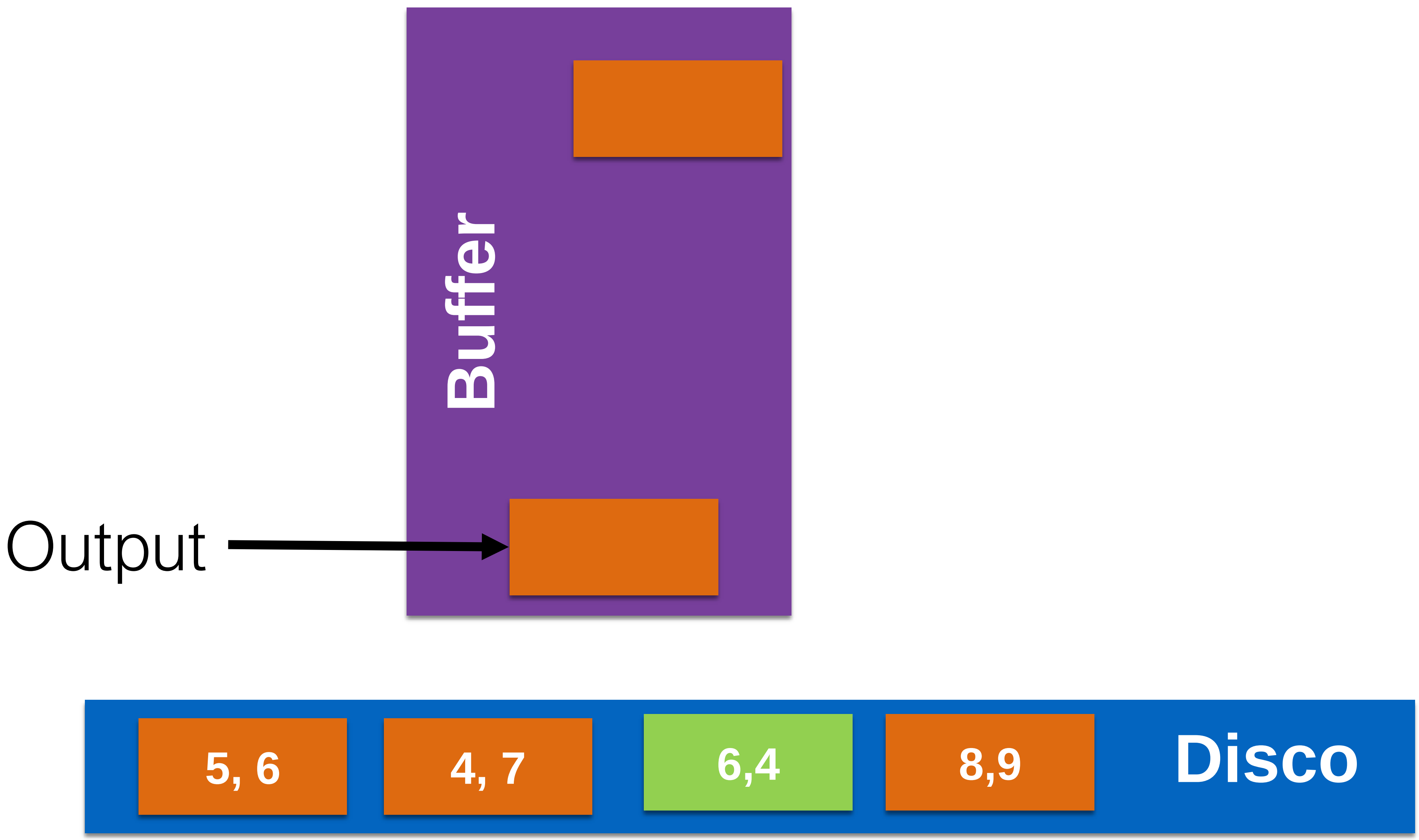
External Merge Sort

Fase 0



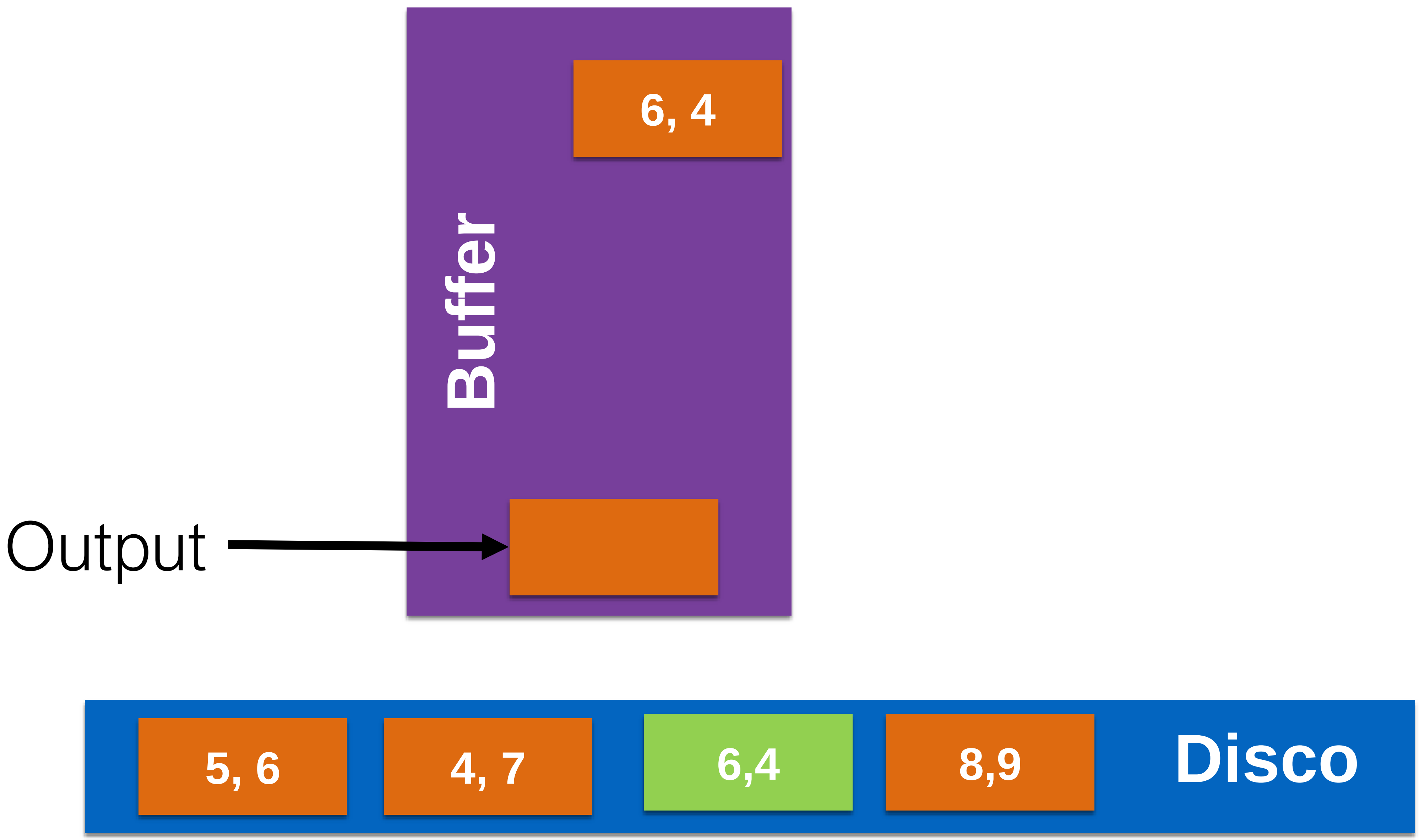
External Merge Sort

Fase 0



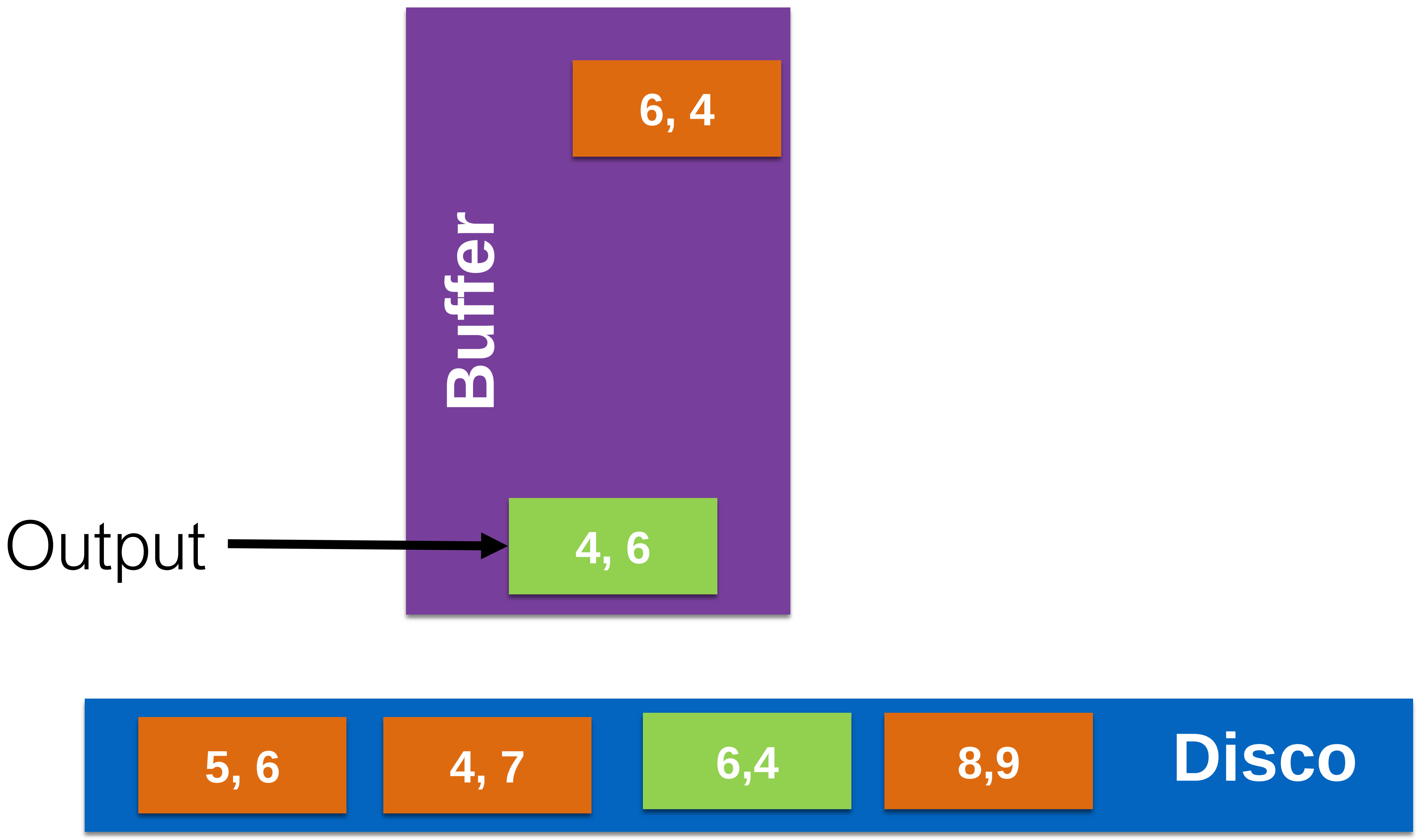
External Merge Sort

Fase 0



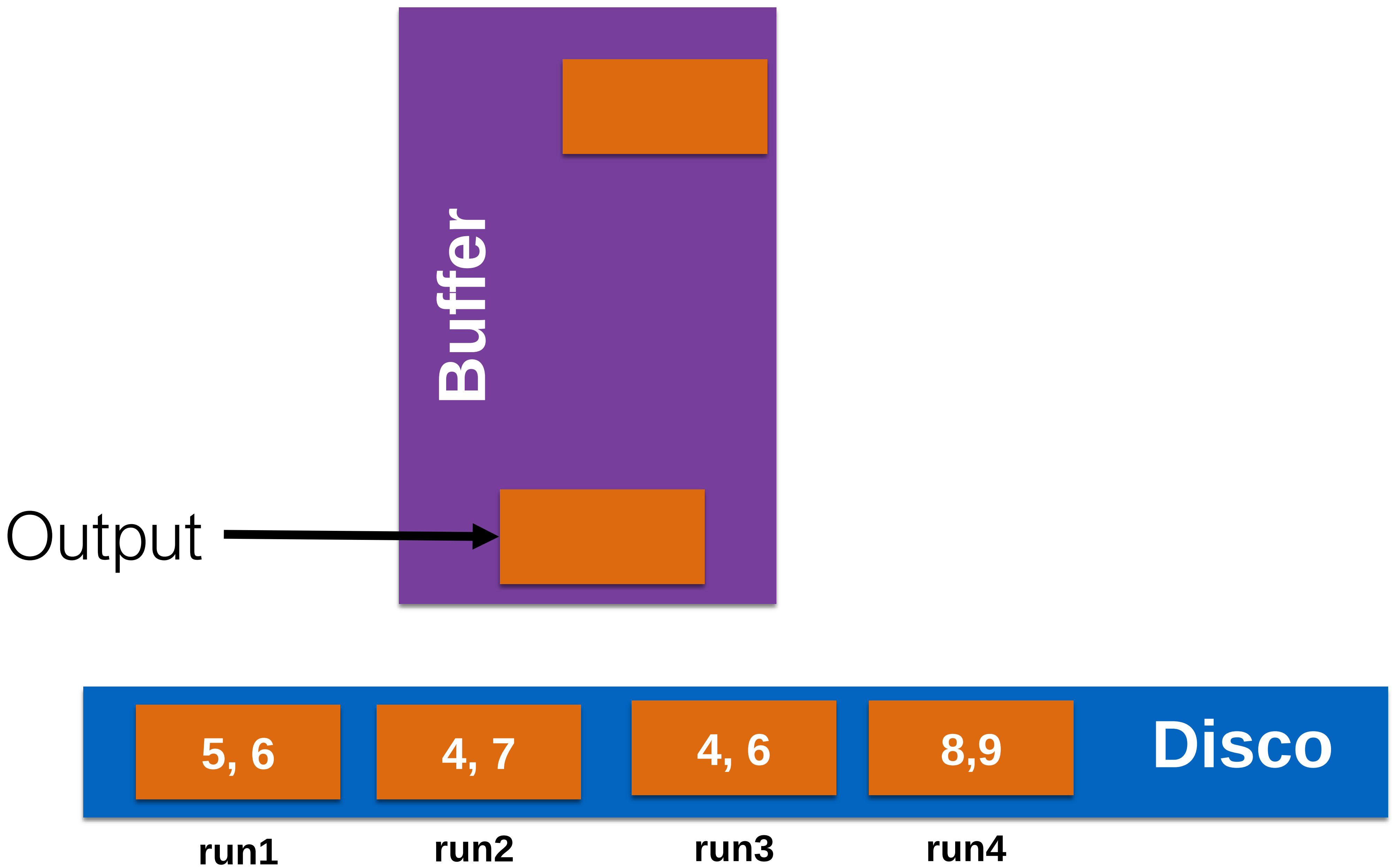
External Merge Sort

Fase 0



External Merge Sort

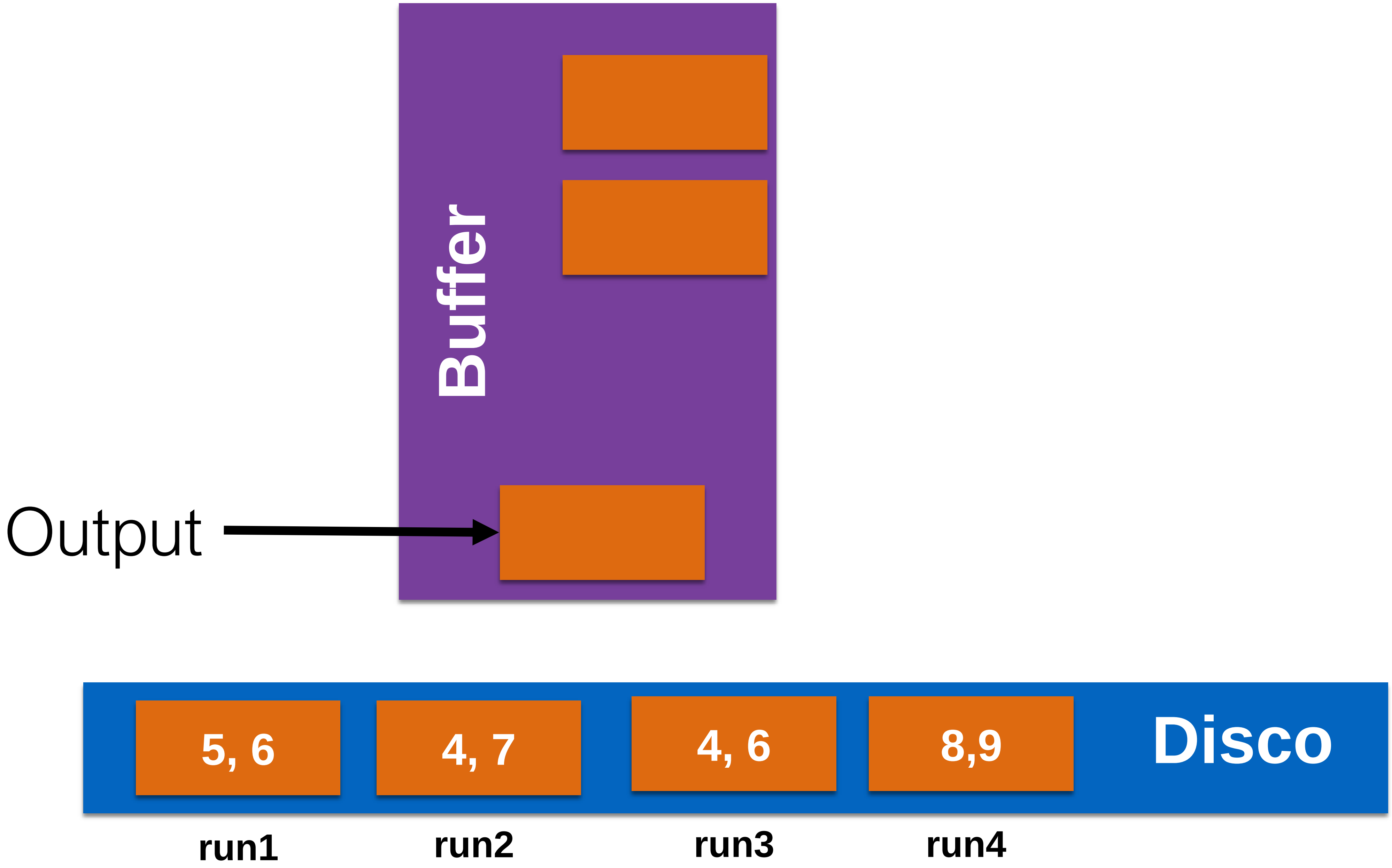
Fase 0



External Merge Sort

Fase 1

