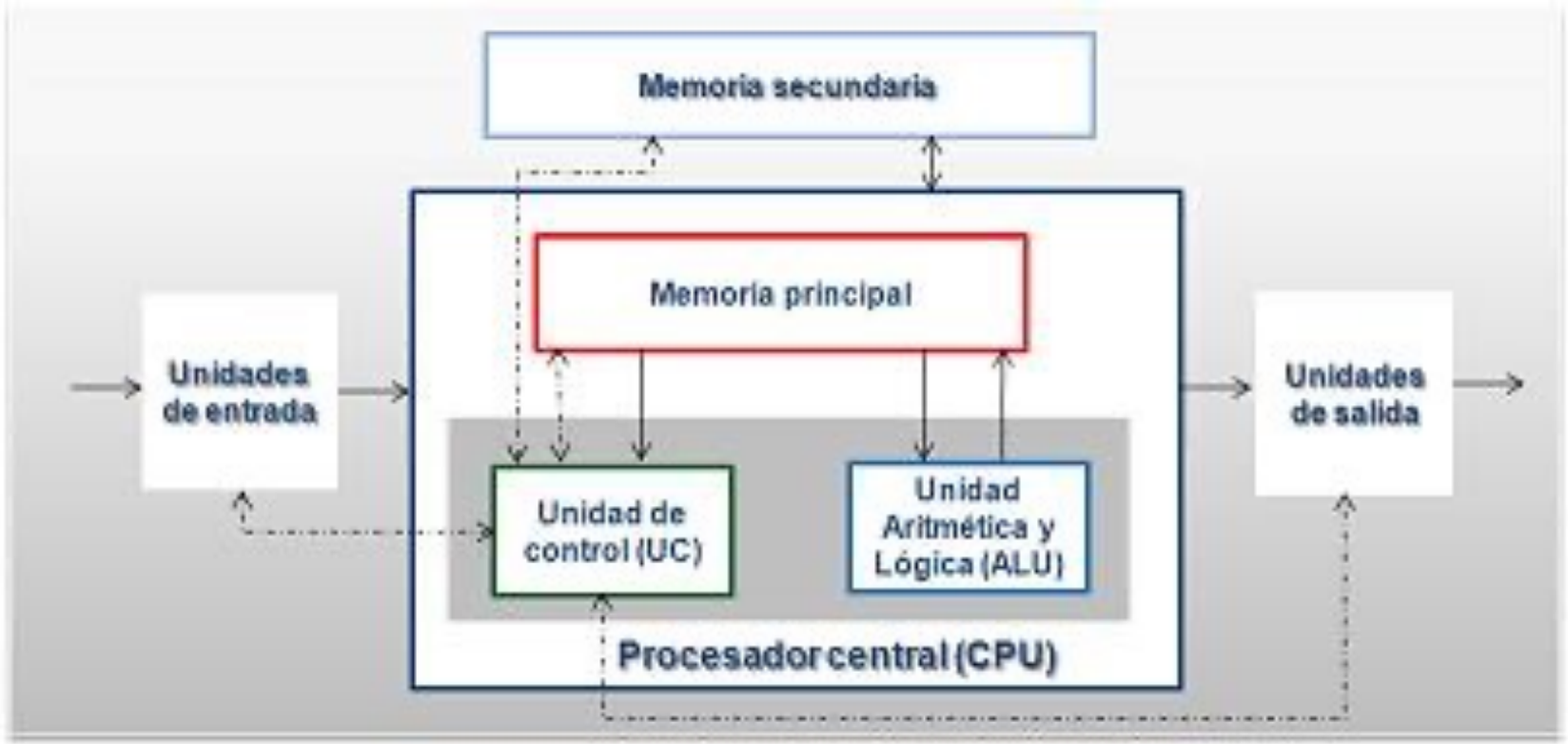
Bases de Datos

Clase 10: Arquitectura de Computadores y S.O, y Recuperación de Fallas

Veamos un poco de Arquitectura de computadores y Sistemas Operativos



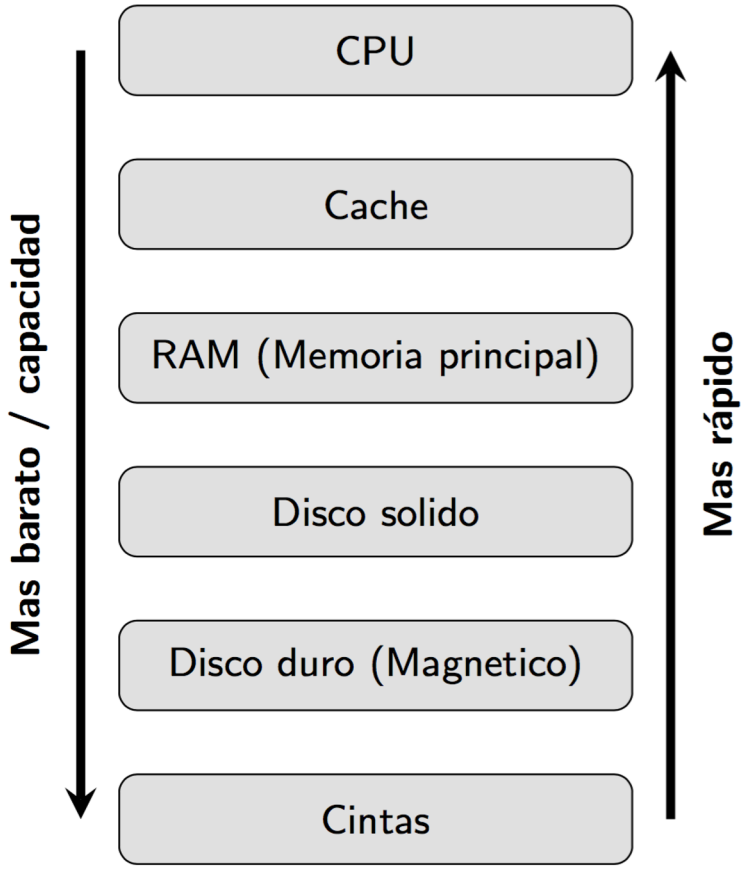
Almacenamiento



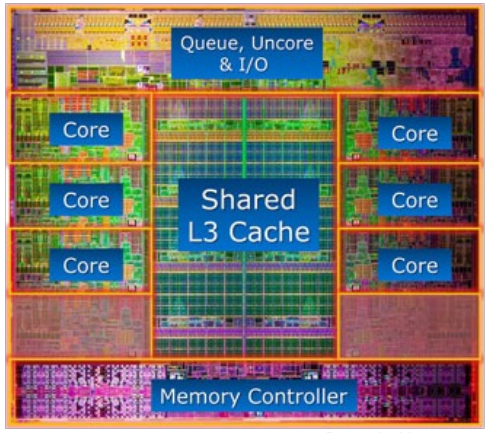




Jerarquía de Memoria



Intel Core i7-3960X Processor Die Detail



ComputerHope.com

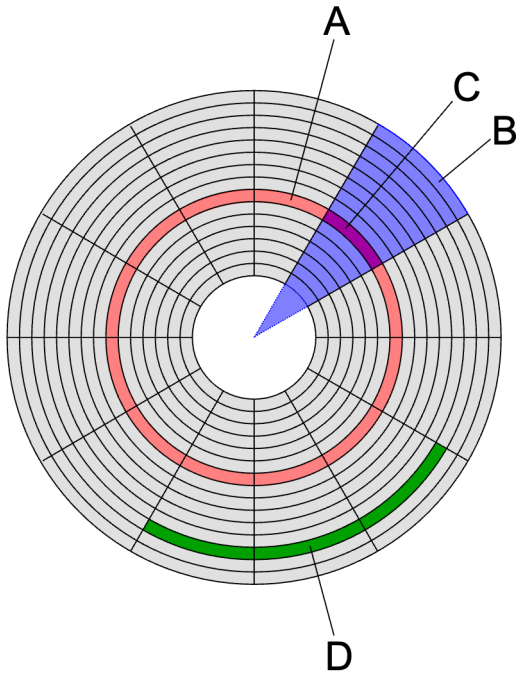


Disco Duro

Sector: unidad física mínima de almacenamiento para el Disco (definido por hardware)

Página: unidad lógica mínima de almacenamiento para el Sistema de Archivos (definido por OS/DBMS)

Estructura de disco magnético:
A es una **pista del disco** (roja),
