Bases de Datos

Clase 8: Evaluación de consultas

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**





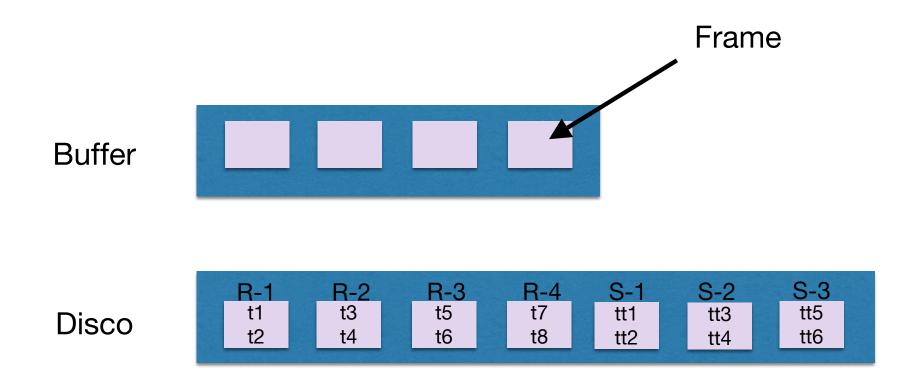


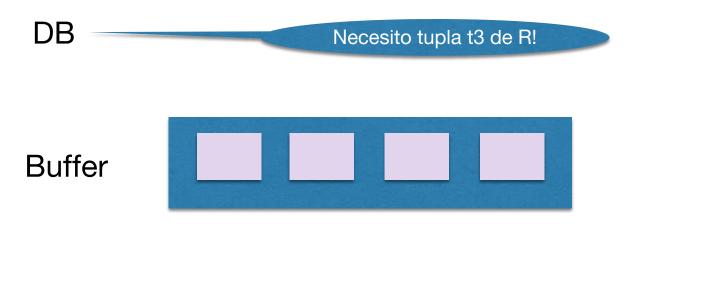




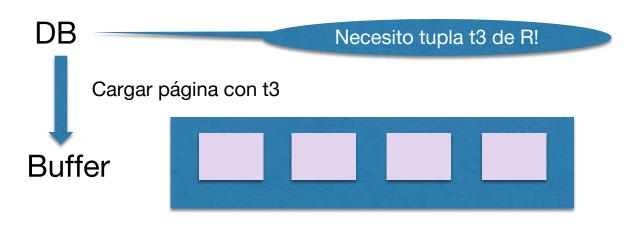
Buffer

R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	t5 t6	t7	tt1 tt2	tt3 tt4	tt5 tt6
t2	t4	t6	t8	tt2	tt4	tt6
				CONTRACT ON	CALL STATE OF THE PARTY	

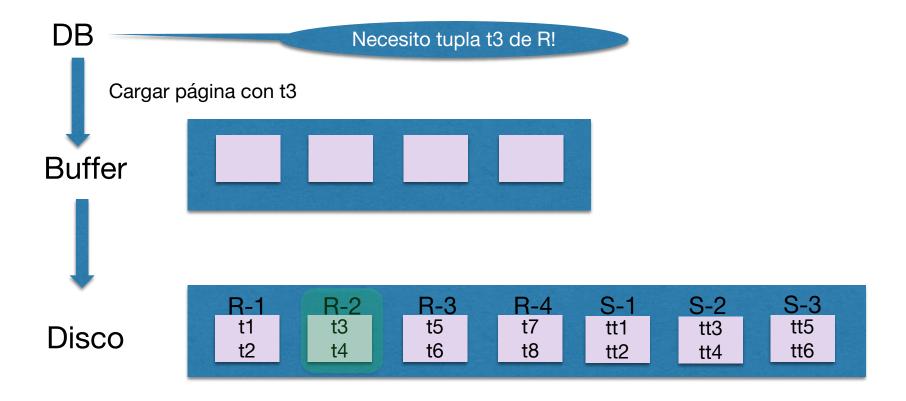


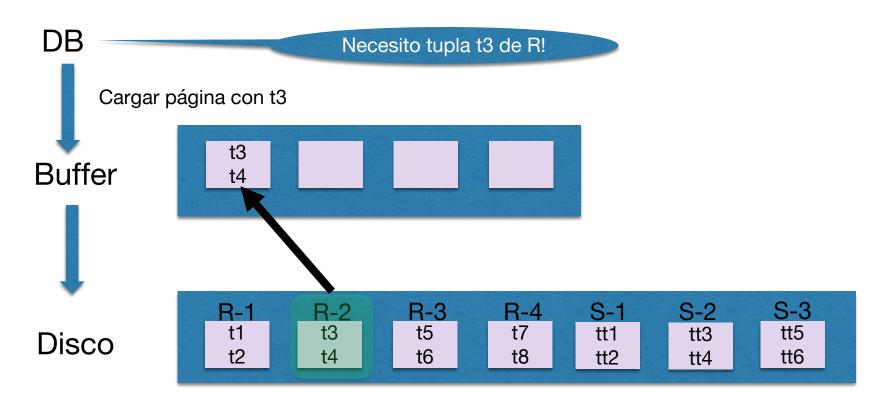


R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	t5	t7	tt1	tt3	S-3 tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt6
				CONTRACT ON	CALL SERVICE VIEW	