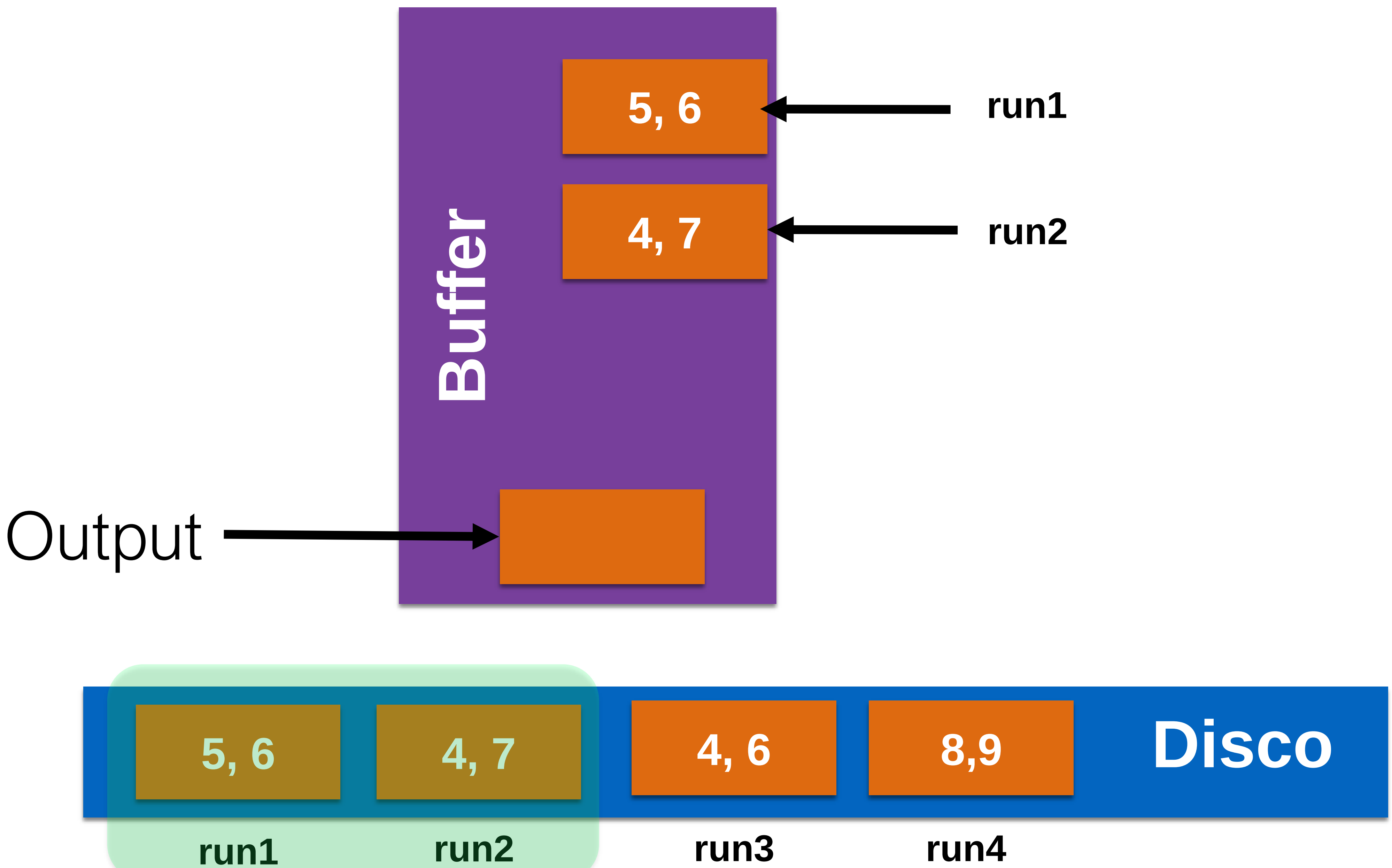
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

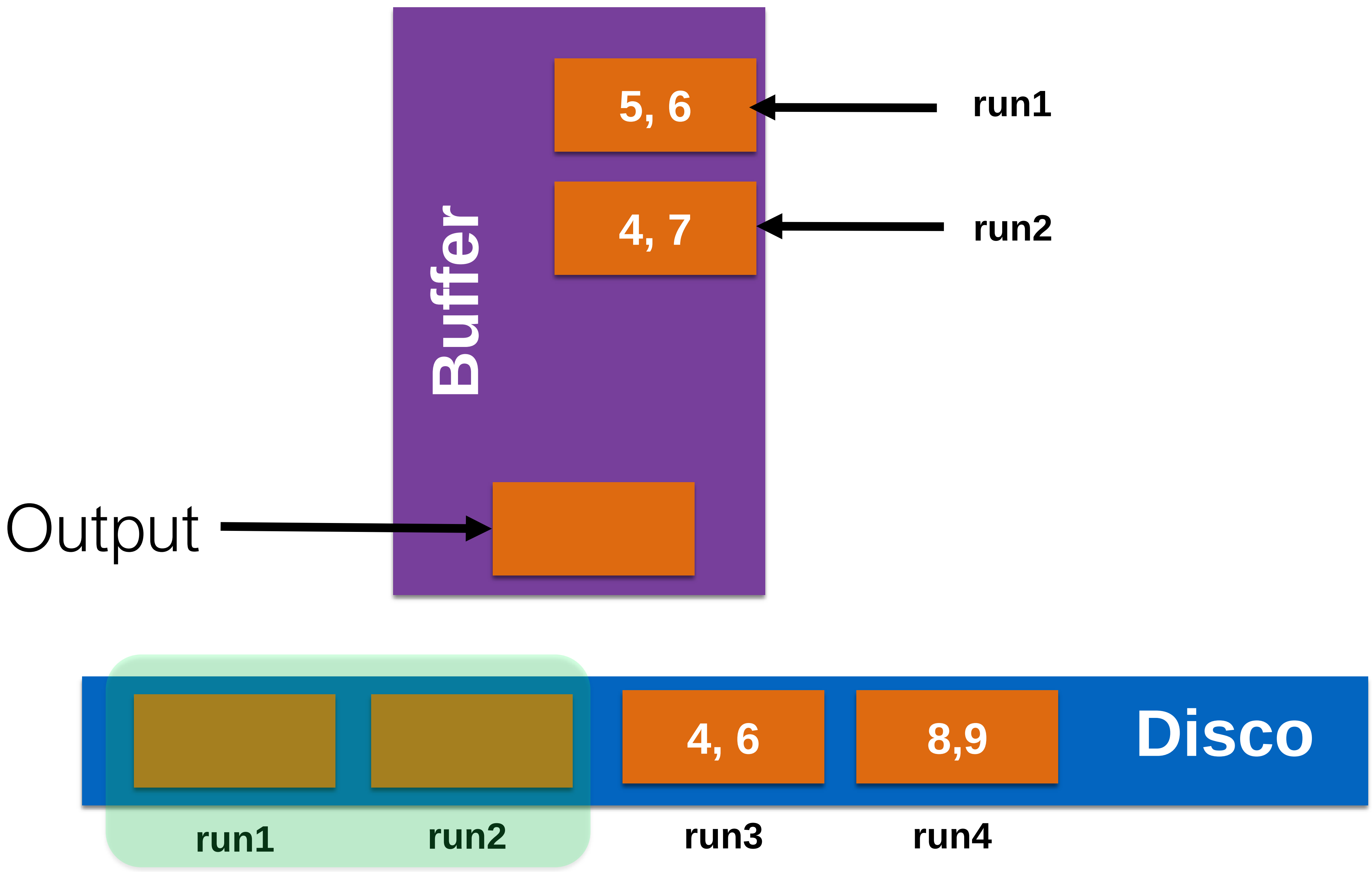
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

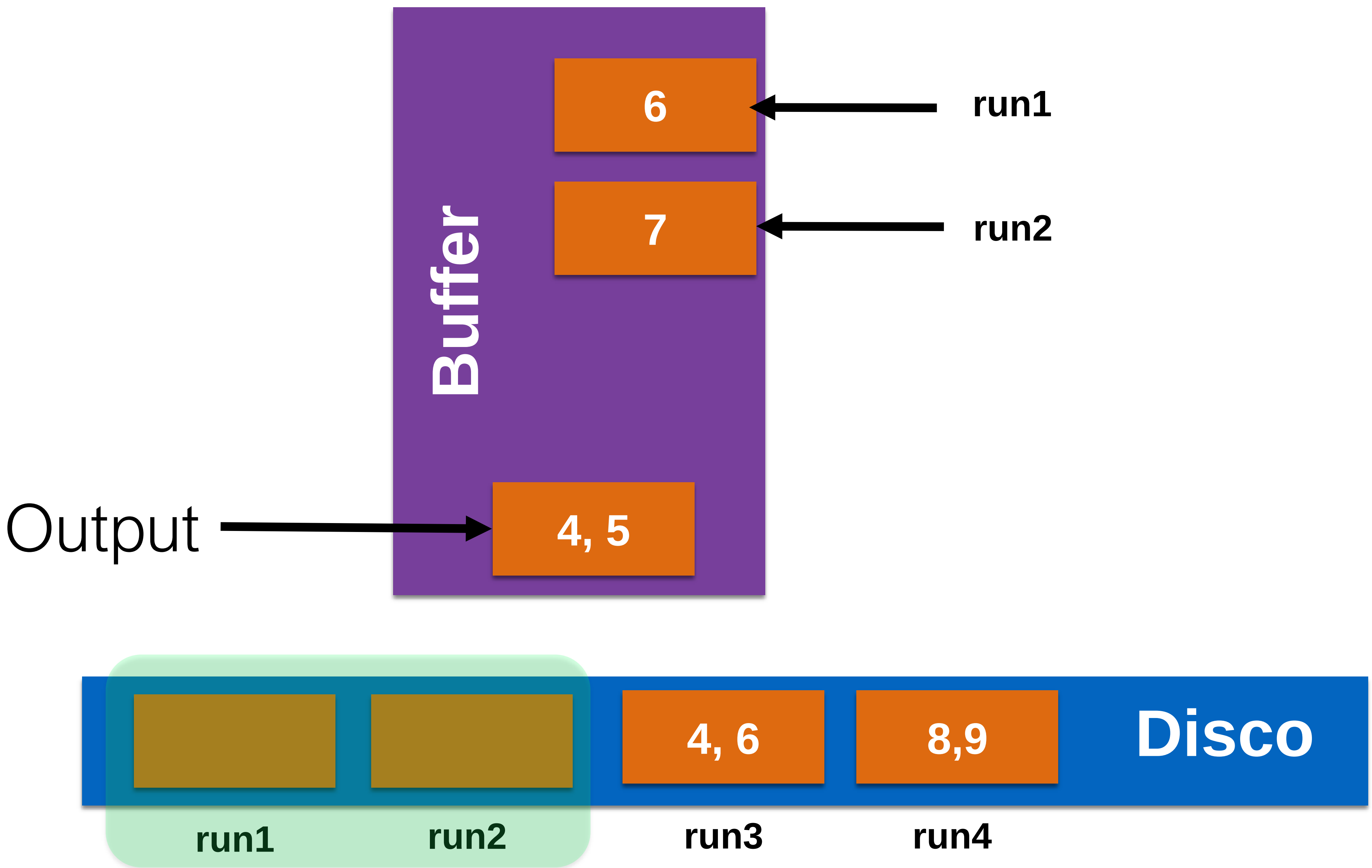
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

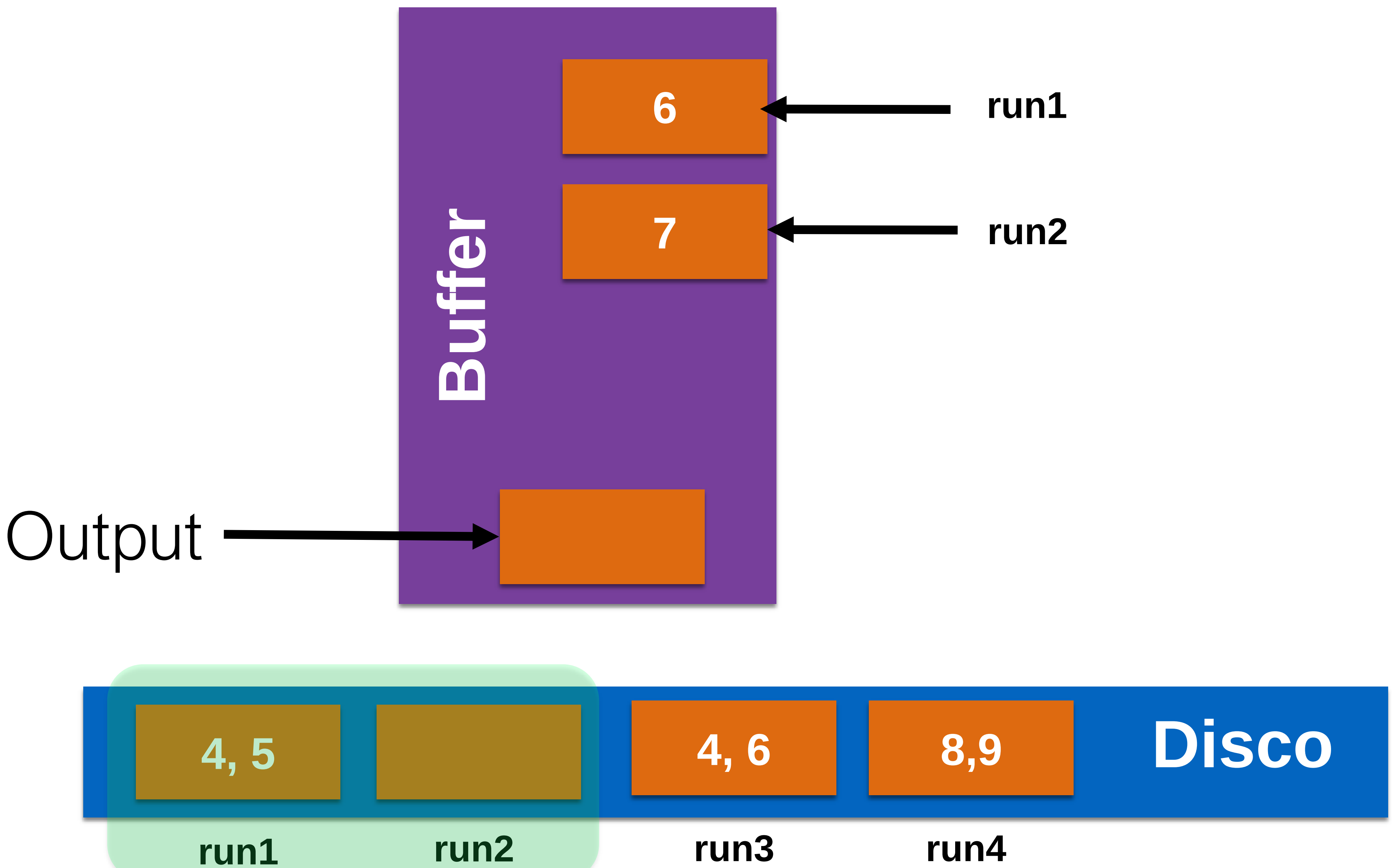
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