B es un sector geométrico (azul),
C es un sector de una pista (magenta),
D es un grupo de sectores o clúster (verde).



¡Un DBMS trabaja al nivel de página!

Implicancias en rendimiento al usar un disco Duro

1. Operación de la DBMS: Los datos deben estar en memoria para que la DBMS los procese.

2. Transferencia de Datos: La unidad de transferencia entre el disco y la memoria principal es un bloque. Si se necesita un elemento del bloque, se transfiere el bloque completo.

3. Tiempo de Operación de I/O: El tiempo para leer o escribir un bloque varía según la ubicación de los datos.

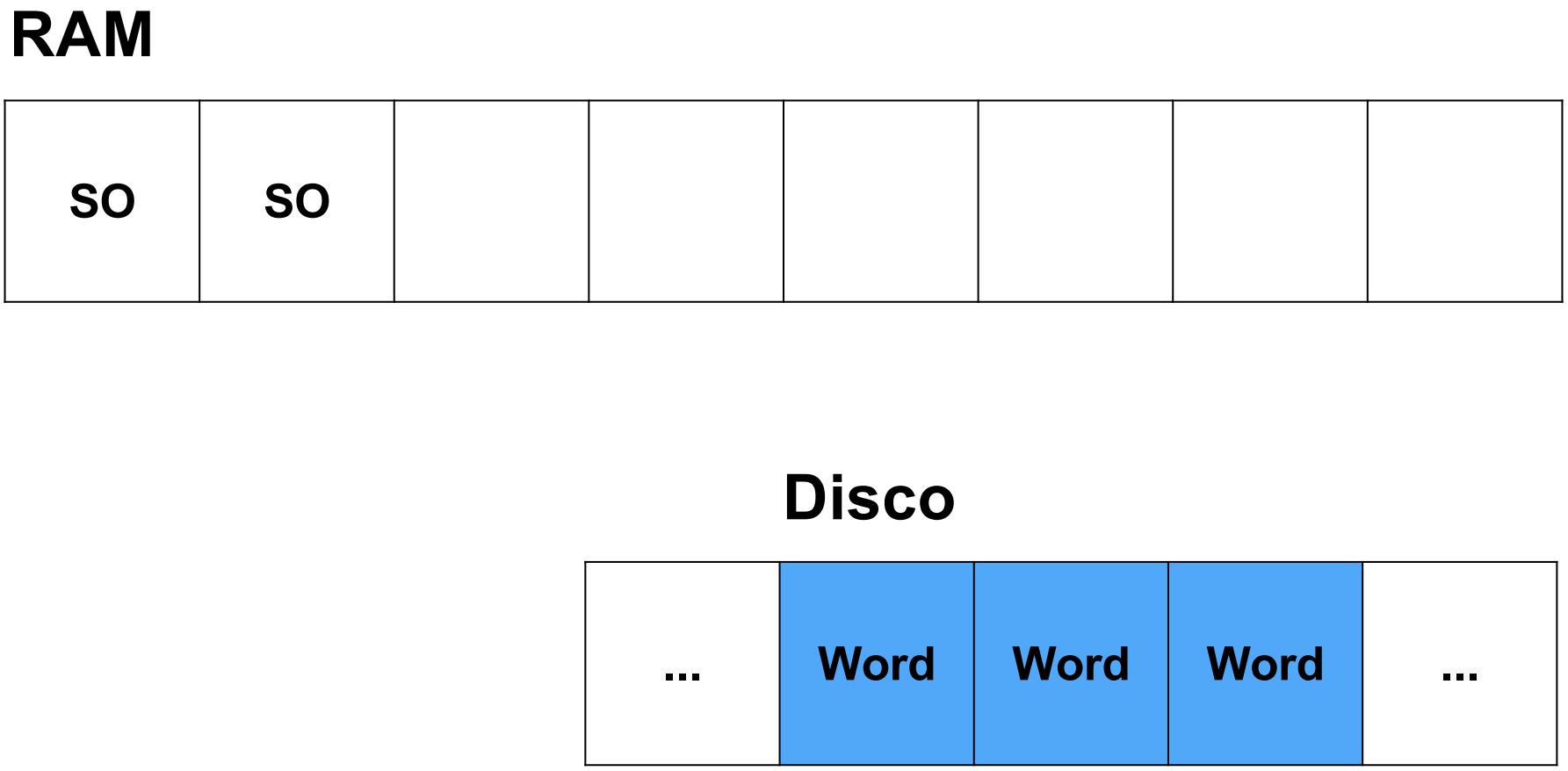
Tiempo de acceso = tiempo de búsqueda + retraso rotacional + tiempo de transferencia.