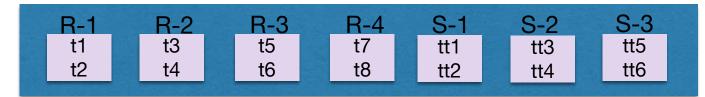


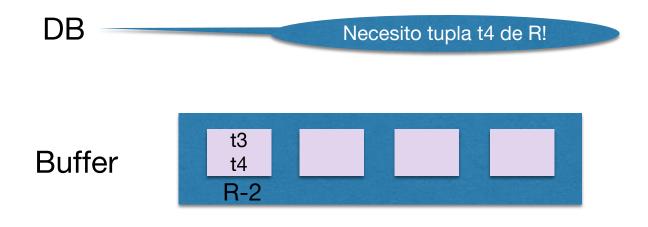


DB

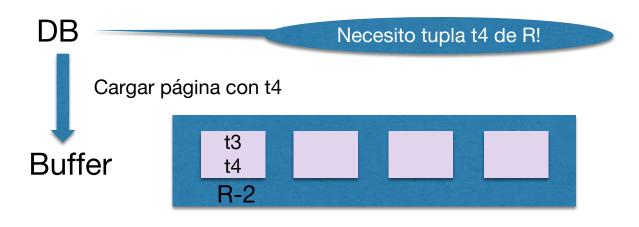
Buffer



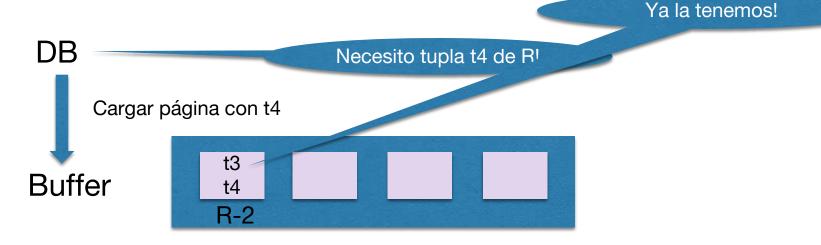




R-1	R-2	R-3	R-4	S-1	S-2	S-3 tt5
t1	t3	t5	t7 t8	tt1	tt3	tt5
t2	t4	t6	t8	tt2	tt3 tt4	tt6
				OF PERSONS ASSESSED.	CAMPINE TO SERVICE	







R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	t5	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt4	tt6
				CONTRACTOR OF THE PARTY OF THE	NAME OF TAXABLE PARTY.	

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

Implementan interfaz de un iterador lineal:

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

Algoritmos en una BD

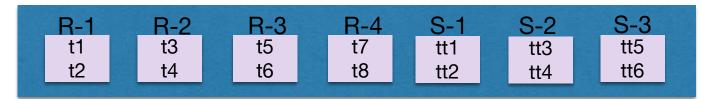
Para una relación R:

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

DB

Buffer





DB R.open()

Buffer

S-3 R-3 S-2 R-4 S-1 tt5 t1 t3 t5 t7 tt1 tt3 Disco t4 t6 tt6 tt2 tt4

DB R.open()

Buffer t1 t2 R-1

S-3 R-3 S-2 R-4 S-1 tt5 t1 t3 t5 t7 tt1 tt3 Disco t4 t6 tt6 tt2 tt4

DB R.next()

Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	R-3 t5	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt5 tt6
				MATERIAL DE	CALL SECTION	

DB R.next()

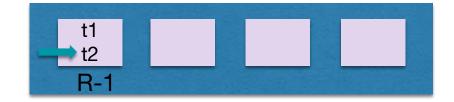
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	R-3 t5	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt5 tt6
				MATERIAL DE	CALL SECTION	

DB R.next()

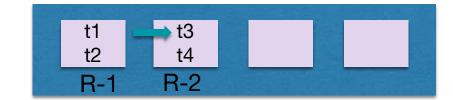
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	t5	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt2	tt4	tt5 tt6
				No. of Lot	(MEETING	

DB R.next()

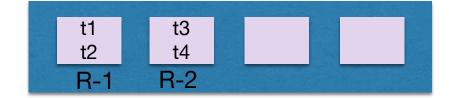
Buffer





DB R.close()

Buffer





DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

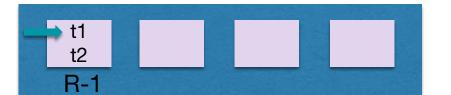
t:= R.next()
while t != null do
output t
t:= R.next()

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

output t
t:= R.next()

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

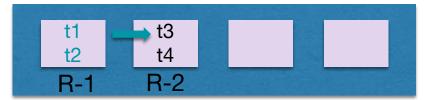
R.close()



R.open() DB t := R.next()while t != null do output t **t**:= **R.next()** Buffer R.close() S-3 S-2 t1 t3 tt5 tt1 tt3 Disco t2 t4 tt2 tt6 tt4

DB

Buffer



R.open()