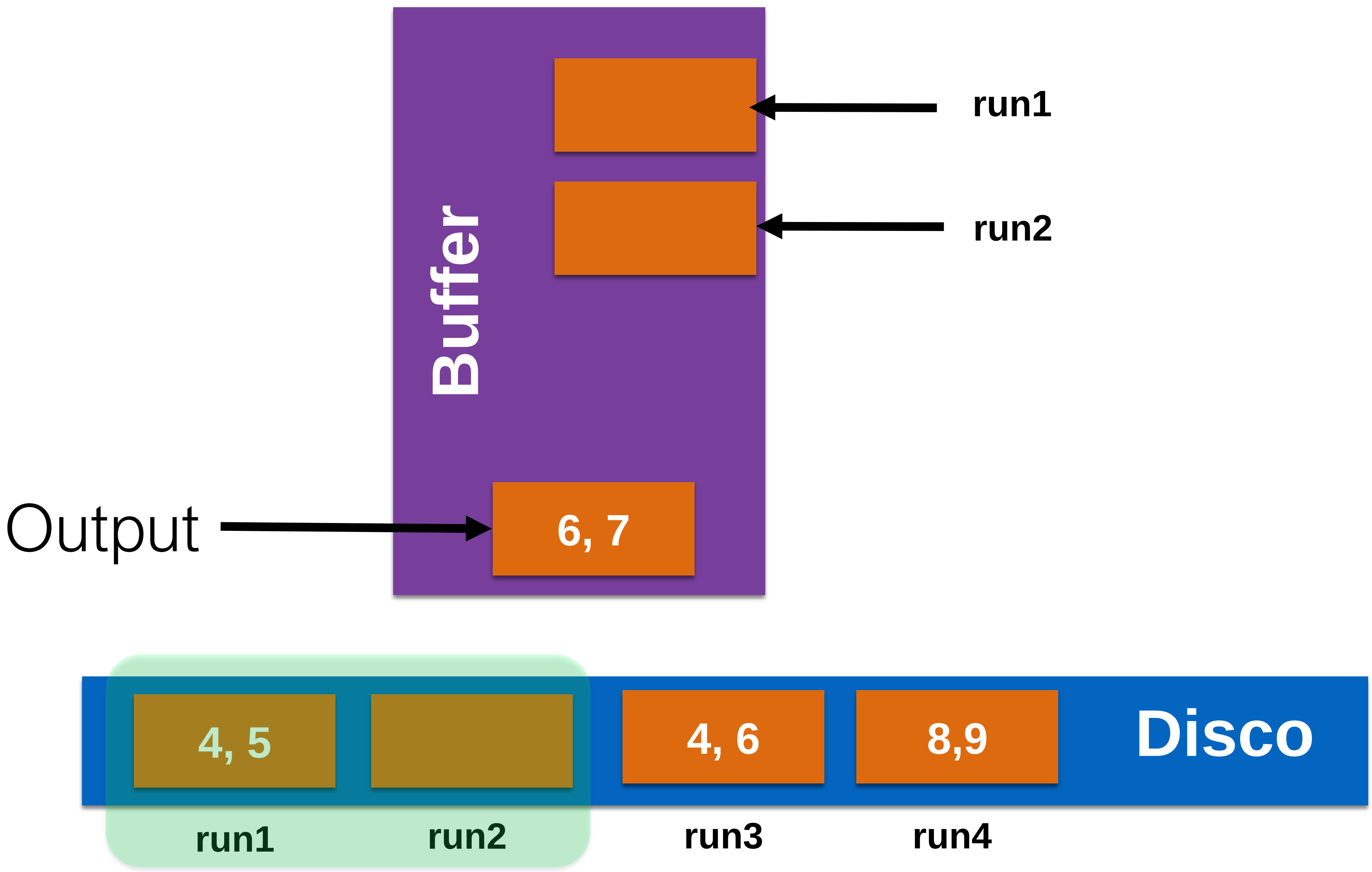
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

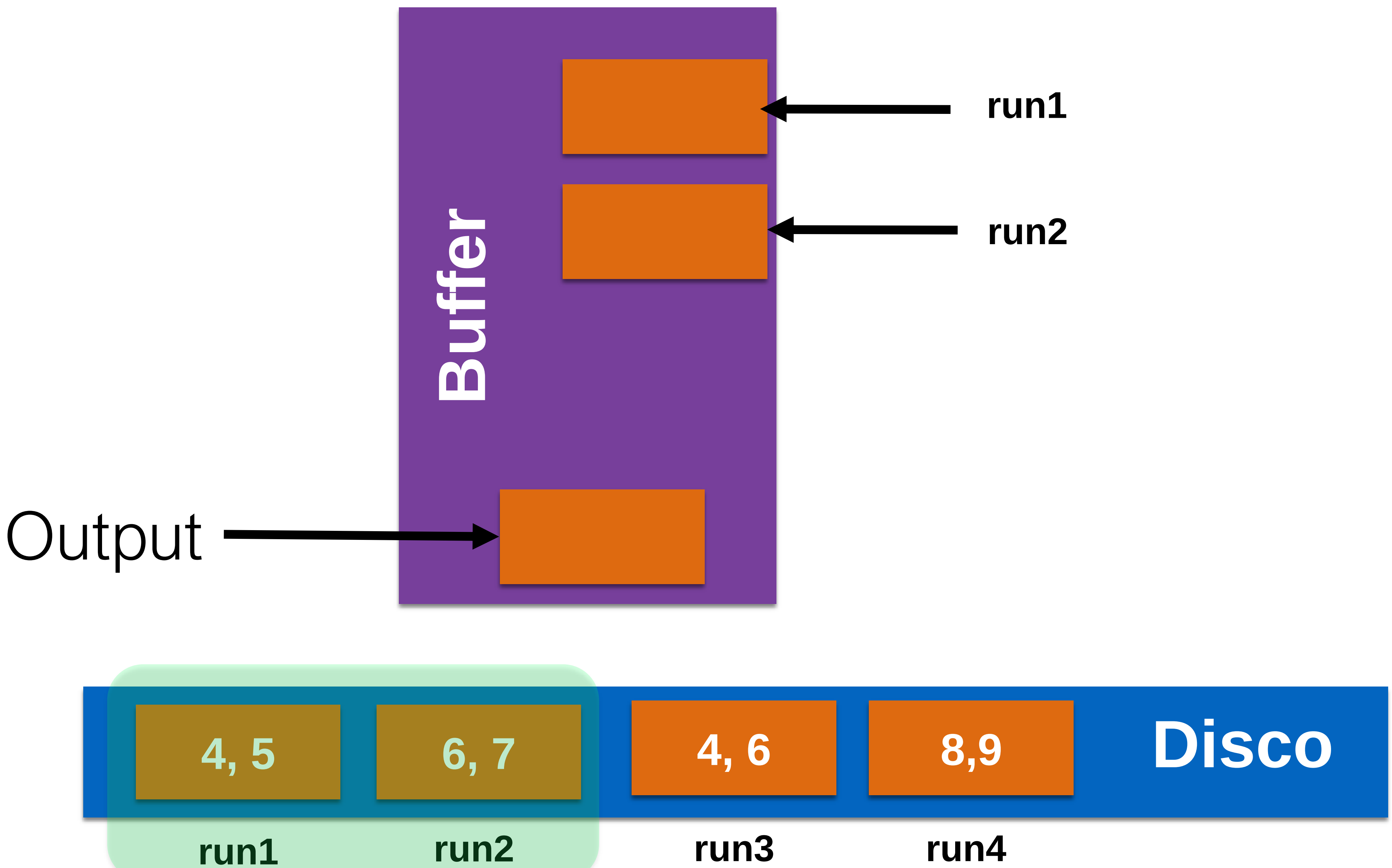
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

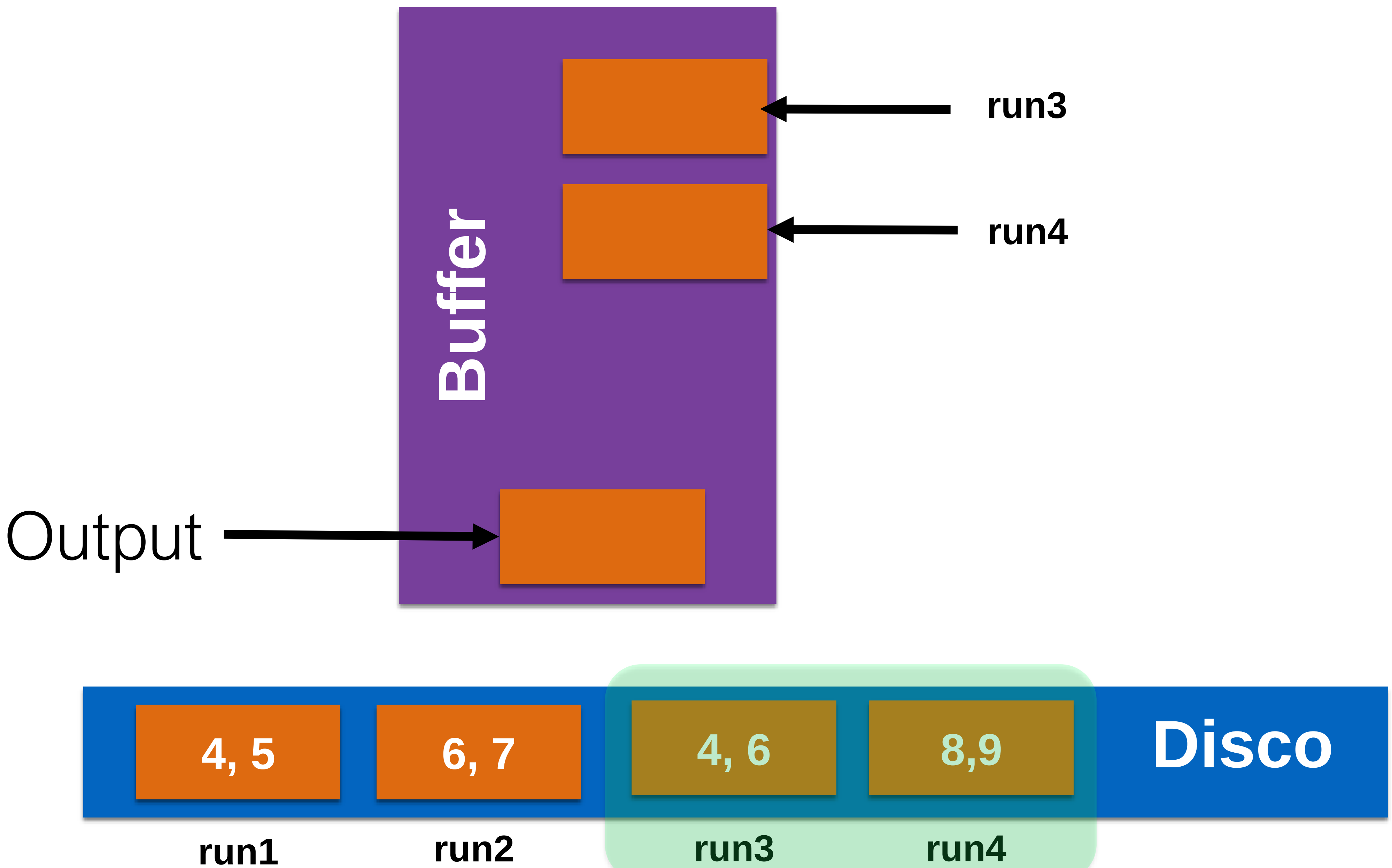
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 1

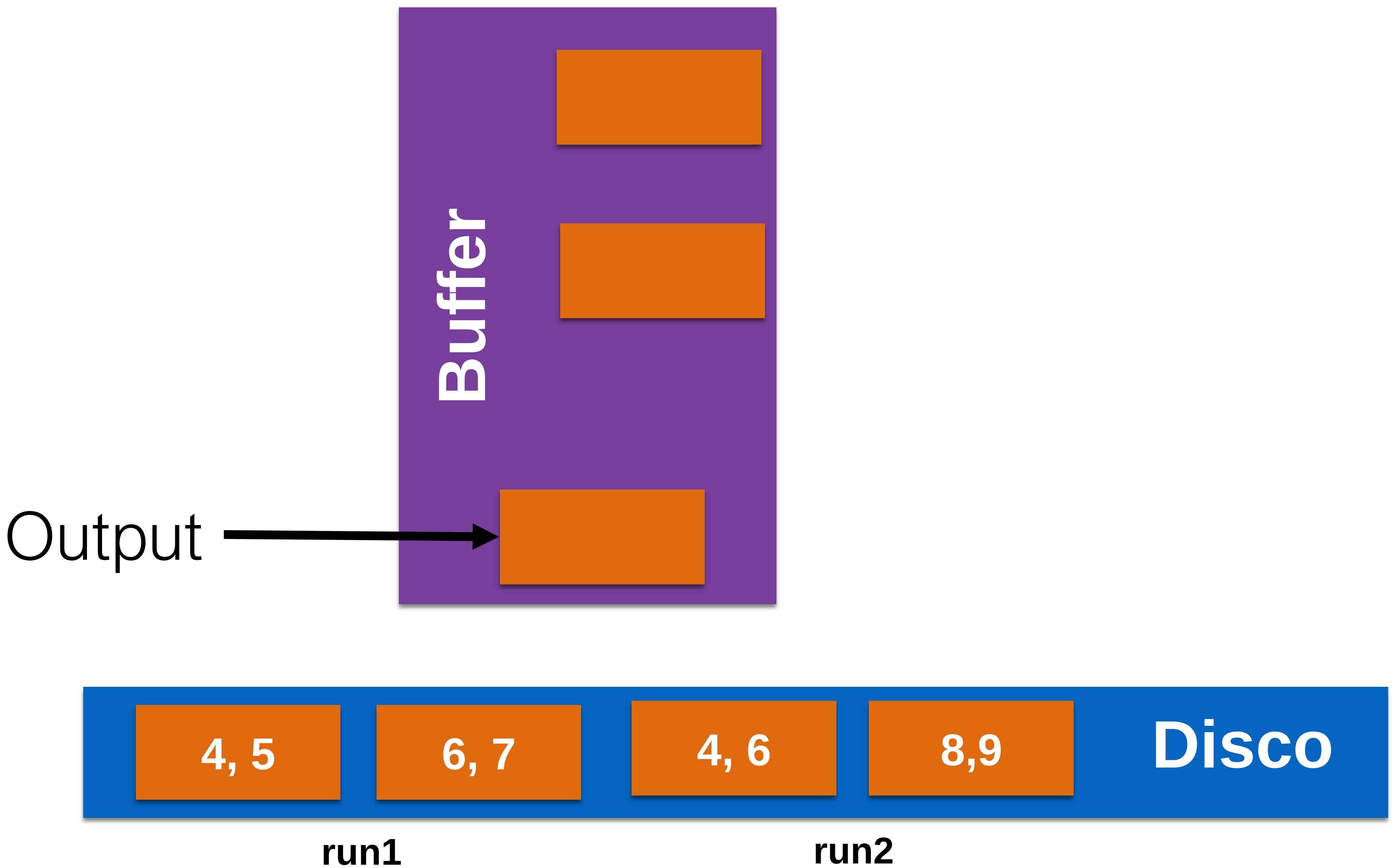
Aquí tenemos runs de tamaño 1, y lo vamos uniendo