Disco y RAM

RAM

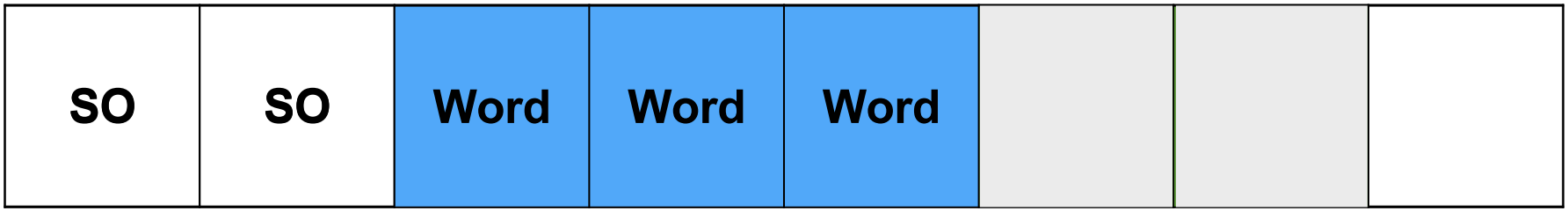
SO	SO						

Disco y RAM

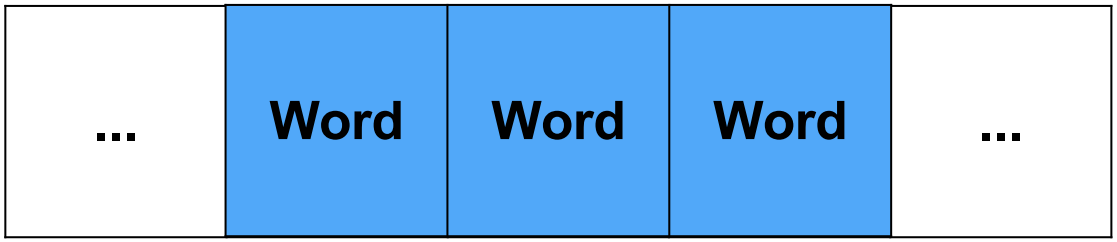


Disco y RAM

RAM

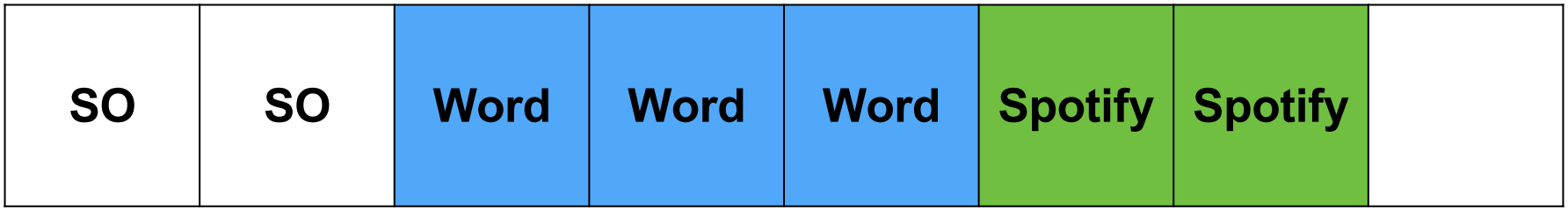


Disco



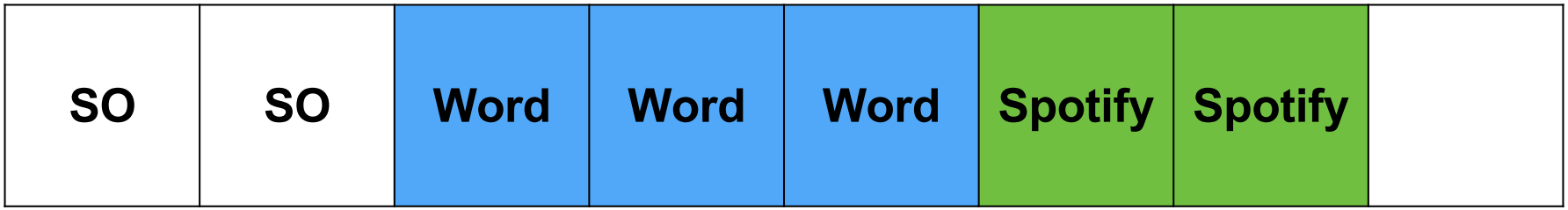
Disco y RAM

RAM

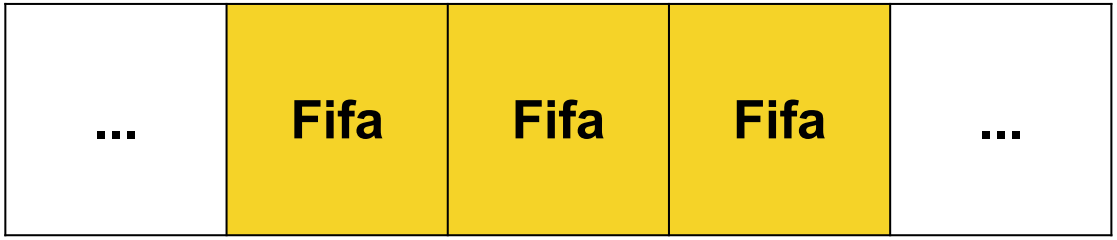


Disco y RAM

RAM

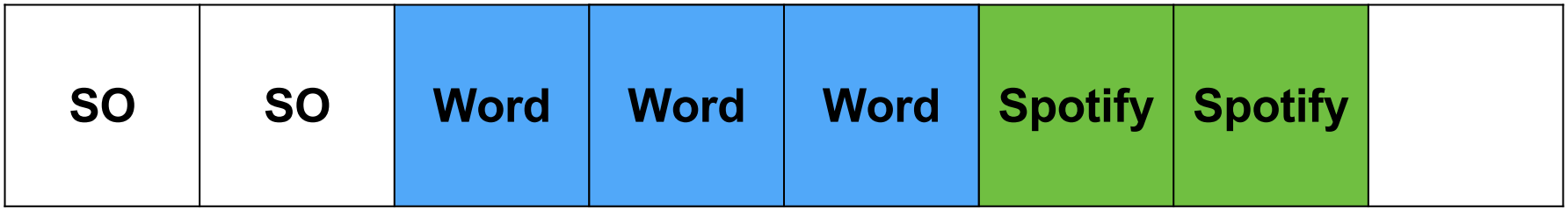


Disco



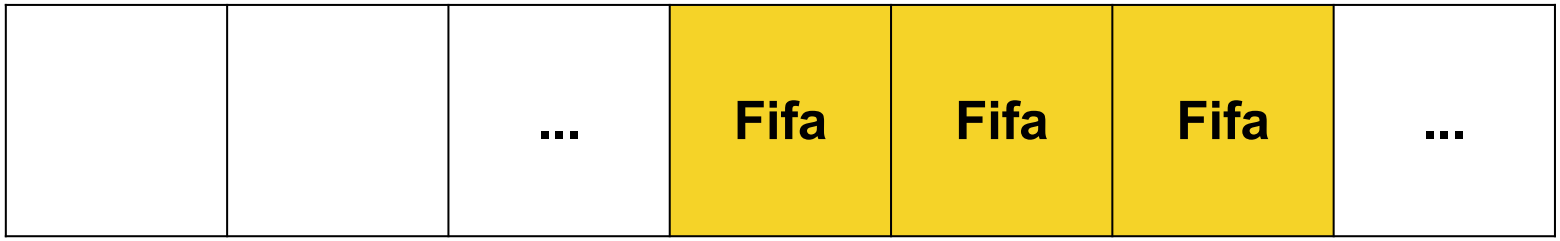