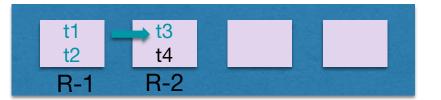
t:= R.next()
while t != null do
output t
t:= R.next()

R.close()



DB

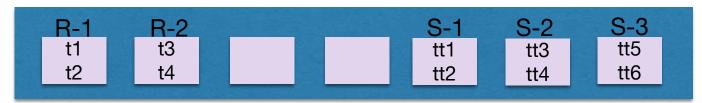
Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



DB

Buffer



R.open()

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

R.close()



En realidad

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

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

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
```

Sin indice

Si queremos hacer una selección sobre una tabla R

```
open()
   R.open()
                                                  Para recorrer la selección Sel = \sigma_{cond} (R)
next() // retorna el siguiente seleccionado
                                                  Sel.open()
   t = R.next()
   while t != null do
      if t satisface condición then
                                                  t := Sel.next()
          return t
                                                  while t != null do
      t:=R.next()
                                                            output t
   return null
                                                            t:= Sel.next()
close()
                                                  Sel.close()
   R.close()
```

Selección Sin índice

Necesariamente tenemos que recorrer todo R

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()
```

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

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)

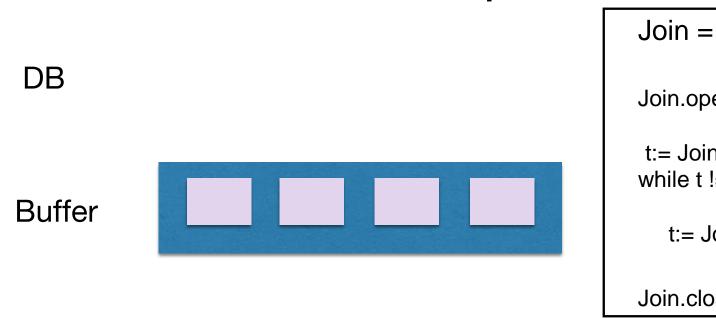
Queremos hacer un join entre **R** y **S**, cuando se satisface un predicado **p**