External Merge Sort

Fase 2

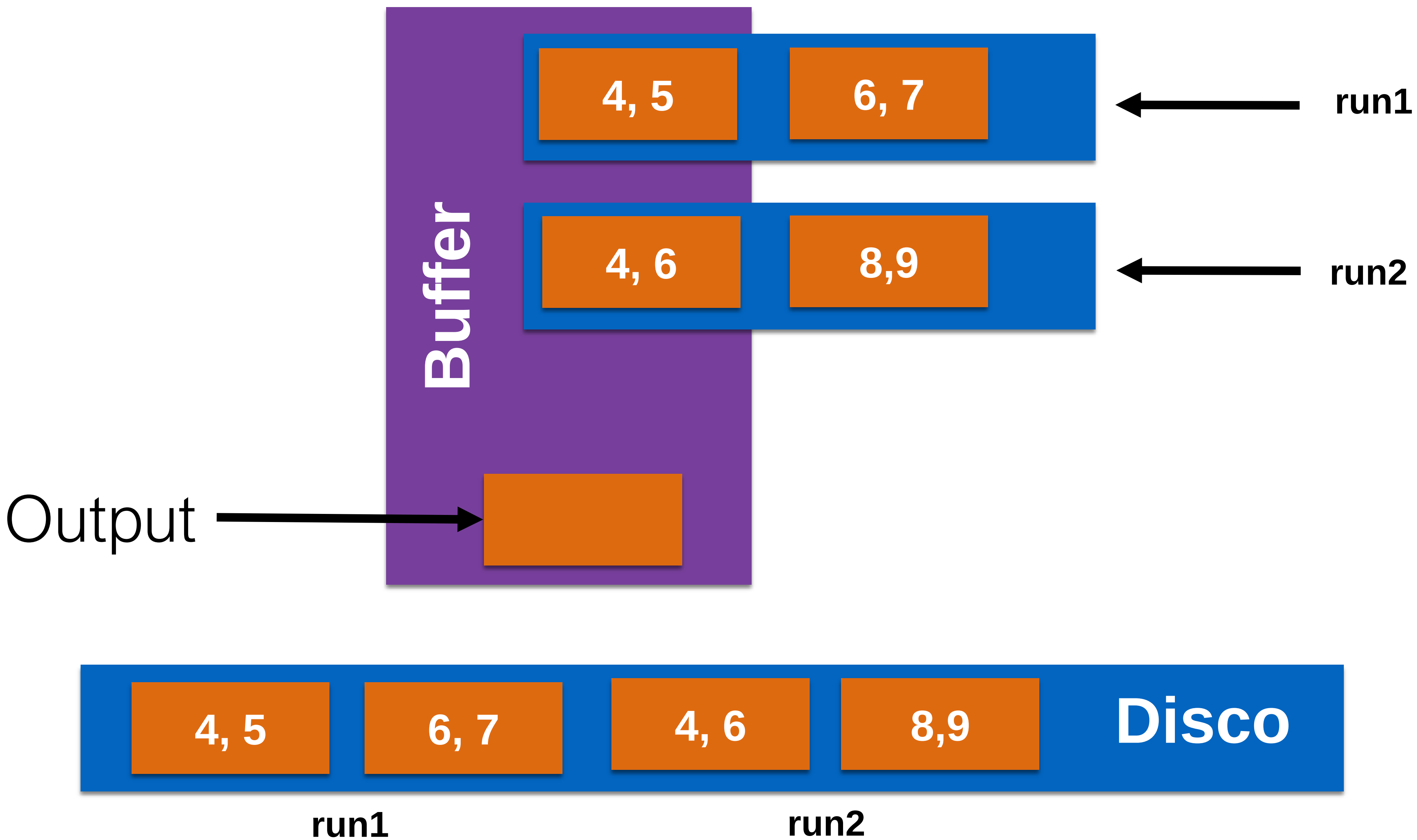
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

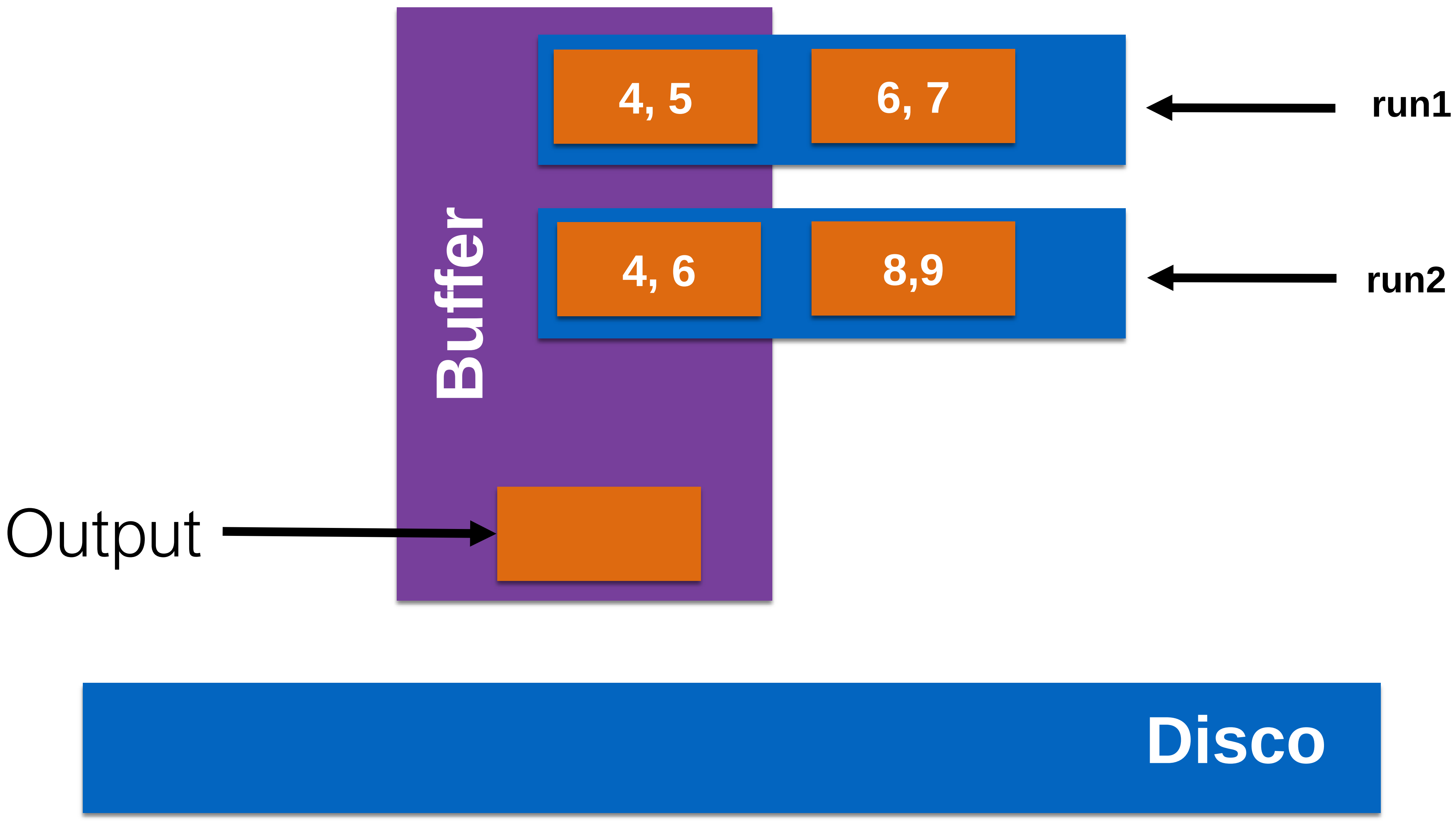
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

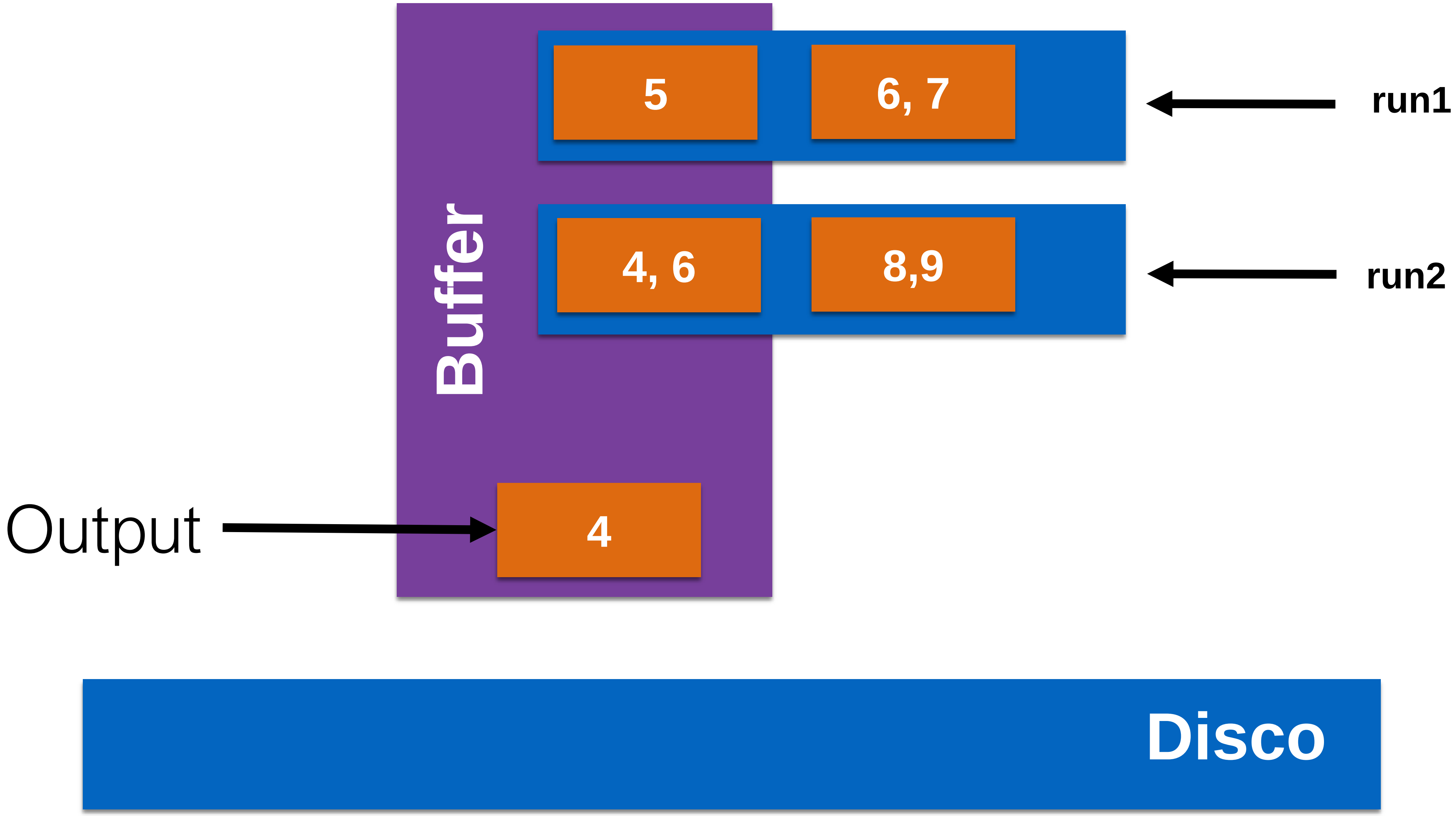
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

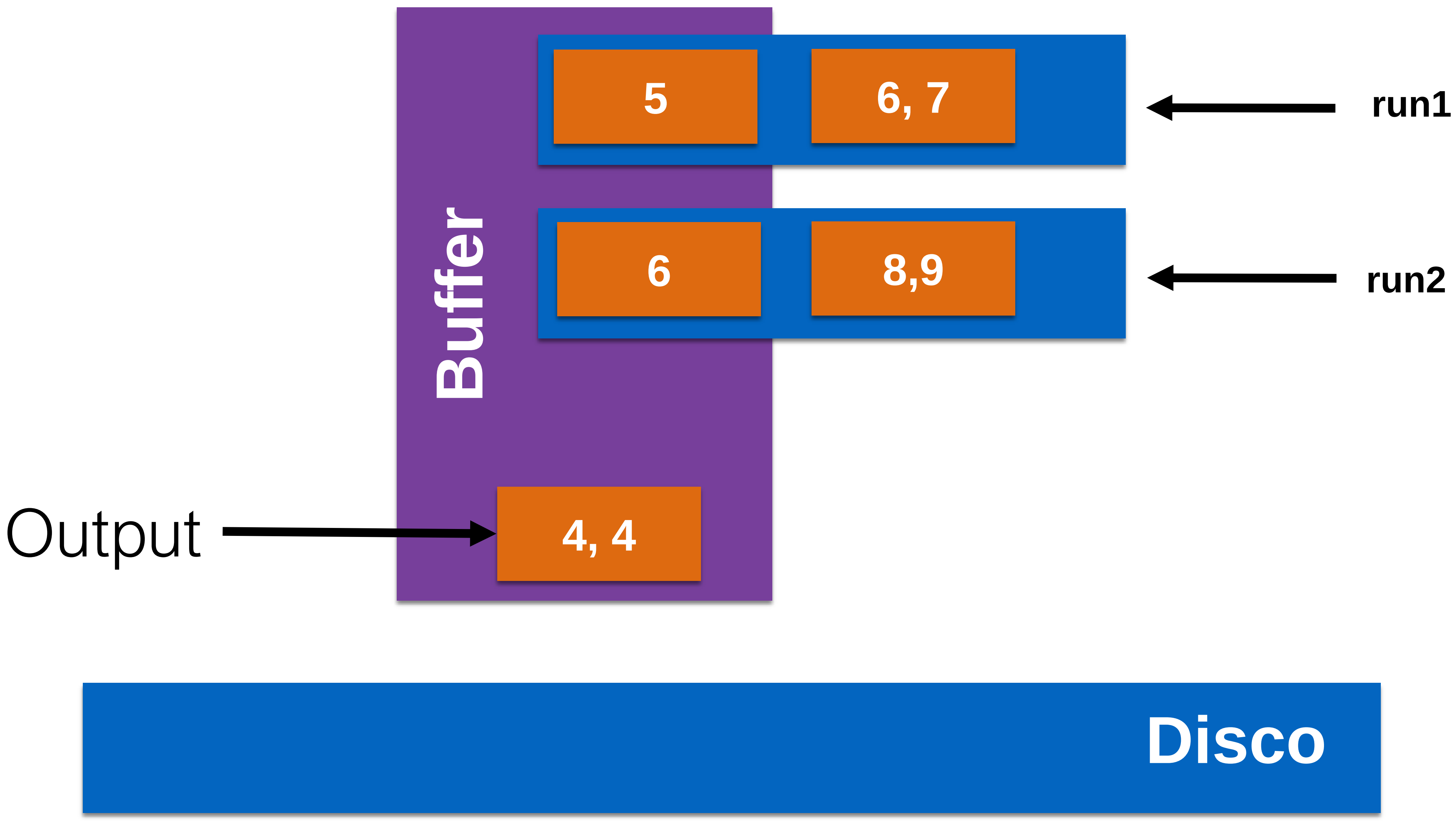
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

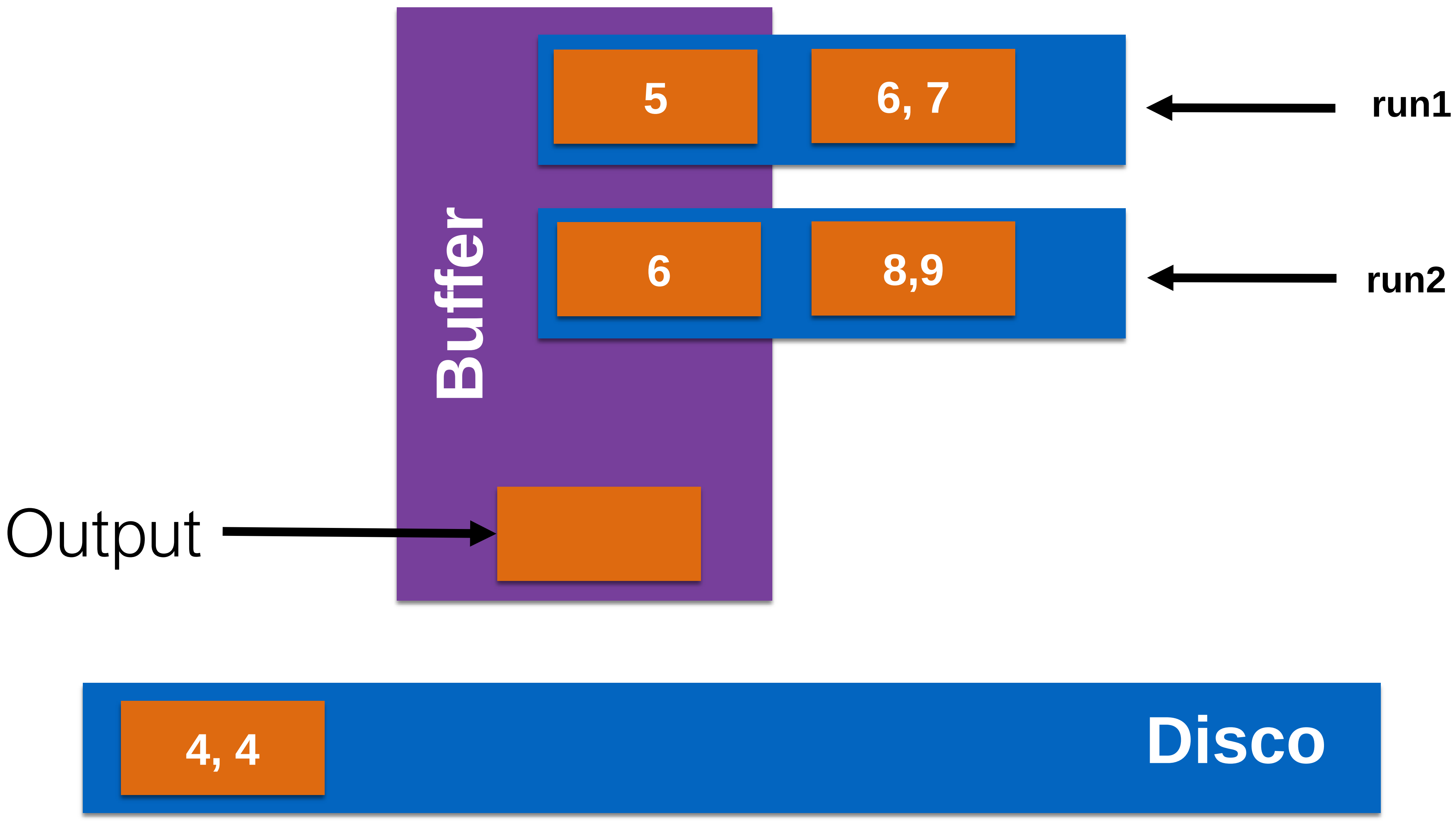
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