Disco y RAM

RAM



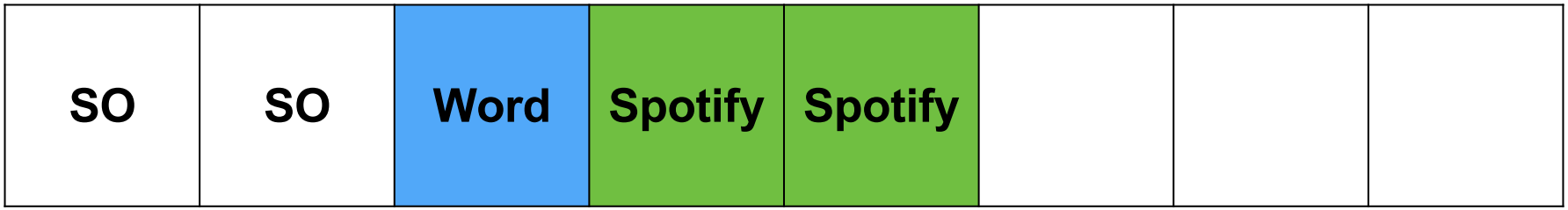
Disco Duro (espacio de swap)

Disco



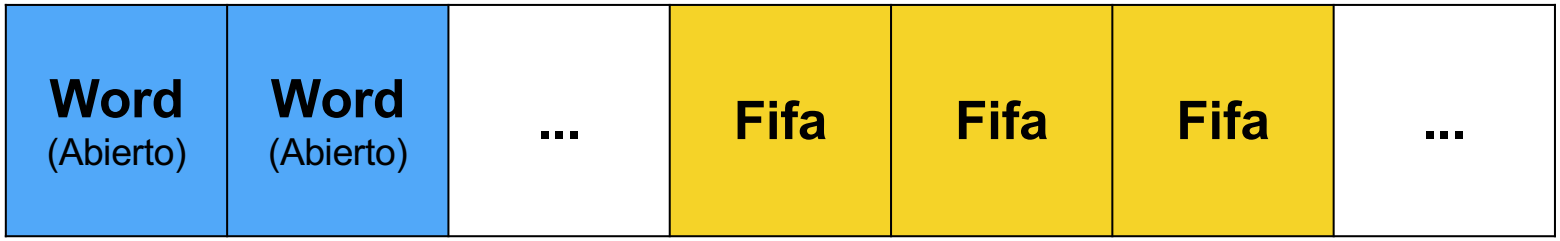
Disco y RAM

RAM



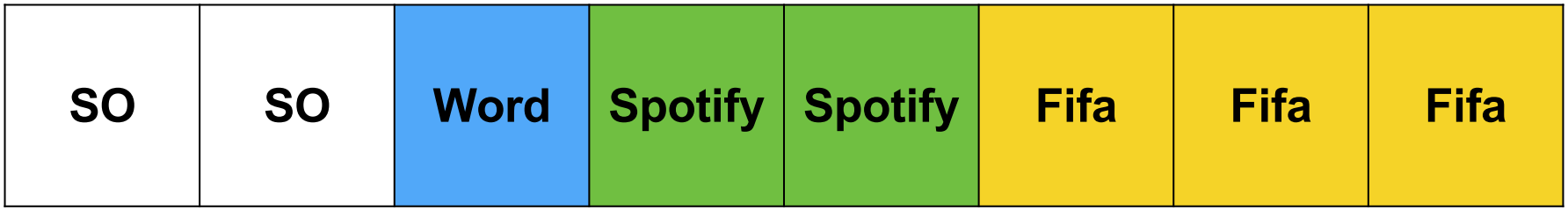
Disco Duro (espacio de swap)

Disco Duro



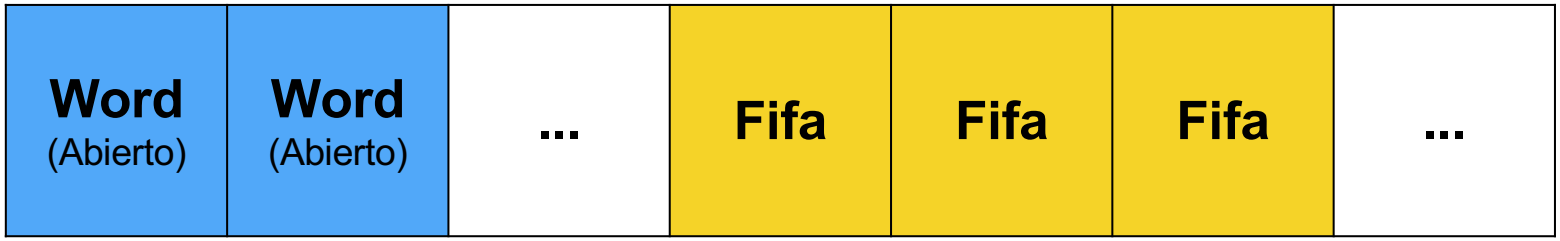
Disco y RAM

RAM



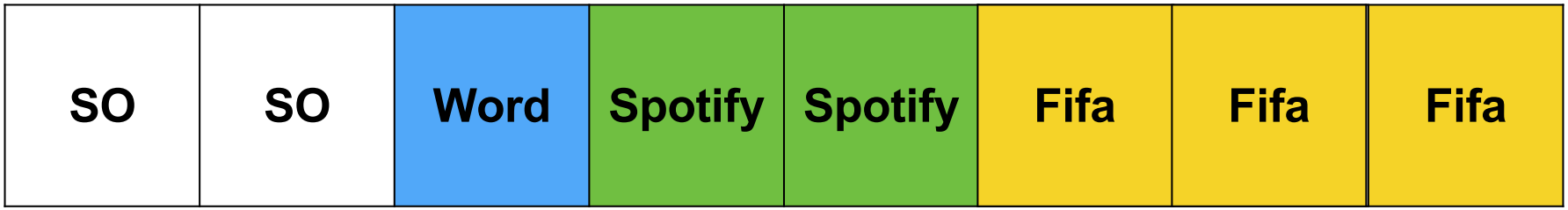
Disco Duro (espacio de swap)