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

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

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
```



 $\mathsf{Join} = \mathsf{R} \bowtie_{\mathsf{p}} \! \mathsf{S}$ Join.open() t:= Join.next() while t != null do output t t:= Join.next() Join.close()

tt6

R-1 R-2 R-3 R-4 S-1 S-2 S-3 t1 t3 t5 t7 tt1 tt3 tt5

DB R⋈_pS ...Join.open()

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

Buffer





DB R⋈_pS ...Join.open()

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

Buffer





DB R⋈_pS ...Join.open()

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

Buffer

t1 tt1 tt2 R-1 S-1

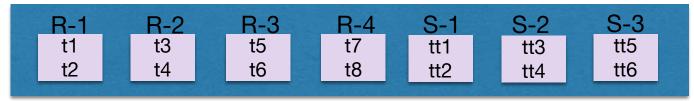


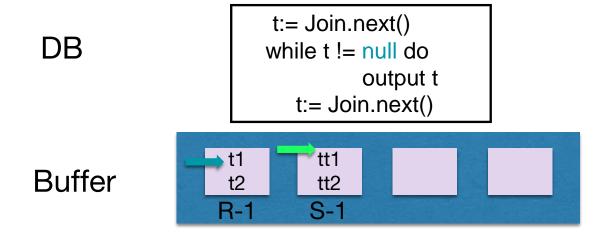
DB R⋈_pS ...Join.open()