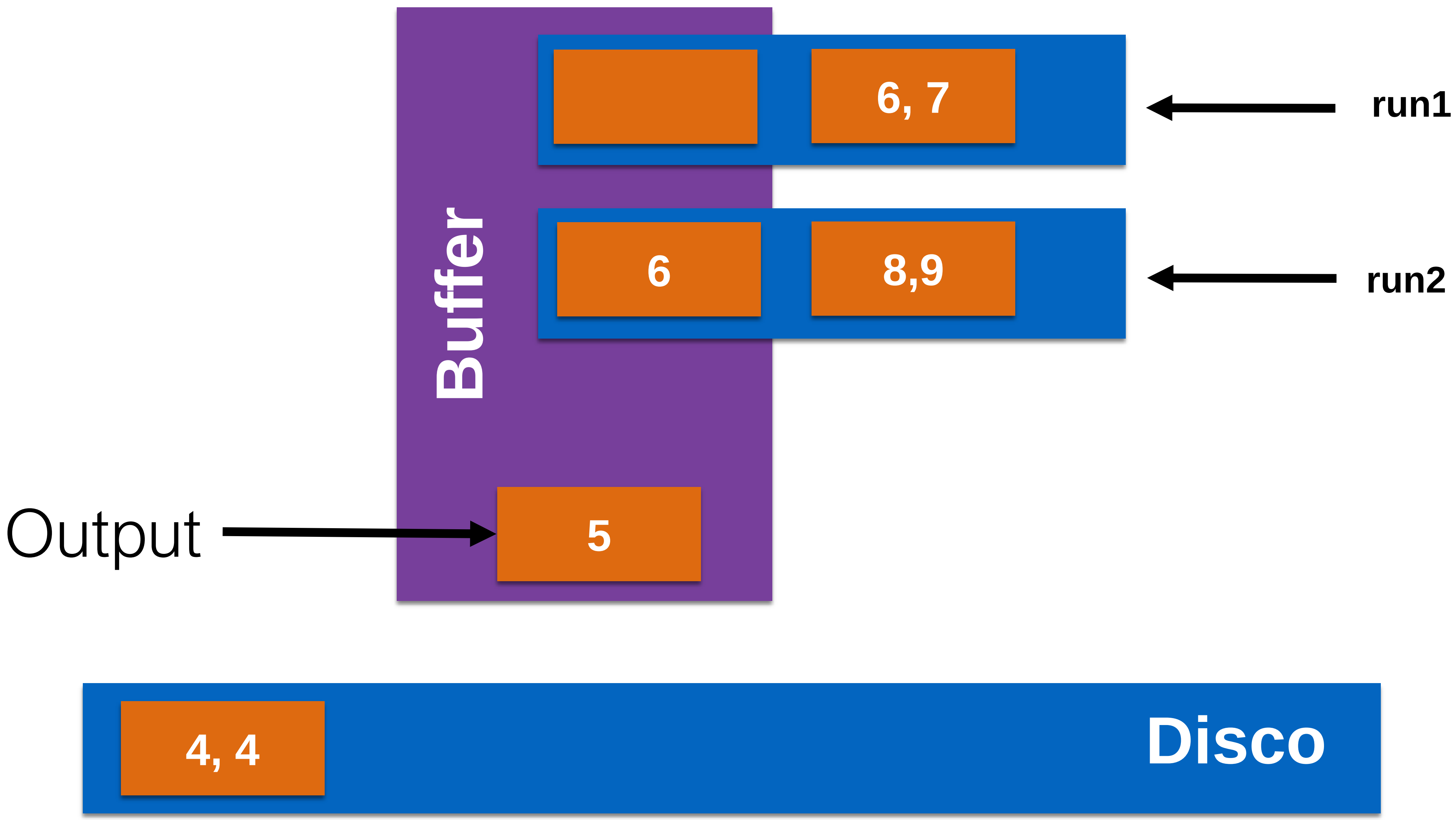
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

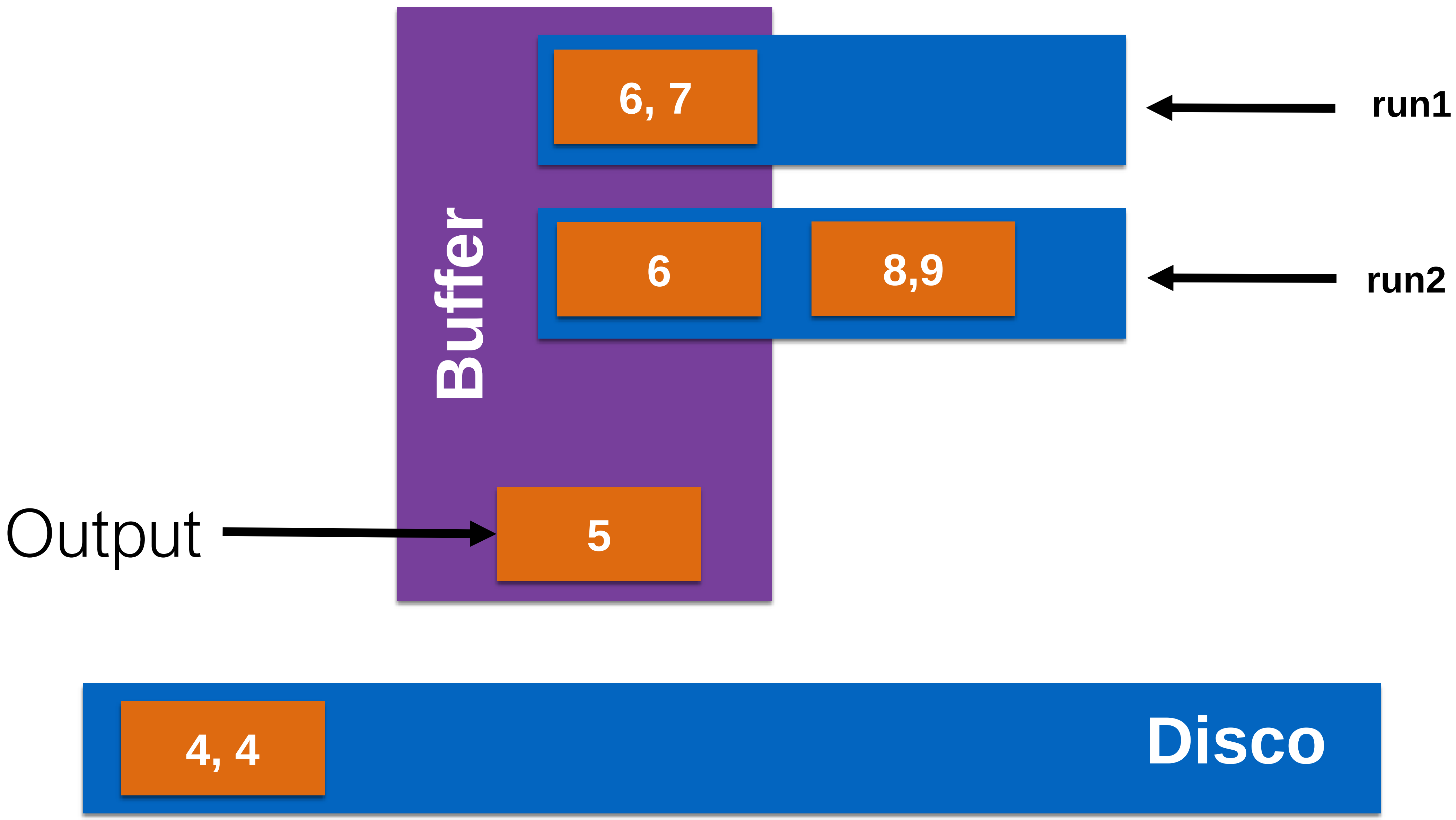
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

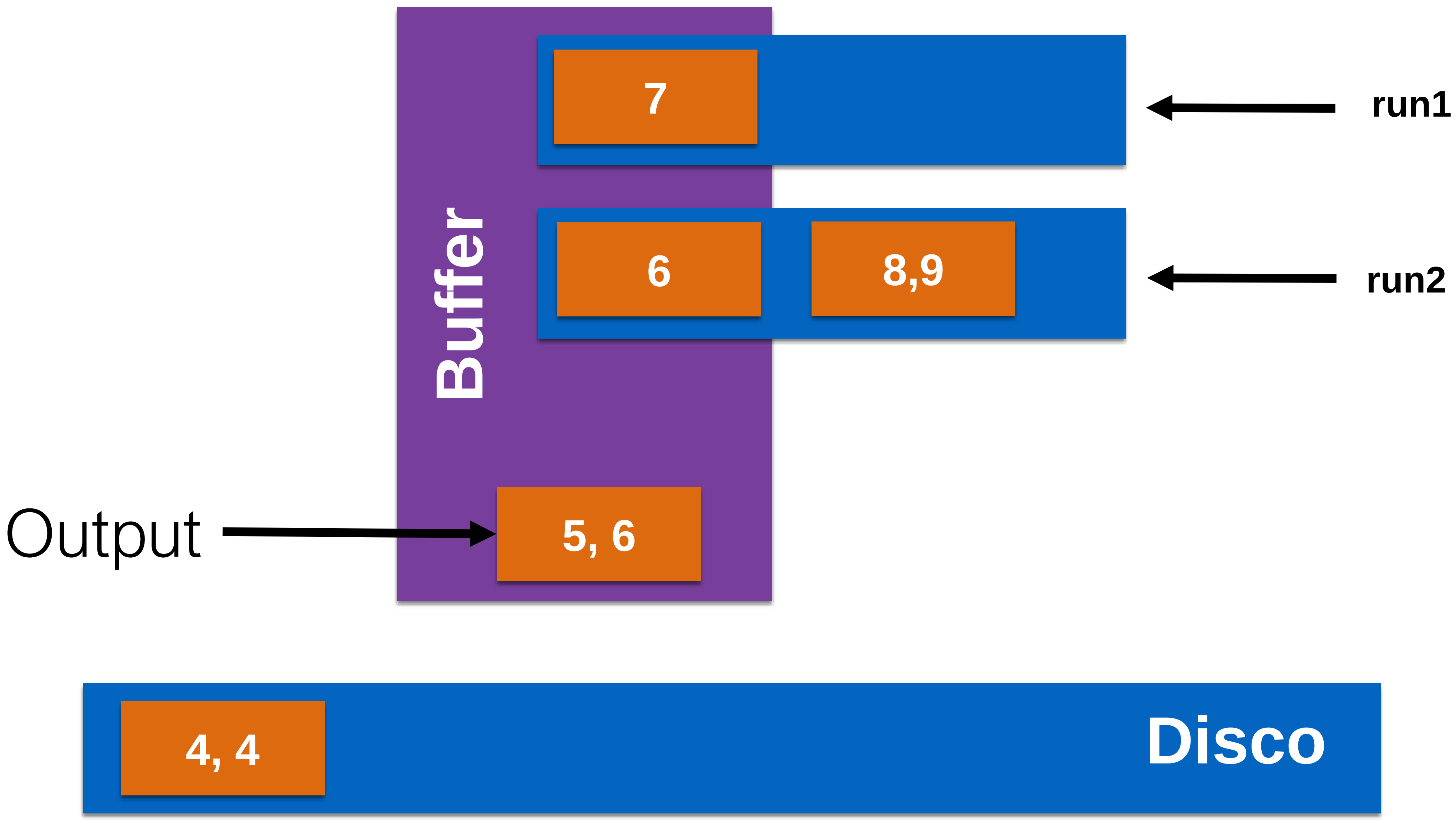
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

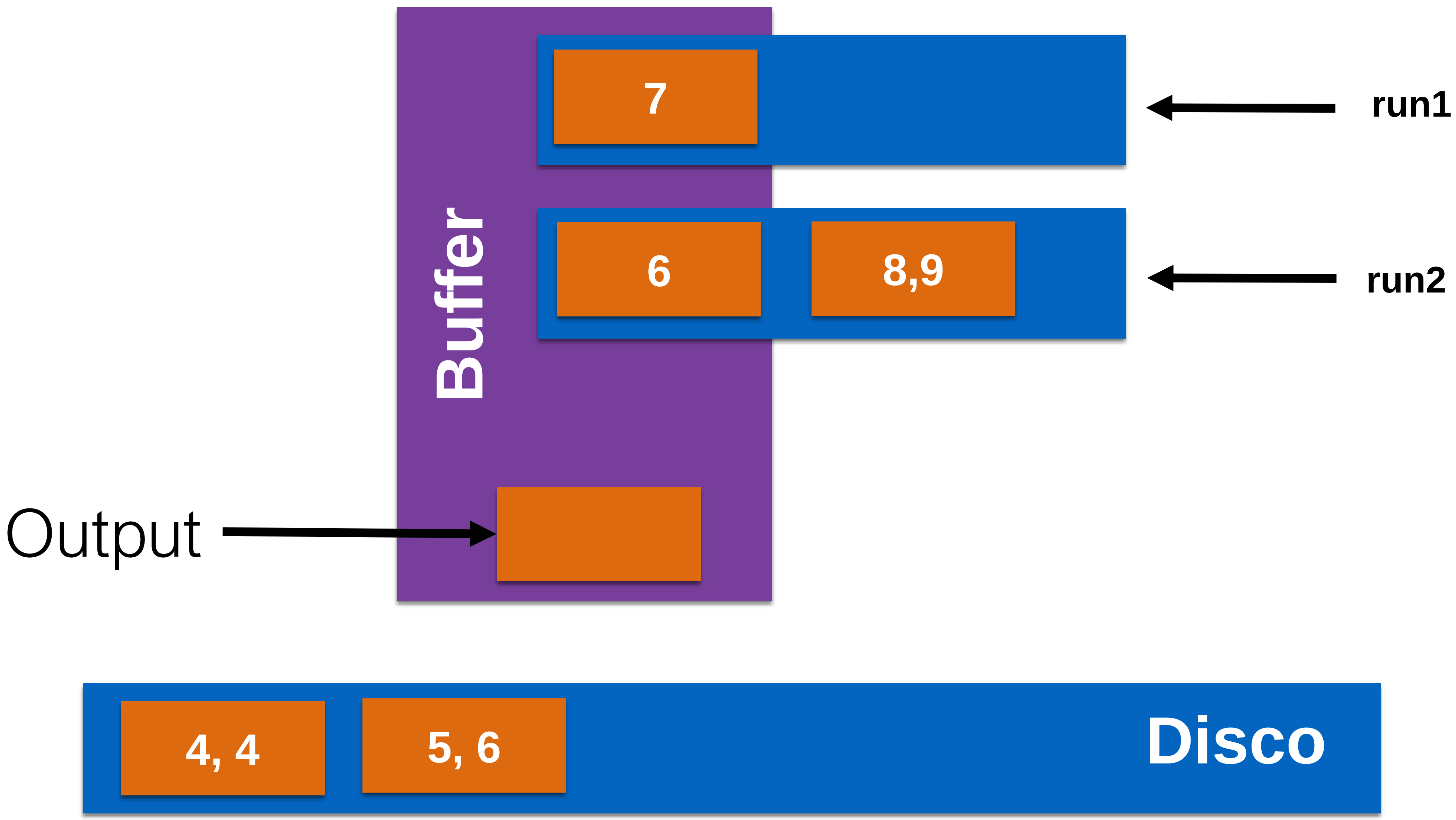
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