Disco Duro

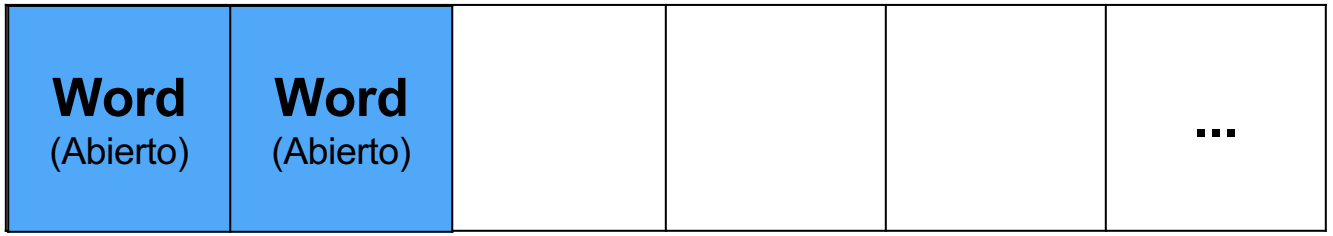


Disco y RAM

RAM



Disco Duro (espacio de swap)



Disco y DBMS

Los records de las bases de datos se almacenan en **páginas** de disco.

A medida que se hace necesario, las páginas son traídas a memoria principal (**buffer**)

Páginas, disco y buffer

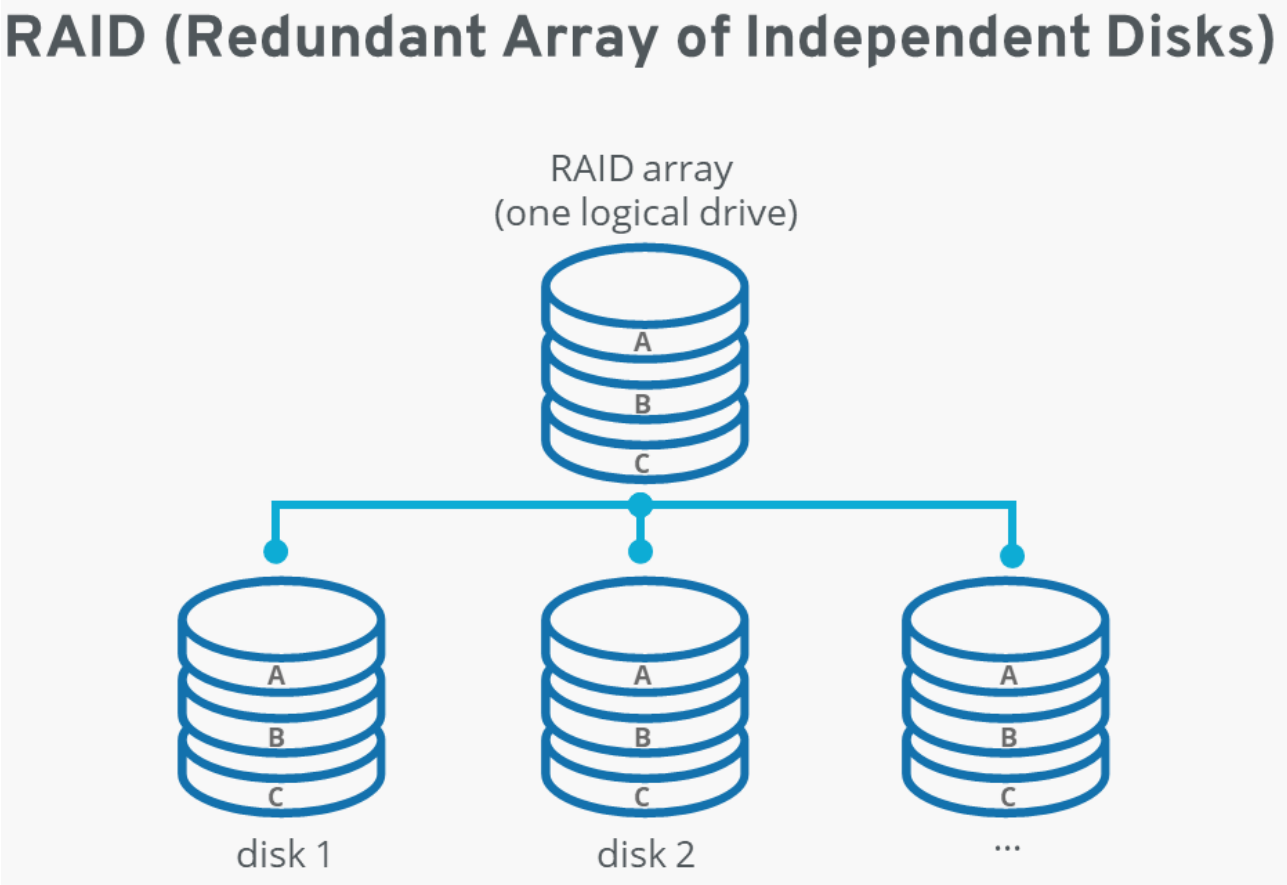
Para trabajar con las tuplas de una relación, la base de datos carga la página con la tupla desde el disco

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

RAID (Matriz redundante de discos independientes)

RAID

Objetivo del RAID: Incrementar el rendimiento y la fiabilidad de sistemas de almacenamiento a través de un arreglo de múltiples discos que operan como uno solo.



RAID - Tecnicas

Técnicas Principales de un RAID:

1. Segmentación de Datos (Data Striping): Los datos se dividen en bloques (unidades de segmentación) distribuidos en varios discos.

2. Redundancia: Uso de discos adicionales para permitir la reconstrucción de datos en caso de fallos.

RAID - Niveles

Niveles de RAID:

- **Nivel 0:** Sin redundancia, solo segmentación.
- **Nivel 1:** Espejado, cada disco tiene una copia idéntica.
- **Nivel 0+1:** Combinación de segmentación y espejado.
- **Nivel 3:** Paridad entrelazada por bit.
- **Nivel 4:** Paridad entrelazada por bloque.
- **Nivel 5:** Paridad distribuida, similar al Nivel 4 pero con bloques de paridad distribuidos entre todos los discos.