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

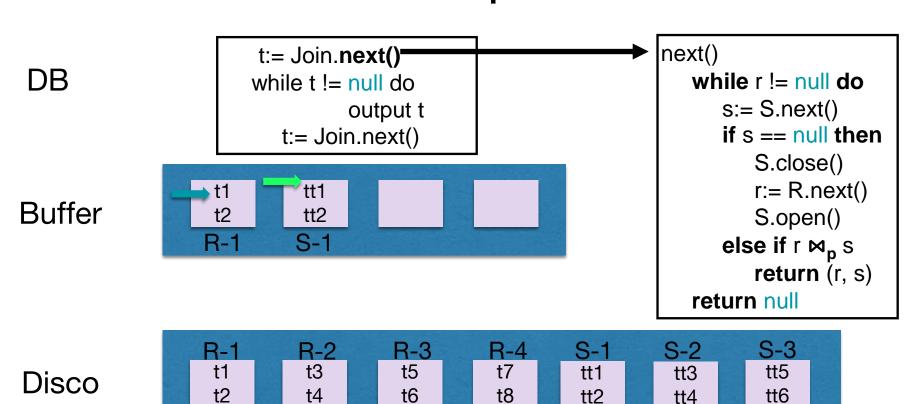
Buffer

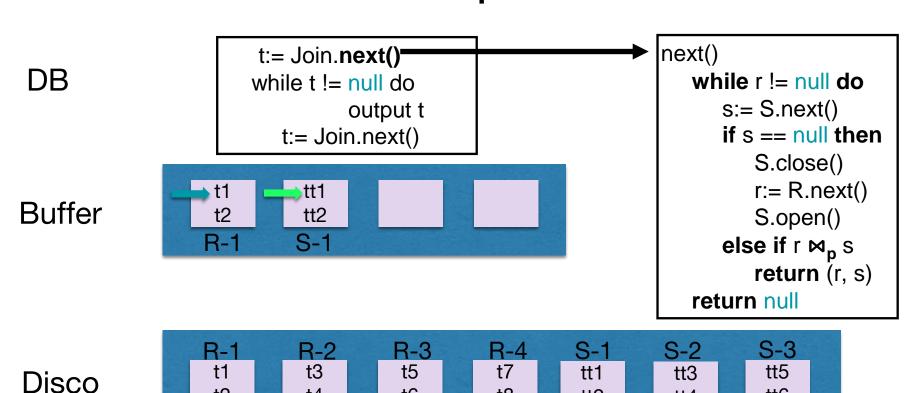
t1 t2 tt2 R-1 S-1





R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1 t2	t3 t4	t5 t6	t7 t8	tt1 tt2	tt3	S-3 tt5 tt6
t2	t4	t6	t8	tt2	tt4	tt6
				OF STREET	CALL STREET	





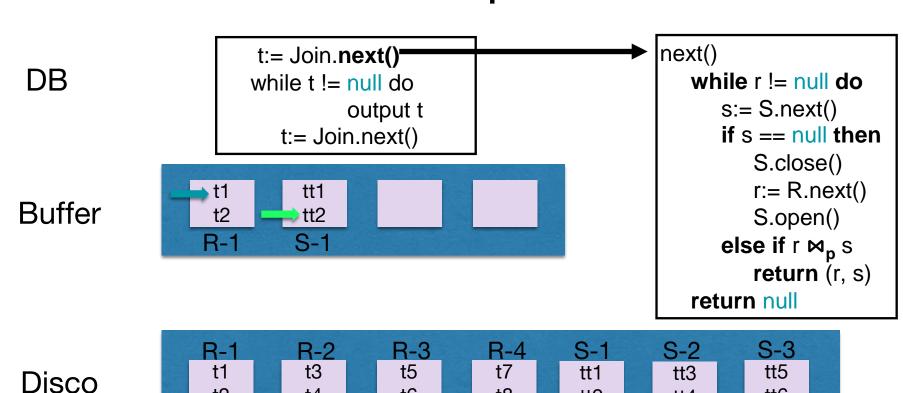
t8

tt2

tt6

tt4

t6



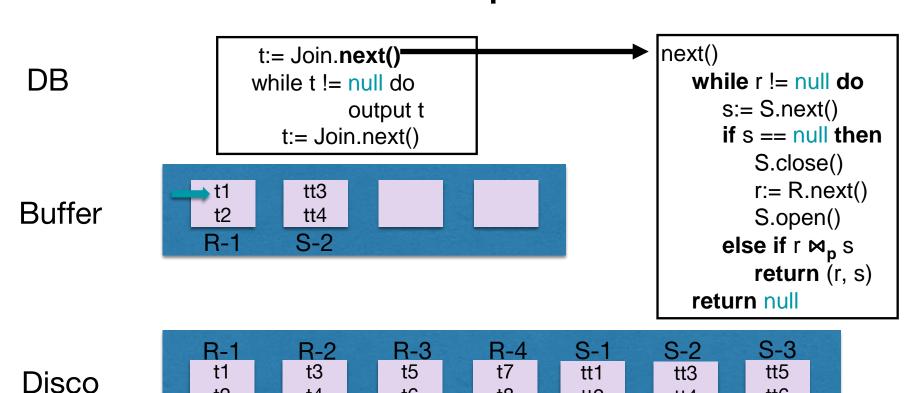
t8

tt2

tt6

tt4

t6



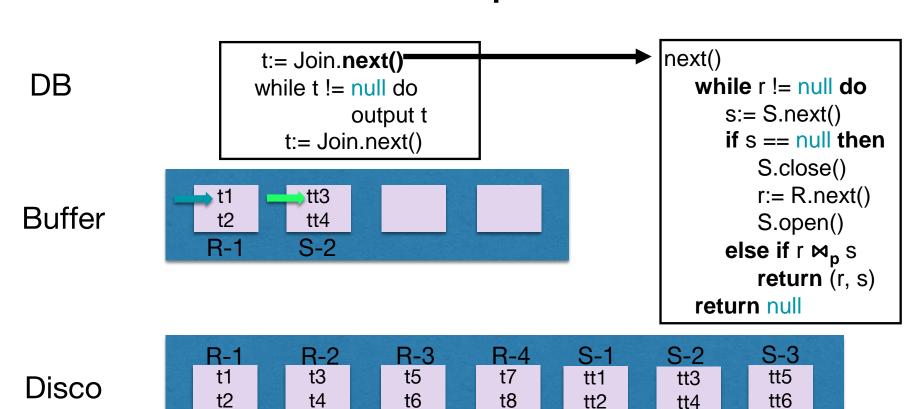
t8

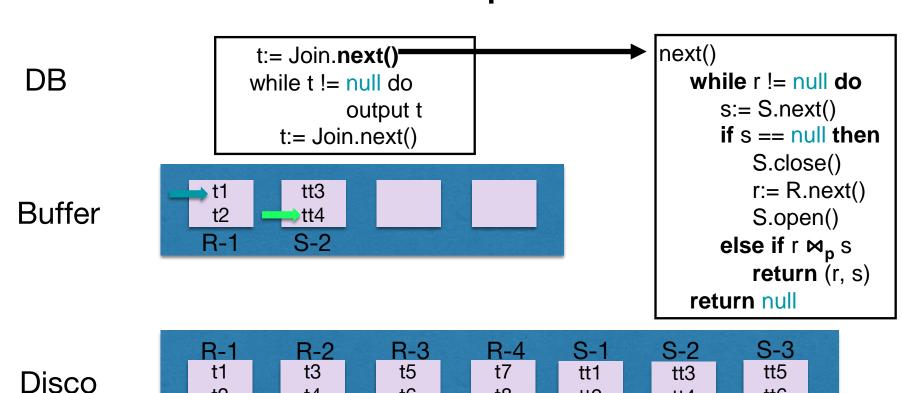
tt2

tt6

tt4

t6





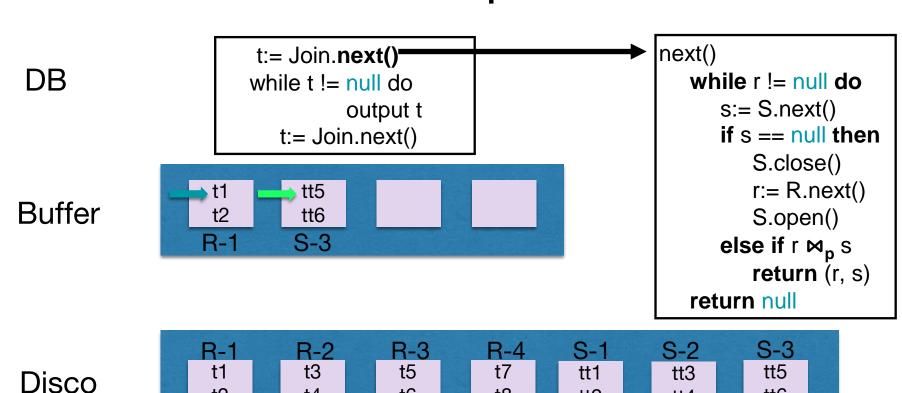
t8

tt2

tt6

tt4

t6



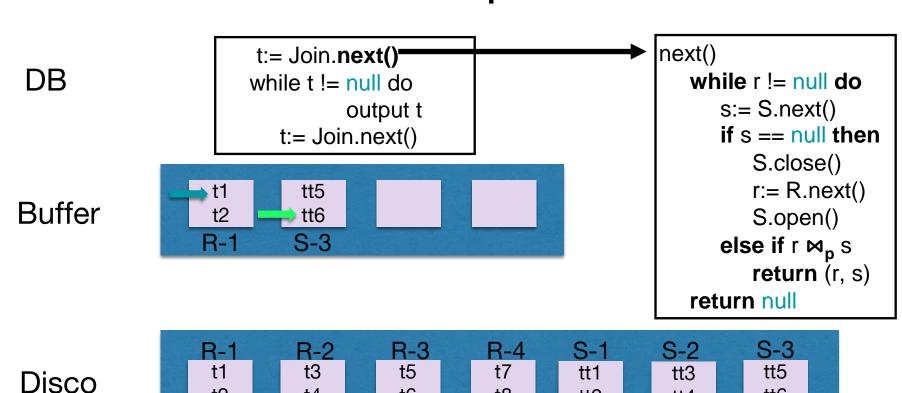
t8

tt2

tt6

tt4

t6



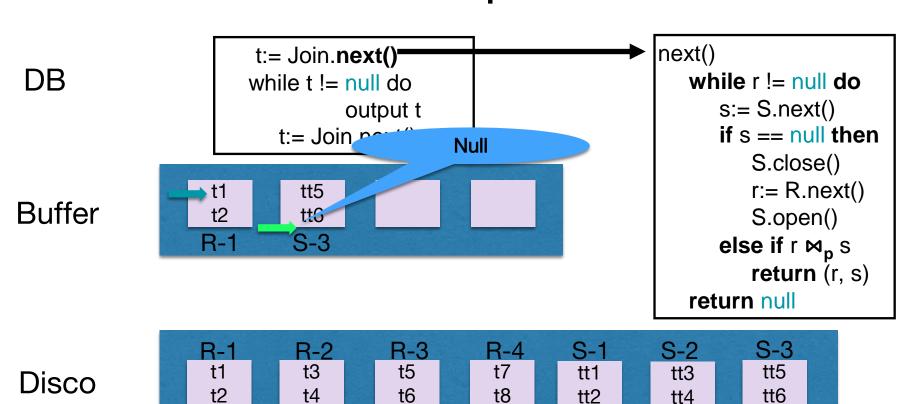
t8

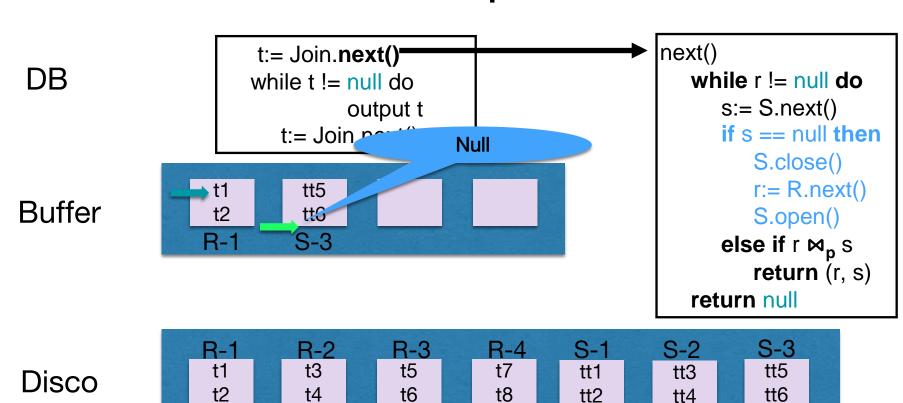