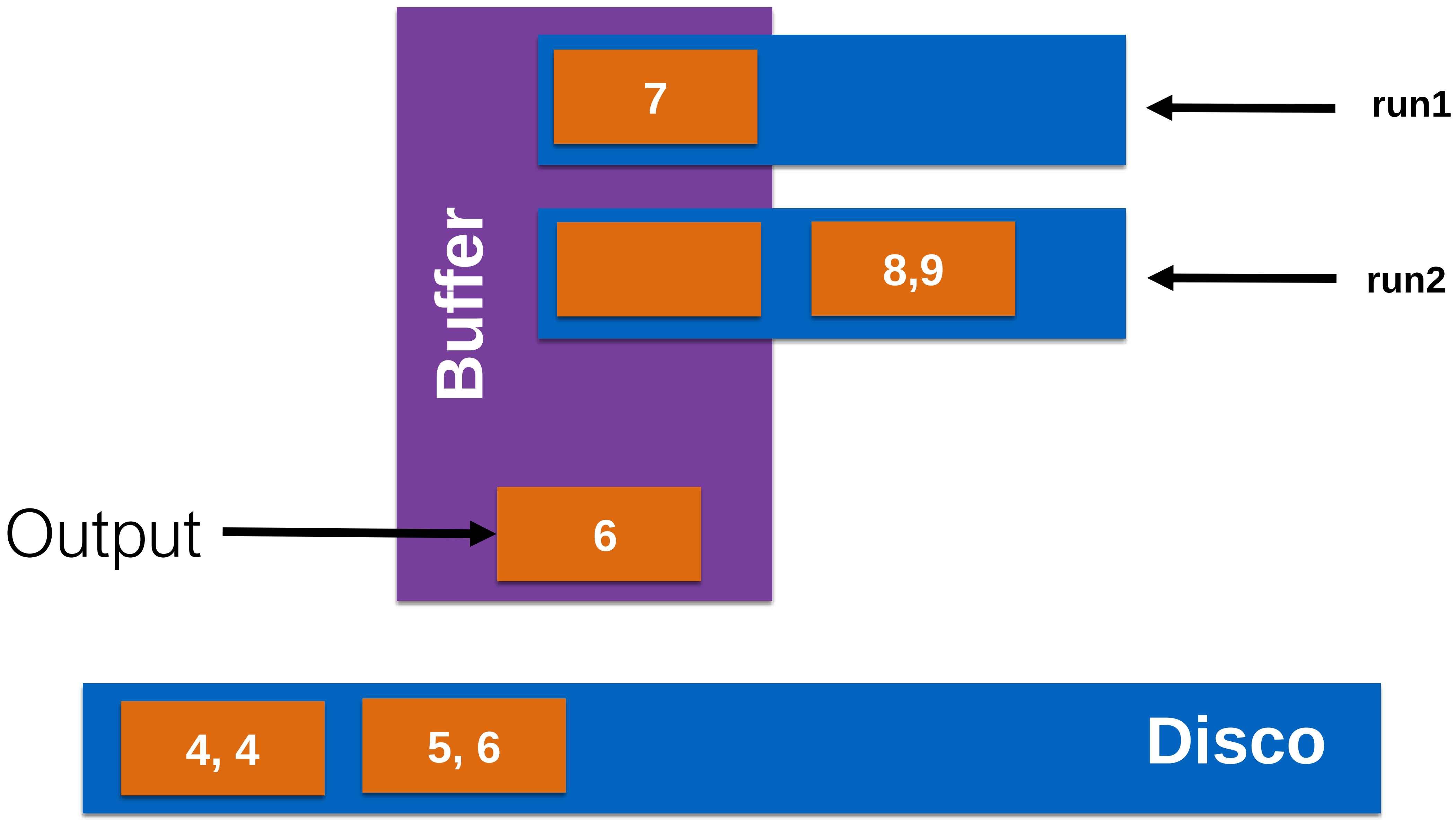
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

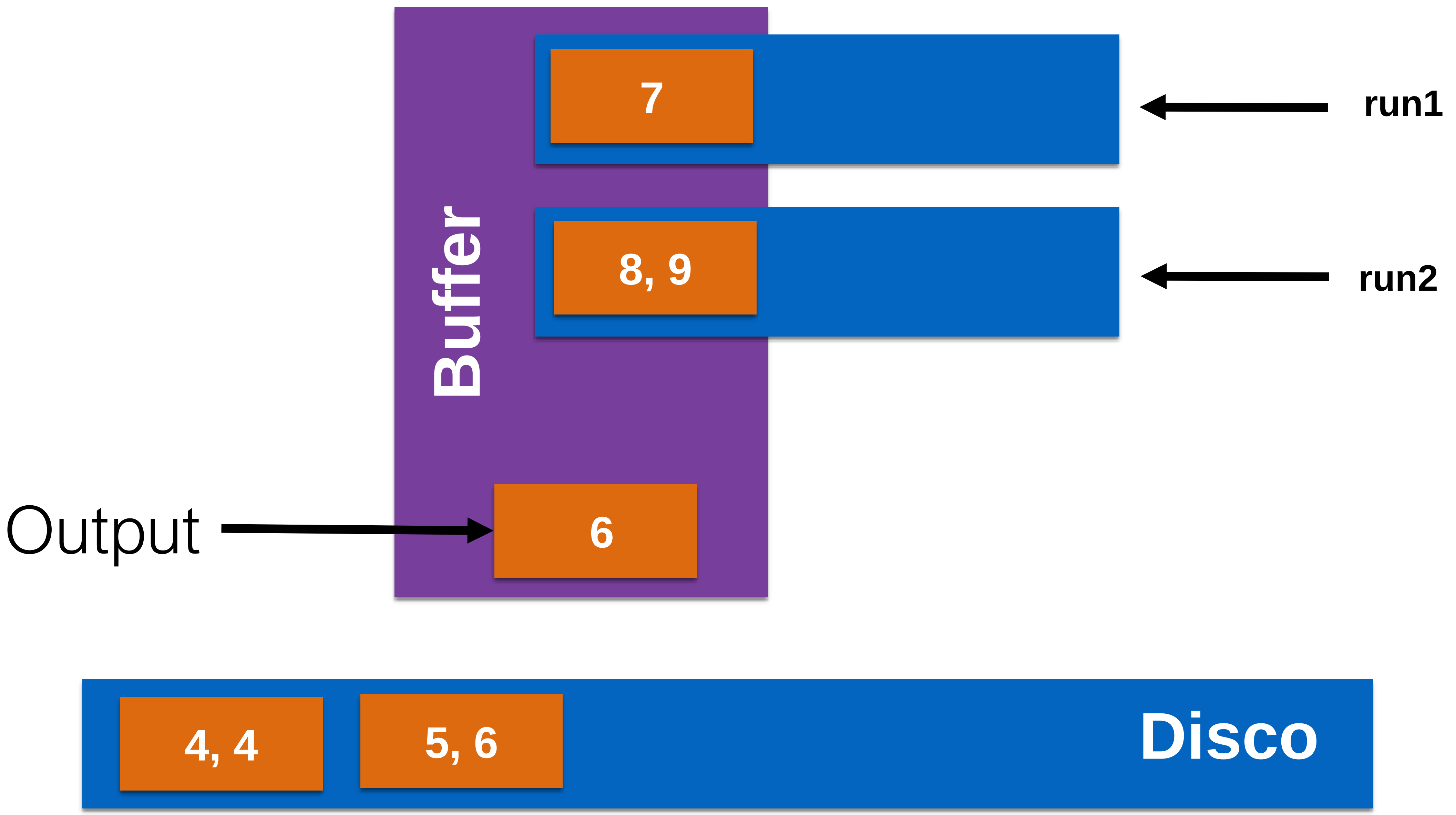
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

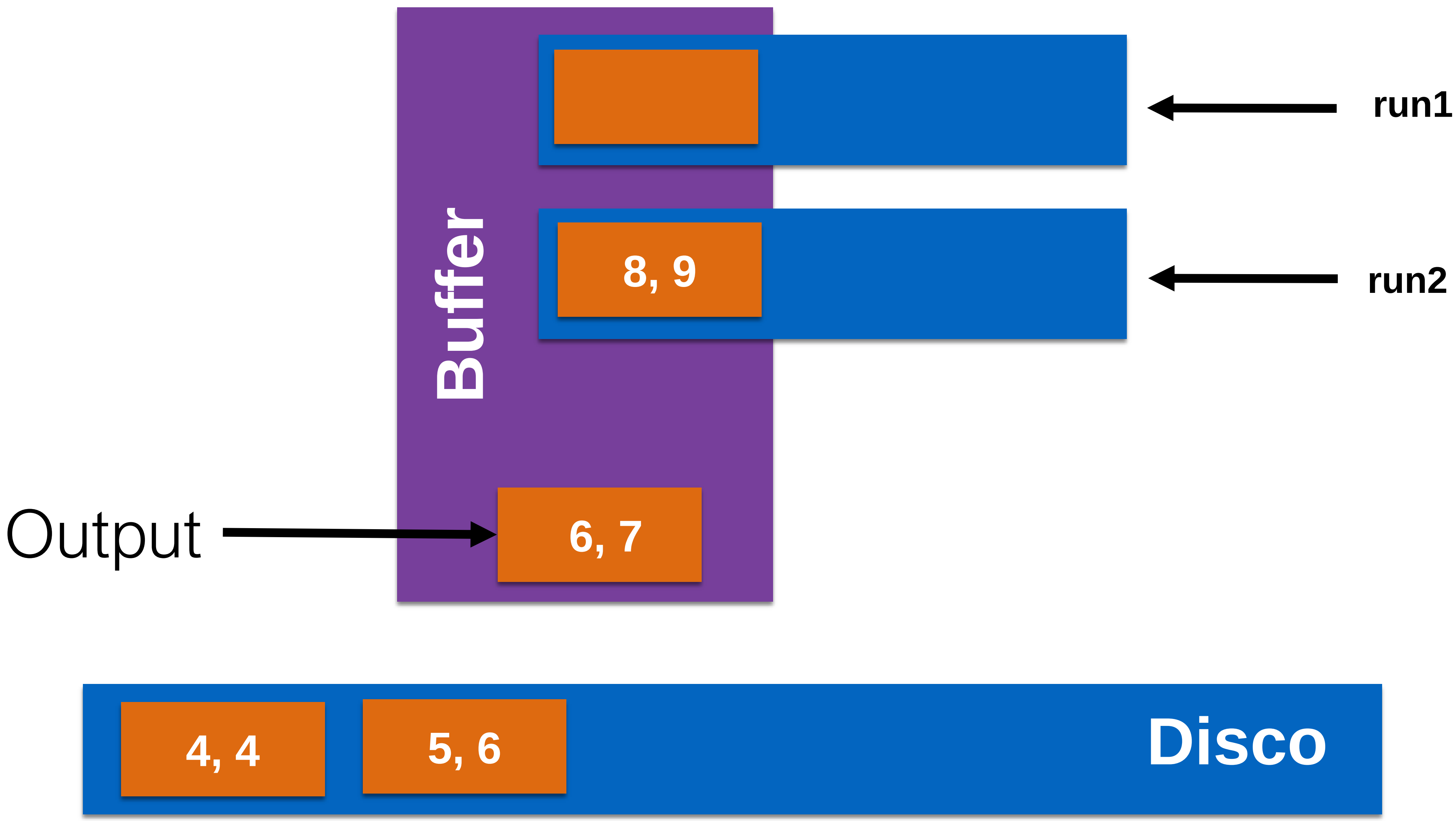
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

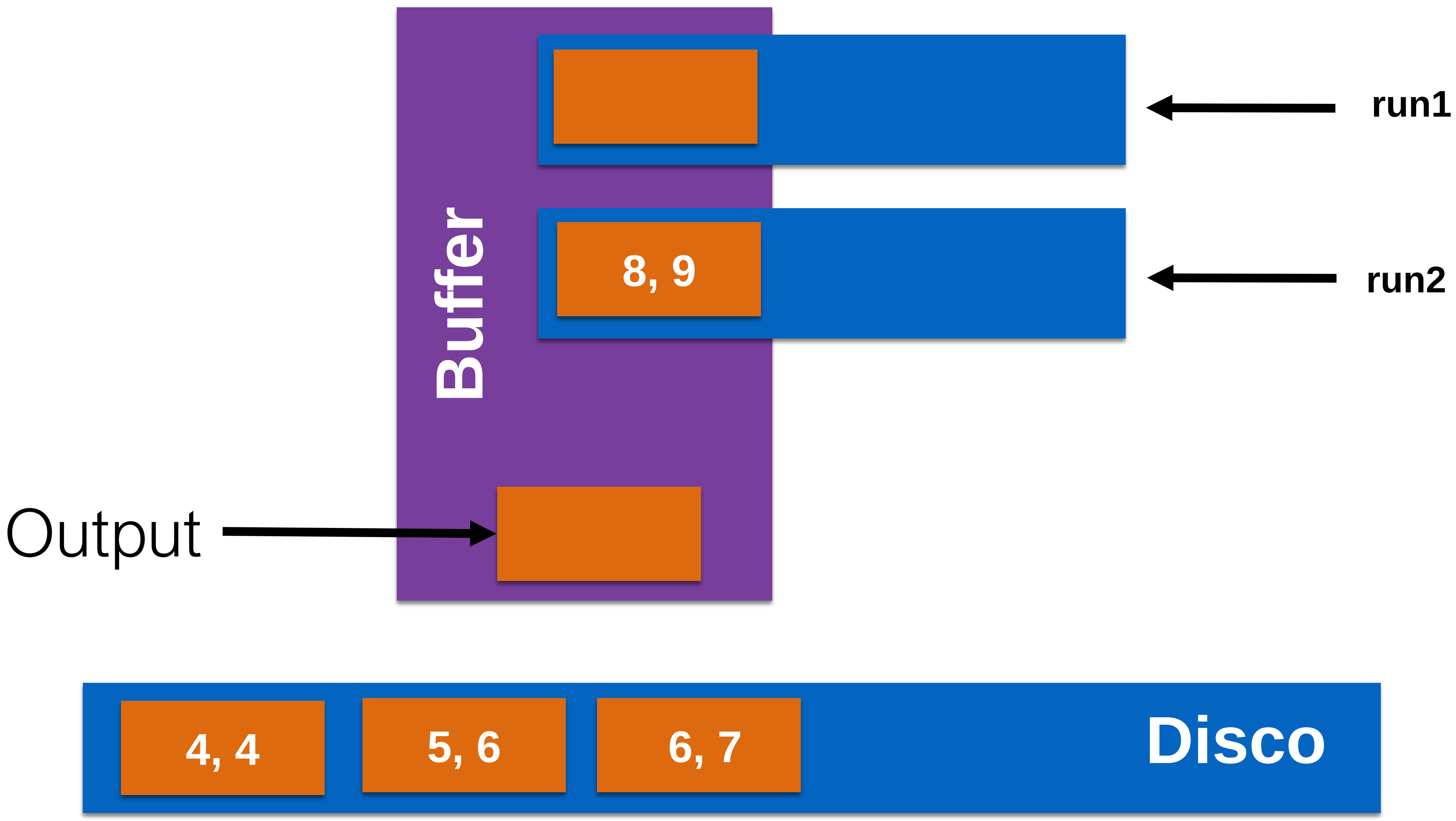
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