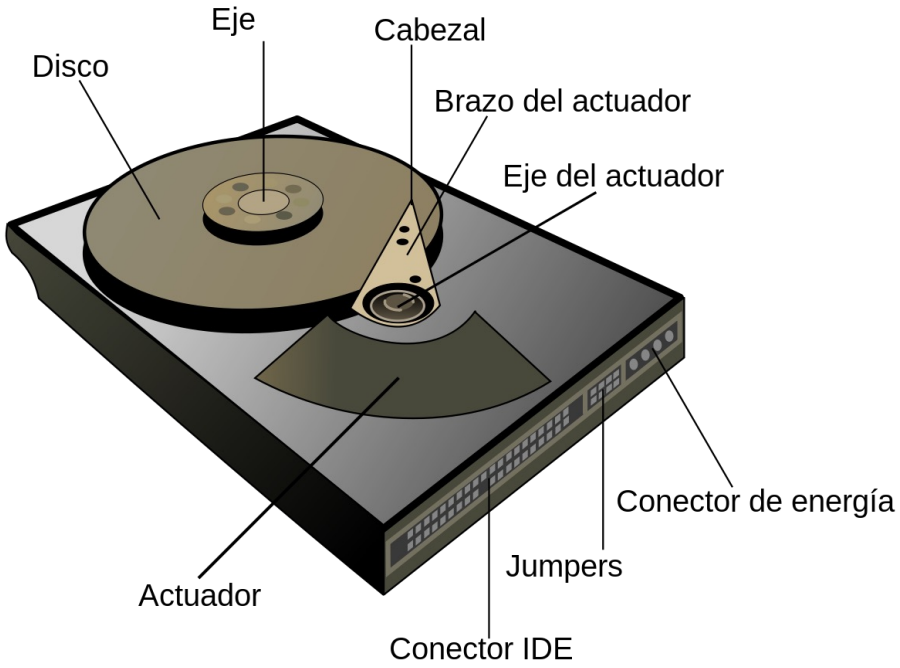
Disco Solido

Celda: Unidad mínima que puede almacenar bits (definida por el tipo de SSD, como SLC, MLC, TLC, QLC)

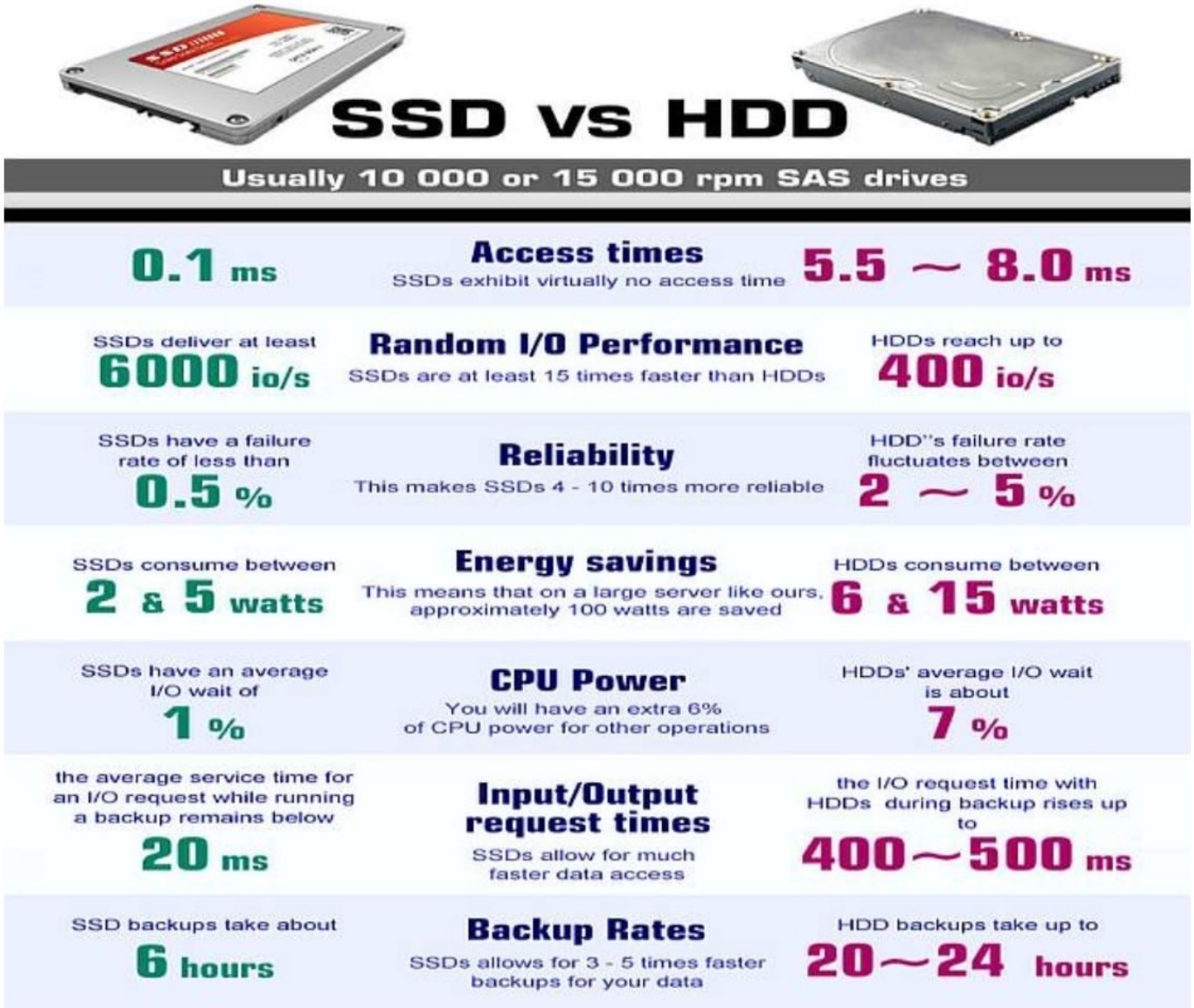
Bloque: Conjunto de páginas que es la unidad mínima de borrado

Plano: Conjunto de bloques gestionados como una unidad

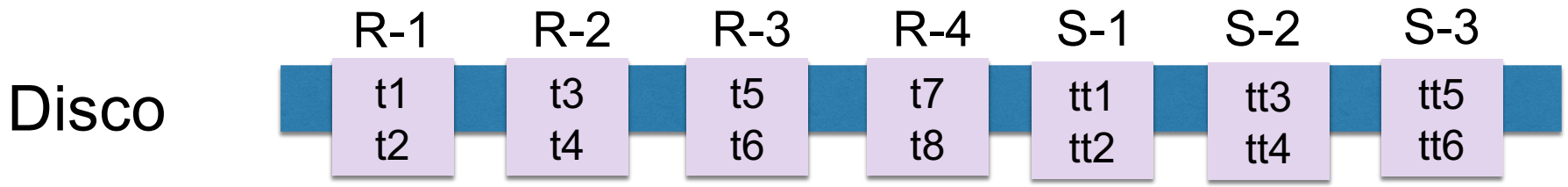
IDE: Un chip de memoria en el SSD que puede contener múltiples planos