tt2

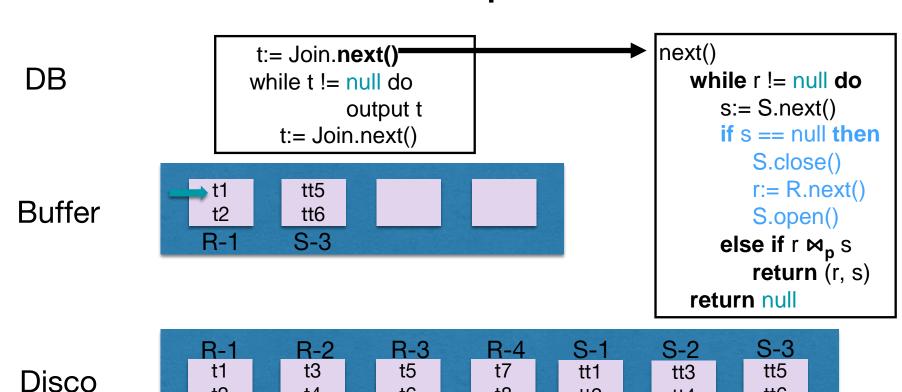
tt6

tt4

t6







t8

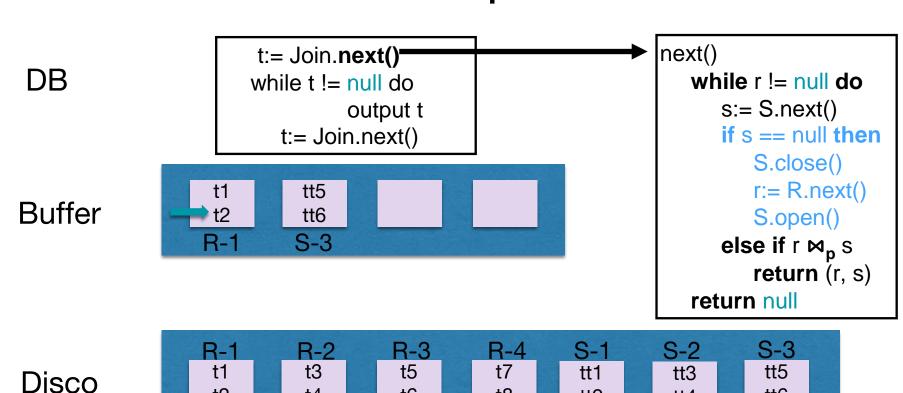
tt2

tt6

tt4

t6

t2



t8

tt2

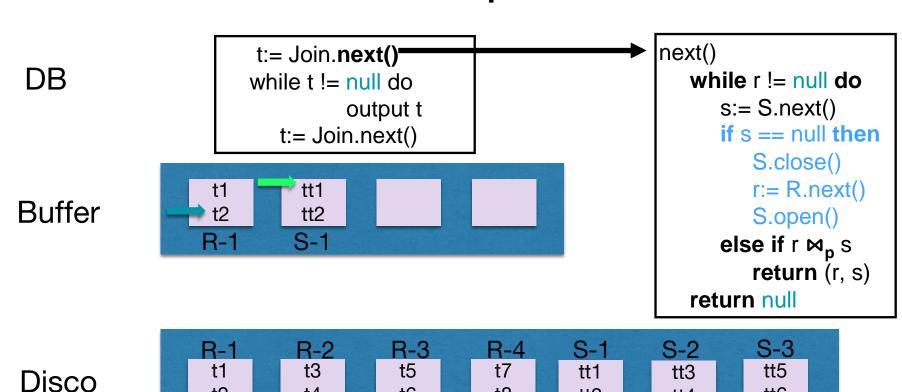
tt6

tt4

t6

t2

t4



t8

tt2

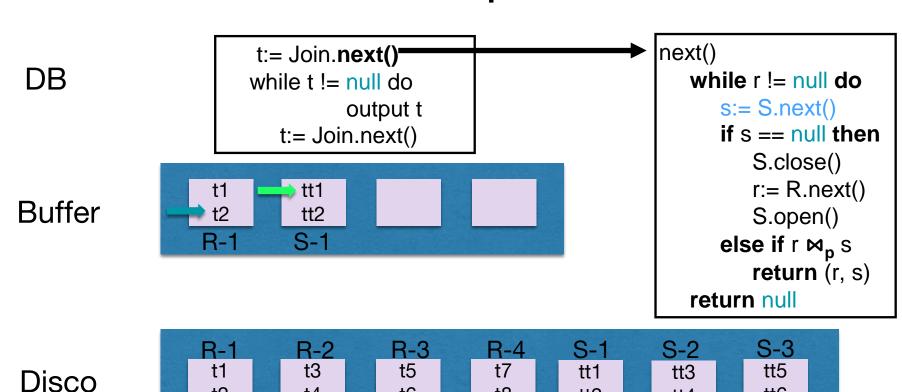
tt6

tt4

t6

t2

t4



t8

tt2

tt6

tt4

t6

t4

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:

Costo(R) + Tuplas(R)·Costo(S)

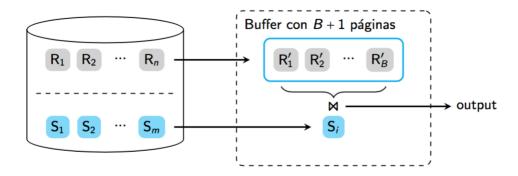
Si R y S son tablas de 16 MB, cada página es de 8 KB y las tuplas son de 300 bytes

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

Aprovechamos mejor el buffer



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

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

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
```