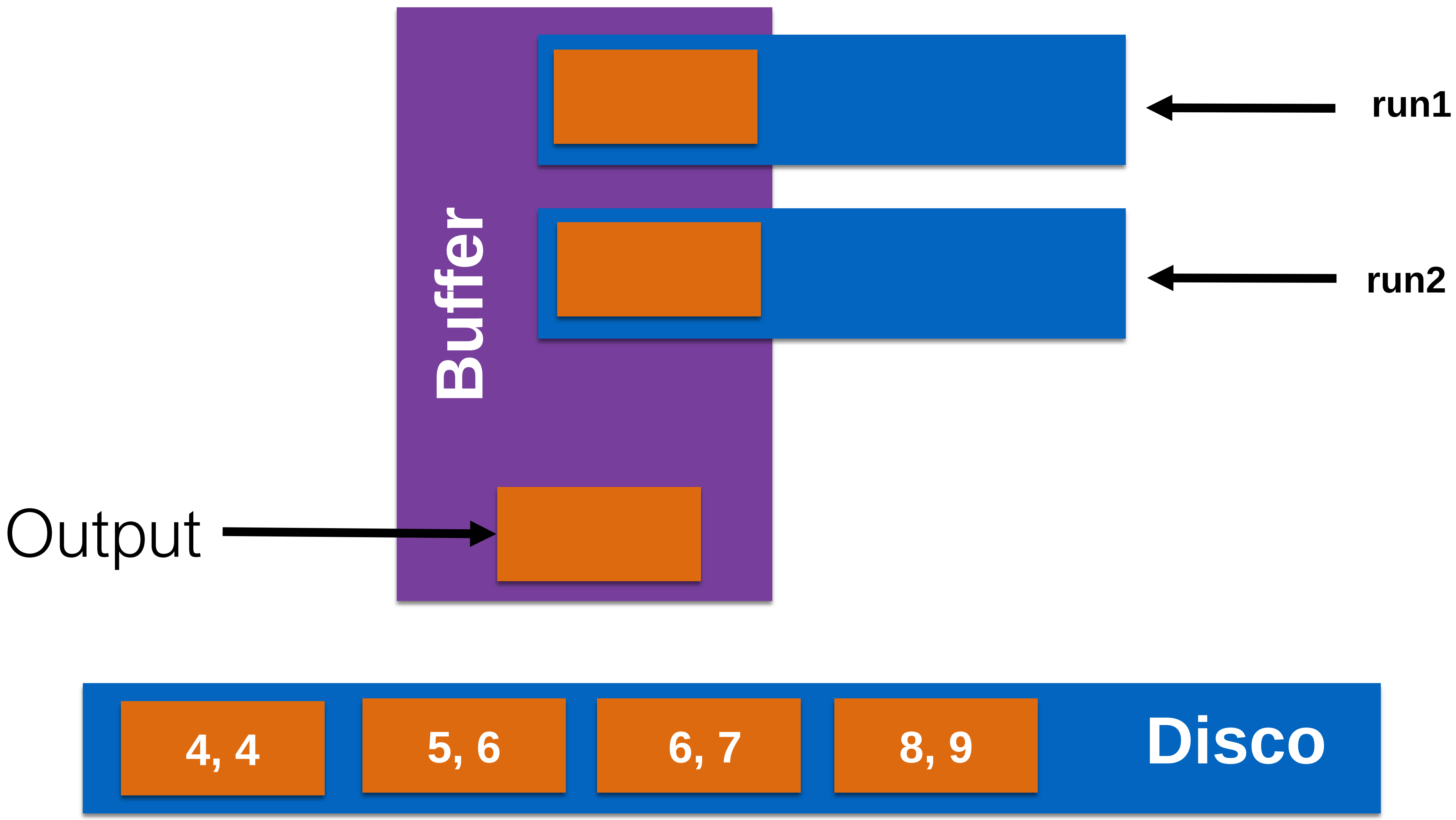
Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Fase 2

Aquí tenemos runs de tamaño 2, y lo vamos uniendo



External Merge Sort

Sea **N** número de páginas del archivo

Sabemos que en cada fase se leen todas las páginas y luego se escriben a disco

Número máximo de fases es:

$$\underbrace{1}_{\text{Fase 0}} + \underbrace{\lceil \log_2 N \rceil}_{\textit{Fases}}$$

El costo en I/O es:

$$2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$$

External Merge Sort

Si tenemos una tabla de 8 GB y páginas de 8 KB ~ 1048576 páginas en total

Costo en I/O es:

$$2 \cdot 1048576 \cdot (1 + \lceil log_2 1048576 \rceil) = 44040192$$

Si cada I/O toma 0.1 ms, ordenar tarda **1.2 horas**

External Merge Sort

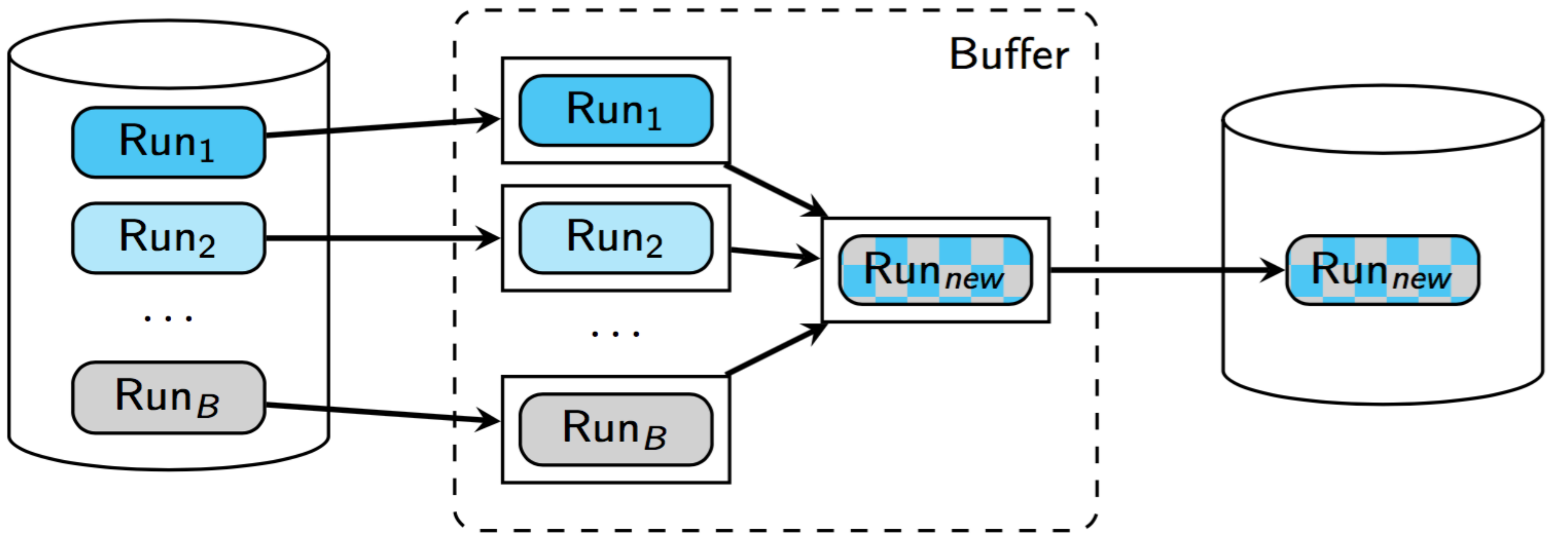
Optimizado

Podemos mejorar el desempeño de nuestro algoritmo

External Merge Sort

Optimizado

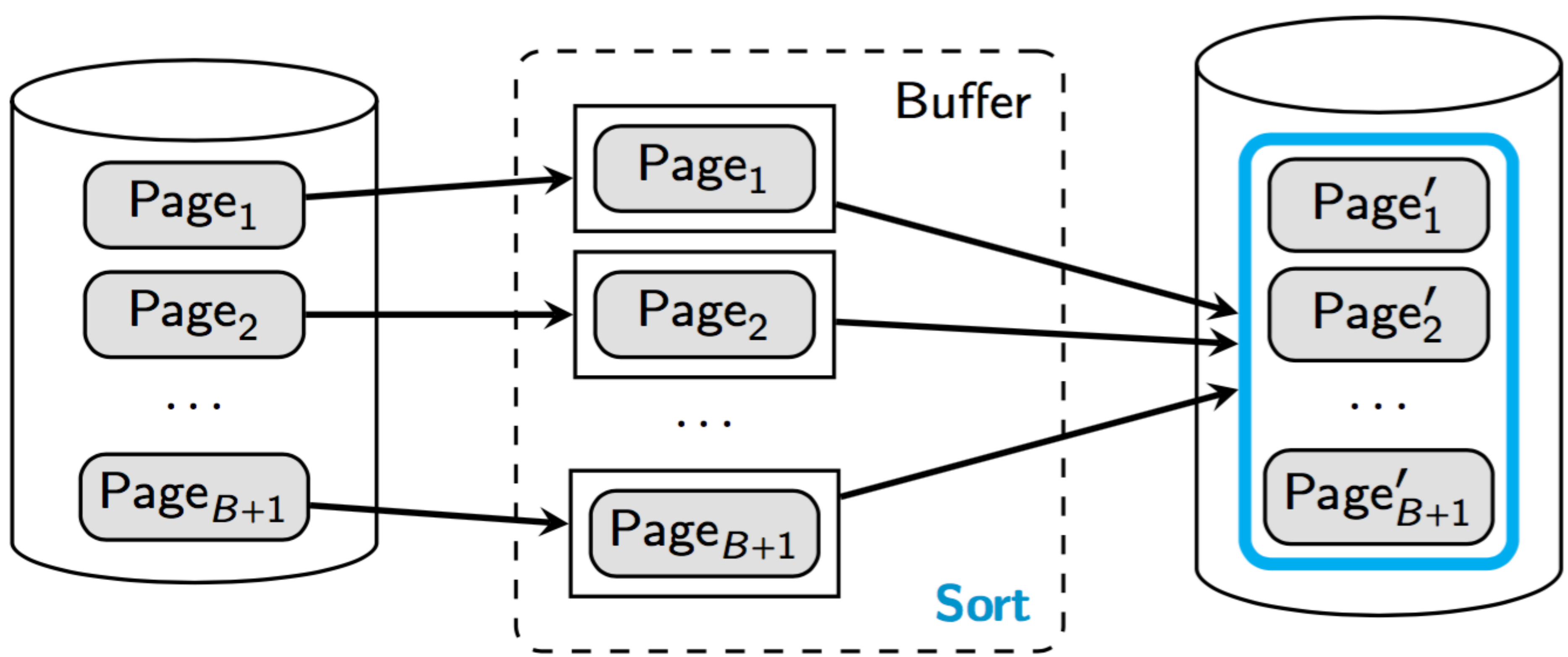
En vez de 3 páginas, tenemos B + 1 páginas en buffer



External Merge Sort

Optimizado

Además, tendremos runs iniciales de B + 1 páginas



External Merge Sort

Optimizado

Supongamos que queremos ordenar la relación **R(a)** = {20, 19, 18, …, 1}

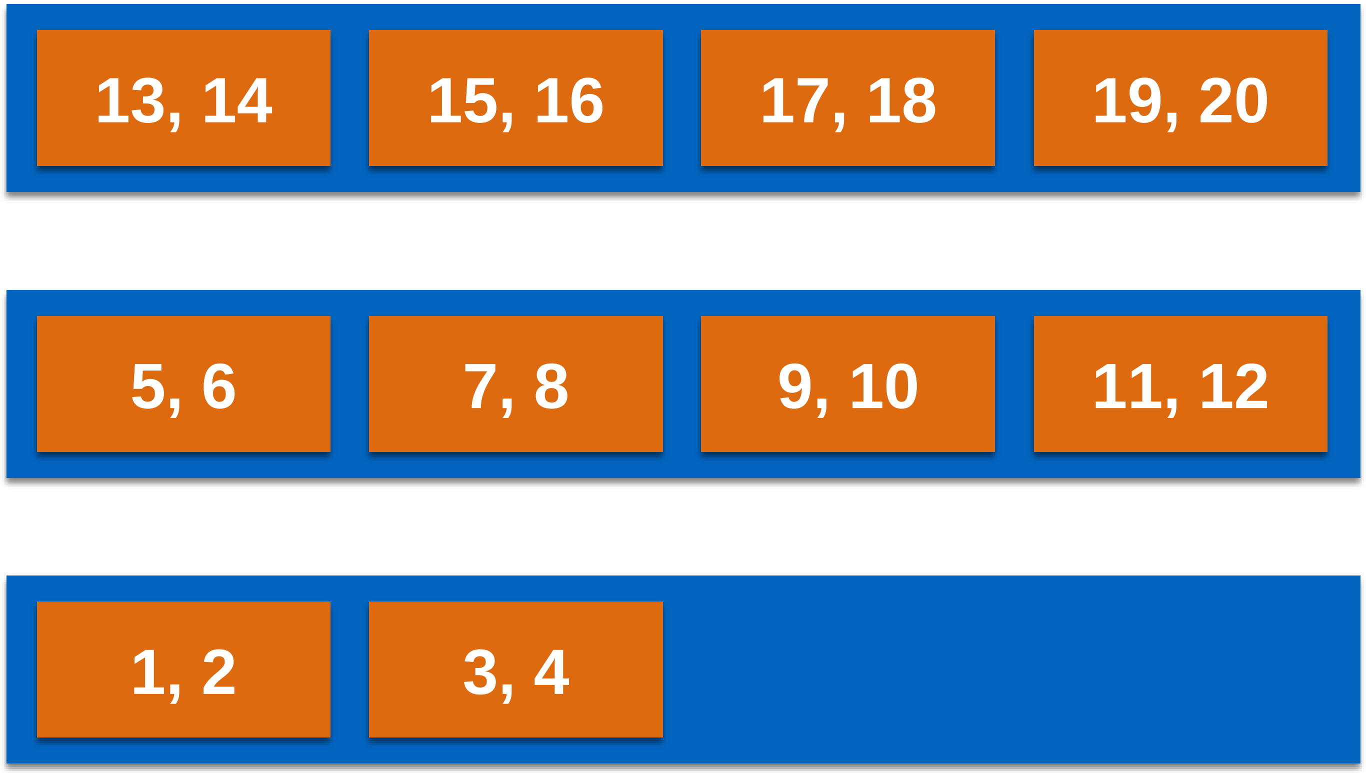
El buffer alcanza para 4 páginas (B+1 = 4)

Runs iniciales tienen 4 páginas

External Merge Sort

Optimizado

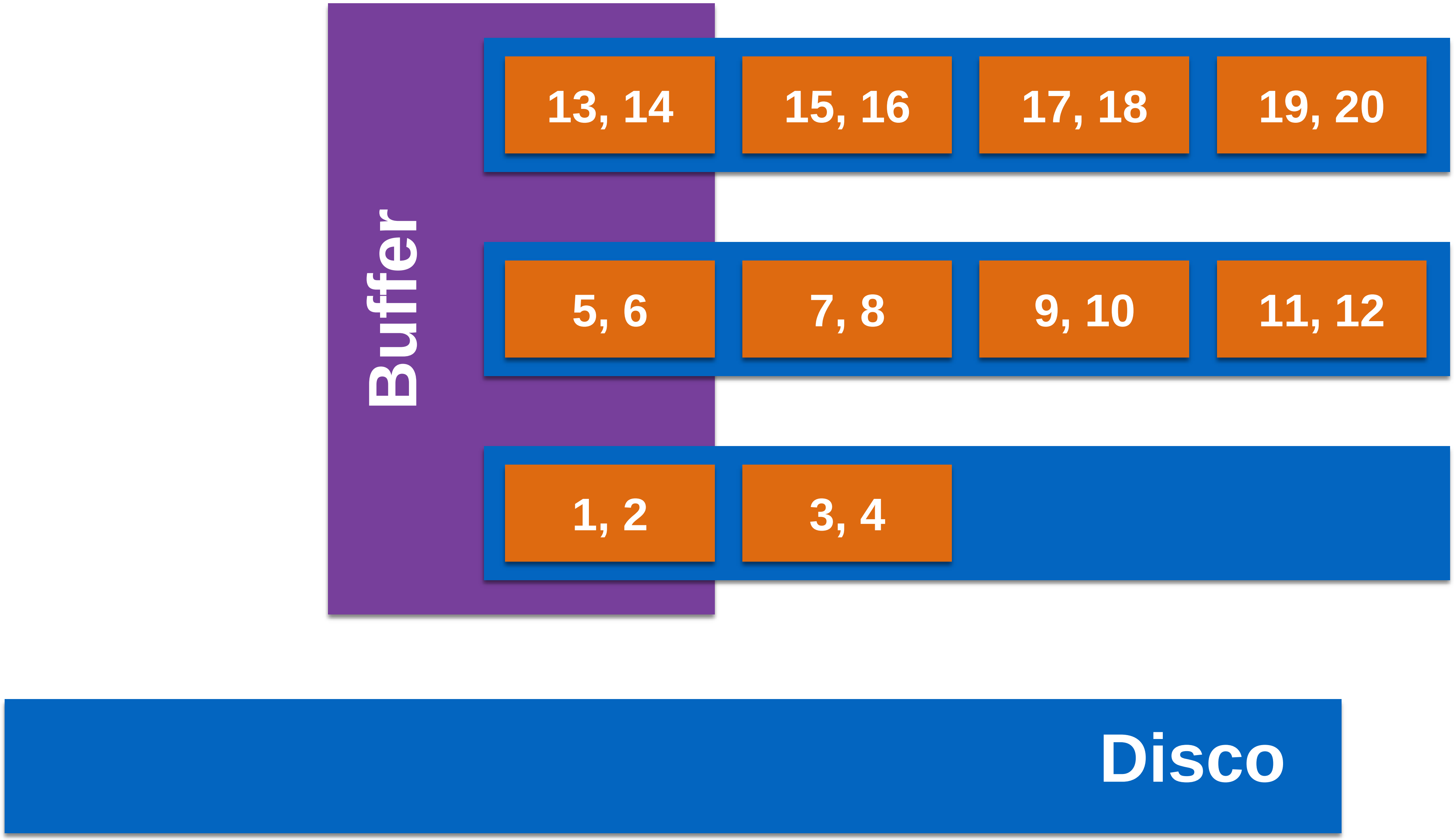
Ordenamos los 3 Runs iniciales:



External Merge Sort

Optimizado

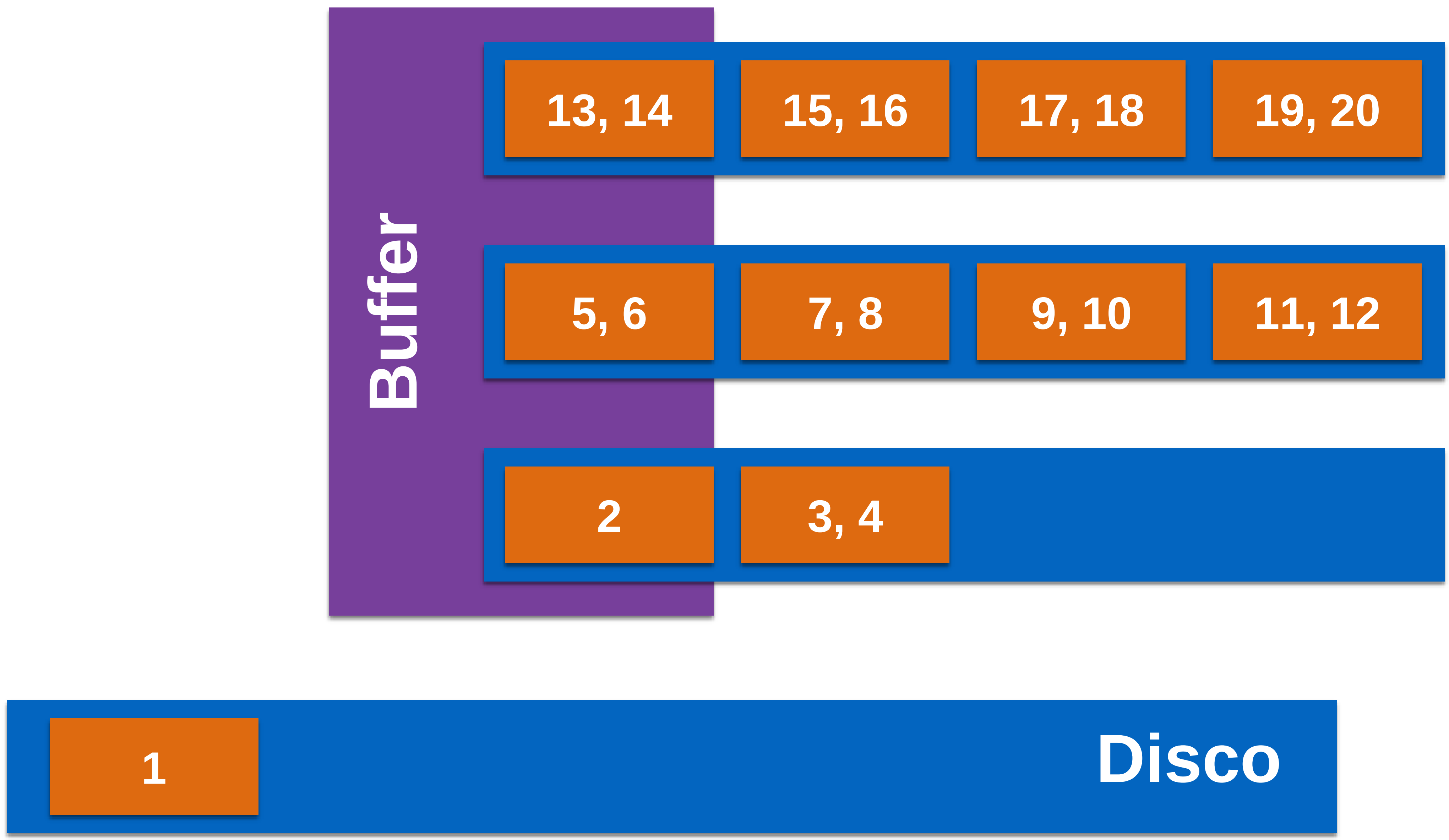
Se van escribiendo las páginas:



External Merge Sort

Optimizado

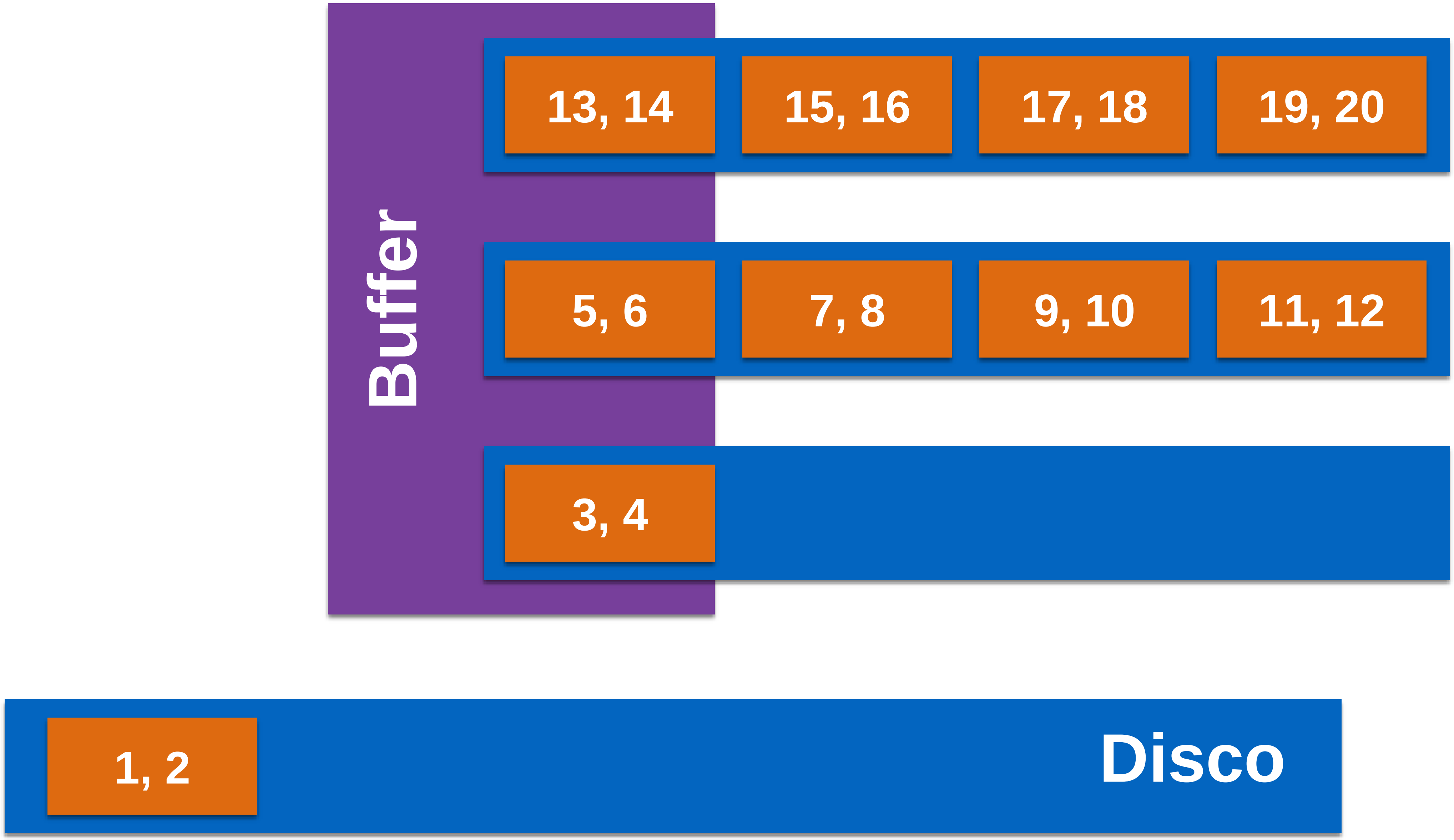
Se van escribiendo las páginas:



External Merge Sort

Optimizado

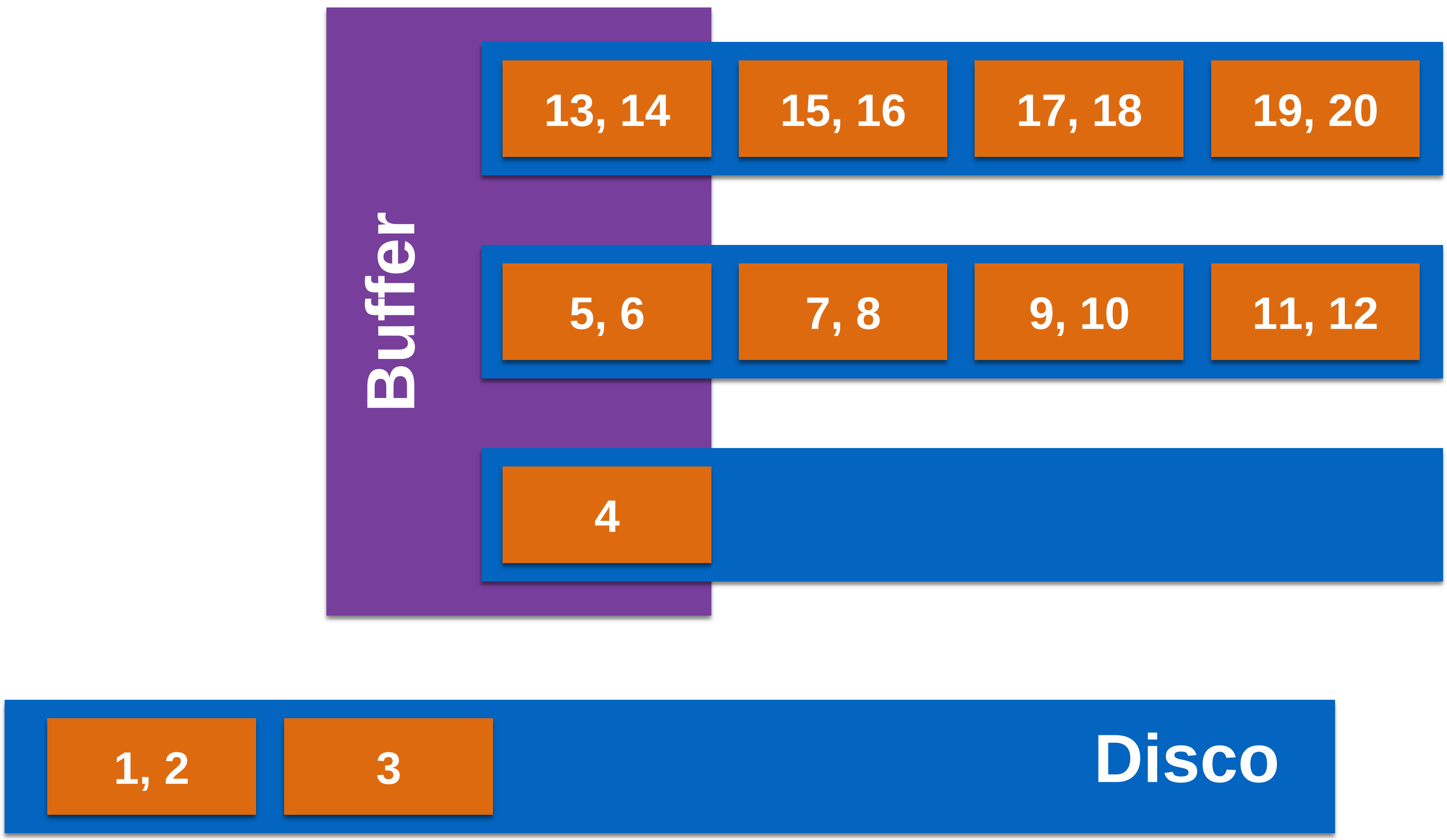
Se van escribiendo las páginas:



External Merge Sort

Optimizado

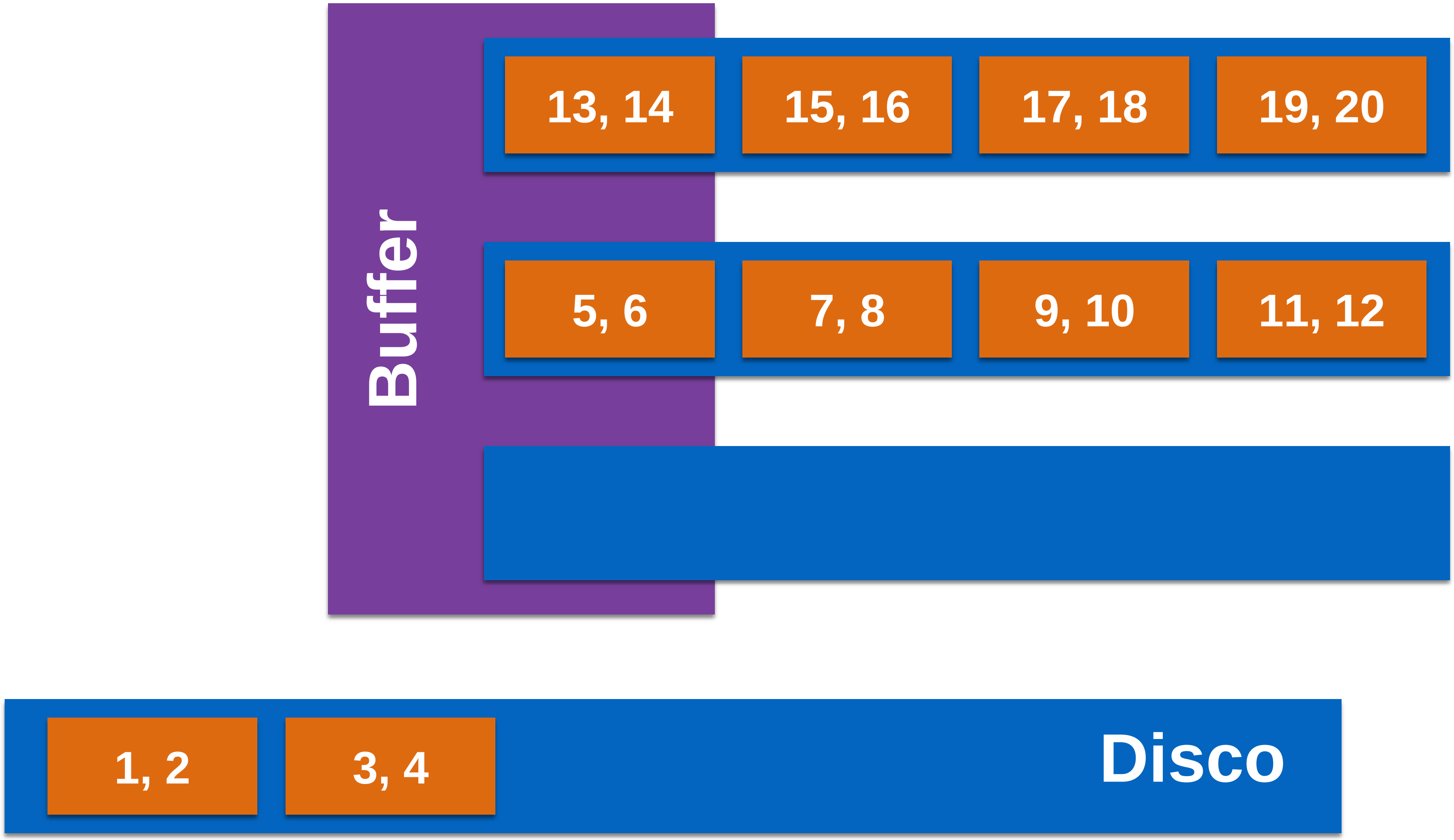
Se van escribiendo las páginas:



External Merge Sort

Optimizado

Se van escribiendo las páginas:



External Merge Sort

Optimizado

Sea **N** número de páginas del archivo y un buffer de tamaño B + 1

Número de runs iniciales:

$$\left\lceil \frac{N}{B+1} \right\rceil$$

Número de fases:

$$\underbrace{1}_{\text{Fase 0}} + \underbrace{\left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil}_{\textit{Fases}}$$

External Merge Sort

Optimizado

Sea **N** número de páginas del archivo y un buffer de tamaño B + 1

Costo en I/O:

$$2 \cdot N \cdot \left(1 + \left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil\right)$$

External Merge Sort

Optimizado

Si tenemos una tabla de 8 GB y páginas de 8 KB ~ 1048576 páginas en total

Memoria RAM para el buffer es de 2 GB, por lo que B + 1 ~ 262145 páginas

Si cada I/O toma 0.1 ms, ordenar tarda **6.7 minutos**

External Merge Sort

Optimizado

Lo más rápido es ejecutar el algoritmo en 2 fases

$$2 = 1 + \left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil$$

$$\Downarrow \quad \text{despejamos B}$$

$$B \geq \sqrt{N}$$

Si suponemos una tabla de 10^9 páginas (60 TB), sólo necesitamos **240 MB de buffer**!

External Merge Sort

Optimizado

Podemos plantear que si la tabla es de **N** páginas, el costo en I/O de ordenarla es **4N**

¿Qué pasa si no escribimos el último output? (pipeline)

Para más detalles tomar IIC3413 - Implementación de sistemas de Bases de Datos