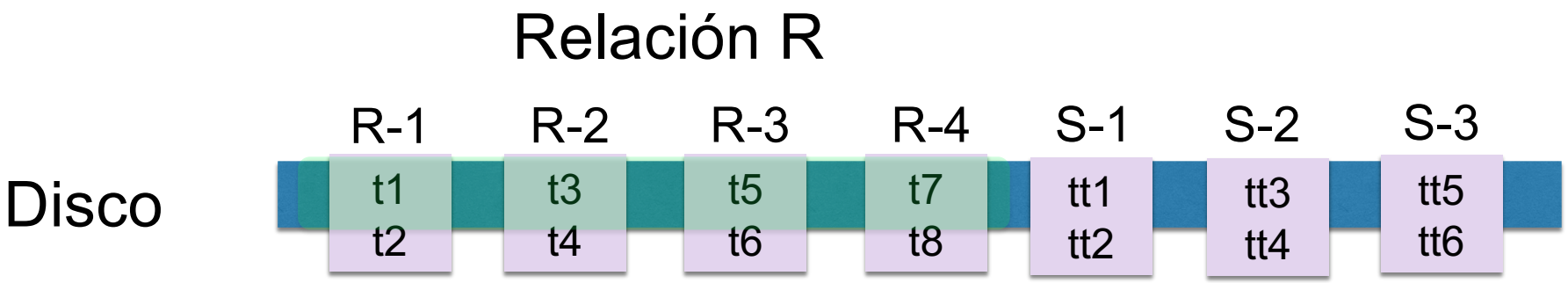
Disco Duro vs Disco Solido



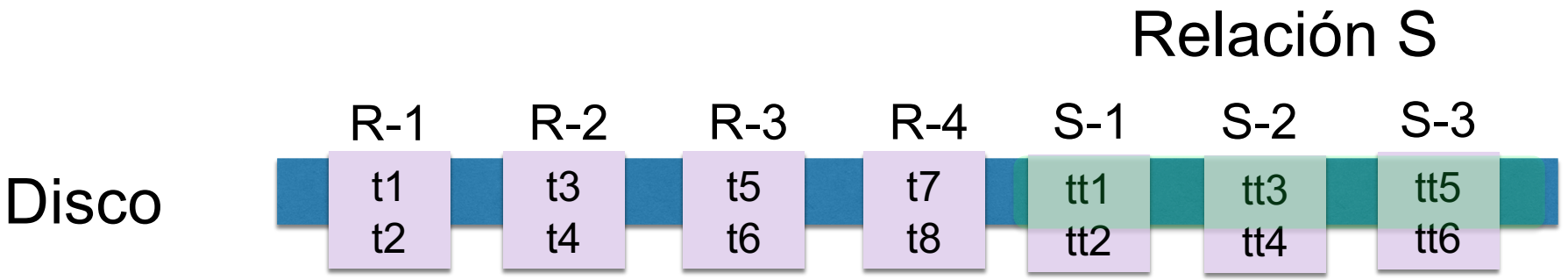
Páginas, disco y buffer



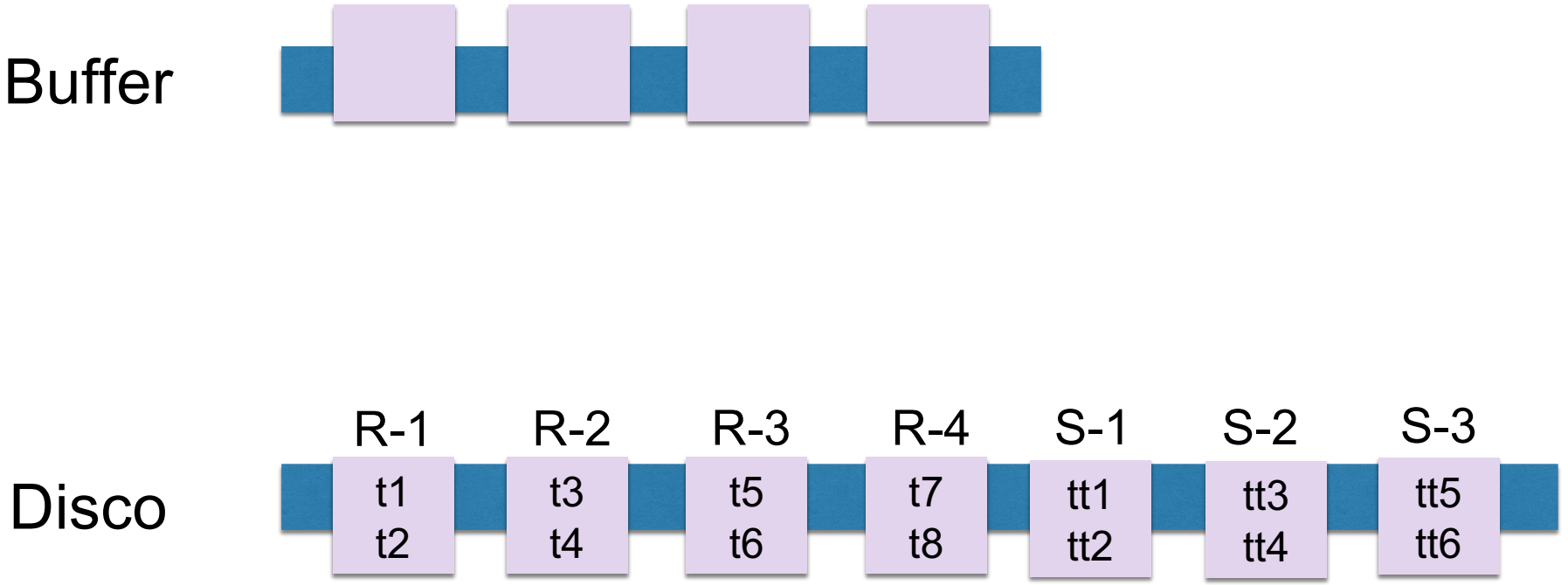