DB R⋈_pS

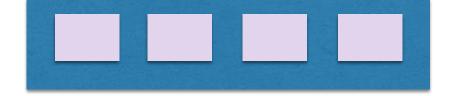
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	t5	t7 t8	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt6
STATE OF THE PARTY	STATE OF STREET	Service / Committee	NAME OF STREET	CONTRACTOR OF STREET	CHEVE IN	LONG HURSON

DB R⋈_pS.open()

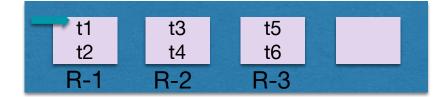
Buffer



R	-1	R-2	R-3	R-4	S-1	S-2	S-3
t	1	t3 t4	t5 t6	t7	tt1	tt3	tt5 tt6
t	2	t4	t6	t8	tt2	tt3 tt4	tt6
		L'T	10	to	IIZ	114	- 110

DB R⋈_pS.open()

Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
				CONTROL OF THE	CALL STATE OF STREET	

DB R⋈_pS.open()

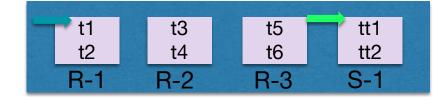
Buffer





DB while (R⋈_pS.next())

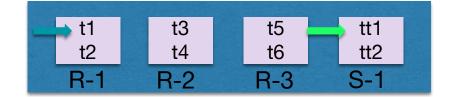
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
				CONTRACT OF	CHECKET TO	

DB while (R⋈_pS.next())

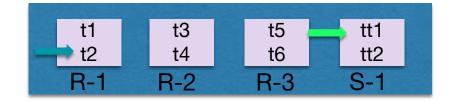
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
				CONTROL OF THE	CALL STATE OF STREET	

DB while (R⋈_pS.next())

Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	R-3 t5 t6	t7	tt1	tt3 tt4	tt5 tt6
t2	t4	t6	t8	tt2	tt4	tt6
					CONTRACTOR	

DB while (R⋈_pS.next())

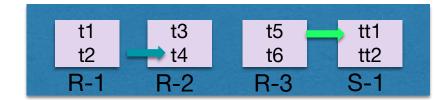
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt6
				With Street		

DB while (R⋈_pS.next())

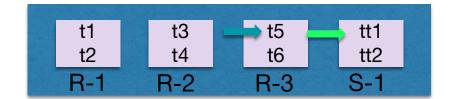
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1 tt2	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
	ART DESCRIPTION		ALCOHOL: N			Total School

DB while (R⋈_pS.next())

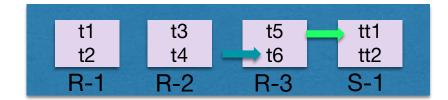
Buffer



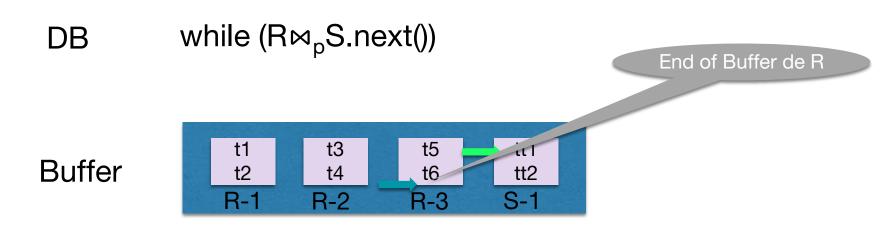
R-1	R-2	R-3	R-4	S-1	S-2		S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3 tt4		tt5
t2	t4	t6	t8	tt2	tt4	B	tt6
					CHICAGO IV		

DB while (R⋈_pS.next())

Buffer



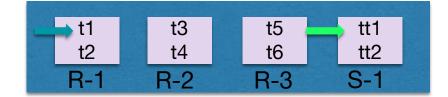
R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt6
CASHING!	STATE OF THE PARTY.	SCHOOL STATE OF	ACCRECATION OF	CHACKE	CALL STREET	THE RESERVE





DB while (R⋈_pS.next())

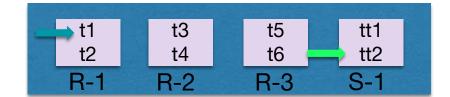
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1 tt2	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
	ART DESCRIPTION		ALCOHOL: N			Total School

DB while (R⋈_pS.next())

Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
				STATE OF THE	CALL SERVICE VIEW	

DB while (R⋈_pS.next())

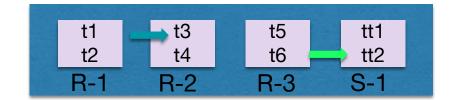
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3	tt5
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt6
CASHING!	STATE OF THE PARTY.	SCHOOL STATE OF	ACCRECATION OF	CHACKE	CALL STREET	THE RESERVE

DB while (R⋈_pS.next())

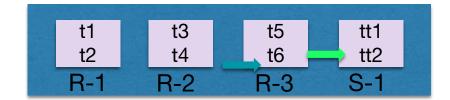
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
THE RESERVE OF THE PERSON NAMED IN	DESCRIPTION OF		Name and Address of the Owner, where	ASSESSMENT OF REAL PROPERTY.	NAME OF TAXABLE PARTY.	THE RESERVE

DB while (R⋈_pS.next())

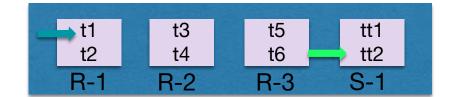
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
THE RESERVE OF THE PERSON NAMED IN	DESCRIPTION OF		Name and Address of the Owner, where	ASSESSMENT OF REAL PROPERTY.	NAME OF TAXABLE PARTY.	THE RESERVE

DB while (R⋈_pS.next())

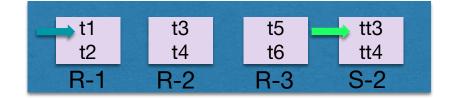
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	R-3 t5 t6	t7	tt1 tt2	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
DASHARDI	ACT DISCOVERY	NO VIEW COM	ALTERNATION OF THE PARTY NAMED IN	of the least one	o had signed and the same	-

DB while (R⋈_pS.next())

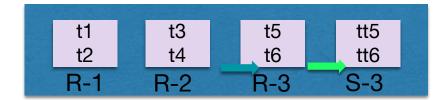
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3	
t1 t2	t3	t5	t7 t8	tt1	tt3	tt5	
t2	R-2 t3 t4	t6	t8	tt1 tt2	tt4	tt5 tt6	
				OL COMPANIES	CHECKE DE		

DB while (R⋈_pS.next())

Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	t5 t6	t7	tt1	tt3	tt5 tt6
t2	t4	t6	t8	tt1 tt2	tt3 tt4	tt6
				No. of Lot		

DB while (R⋈_pS.next())

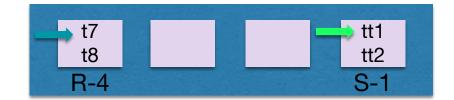
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3 t4	t5 t6	t7	tt1	tt3 tt4	tt5
t2	t4	t6	t8	tt2	tt4	tt6
				CONTRACT OF	CHECKET TO	

DB while (R⋈_pS.next())

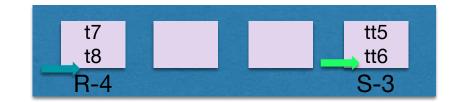
Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3
t1	t3	R-3 t5 t6	t7	tt1	tt3 tt4	tt5 tt6
t2	t4	t6	t8	tt2	tt4	tt6
					CONTRACTOR	

DB while (R⋈_pS.next())

Buffer



R-1	R-2	R-3	R-4	S-1	S-2	S-3	
t1	t3	t5	t7	tt1	tt3	tt5	
t2	t4	t6	t8	tt2	tt4	tt5 tt6	

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:

Costo(R) + (Páginas(R)/Buffer)·Costo(S)

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

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

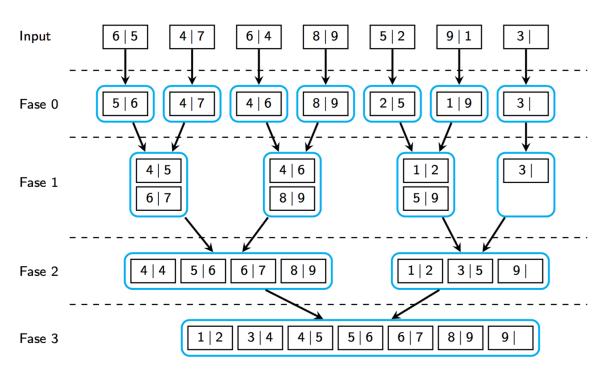
External Merge Sort

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



Implementación de sistemas de Bases de Datos

Para más detalles tomar IIC3413 -