Páginas, disco y buffer



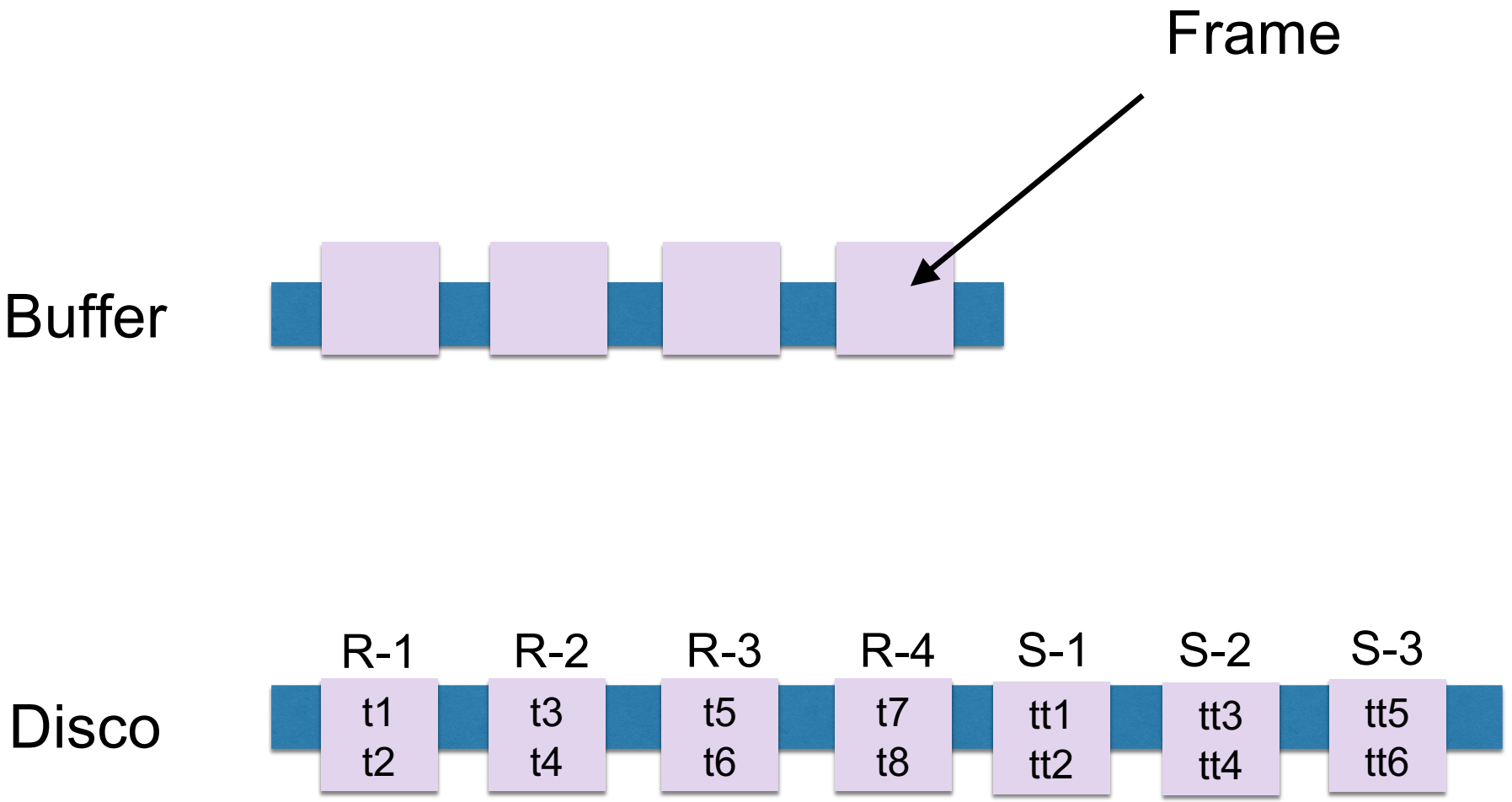
Páginas, disco y buffer



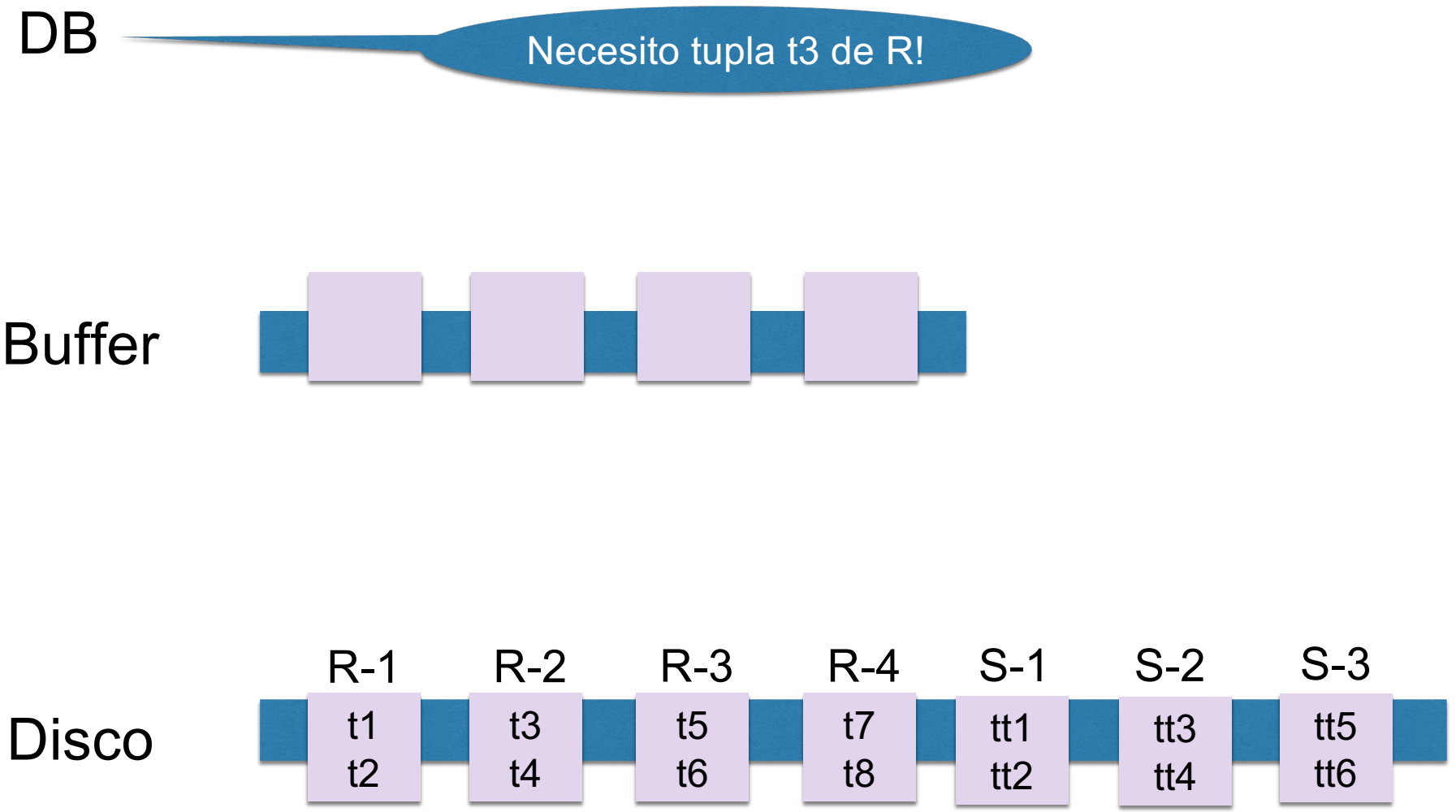
Páginas, disco y buffer



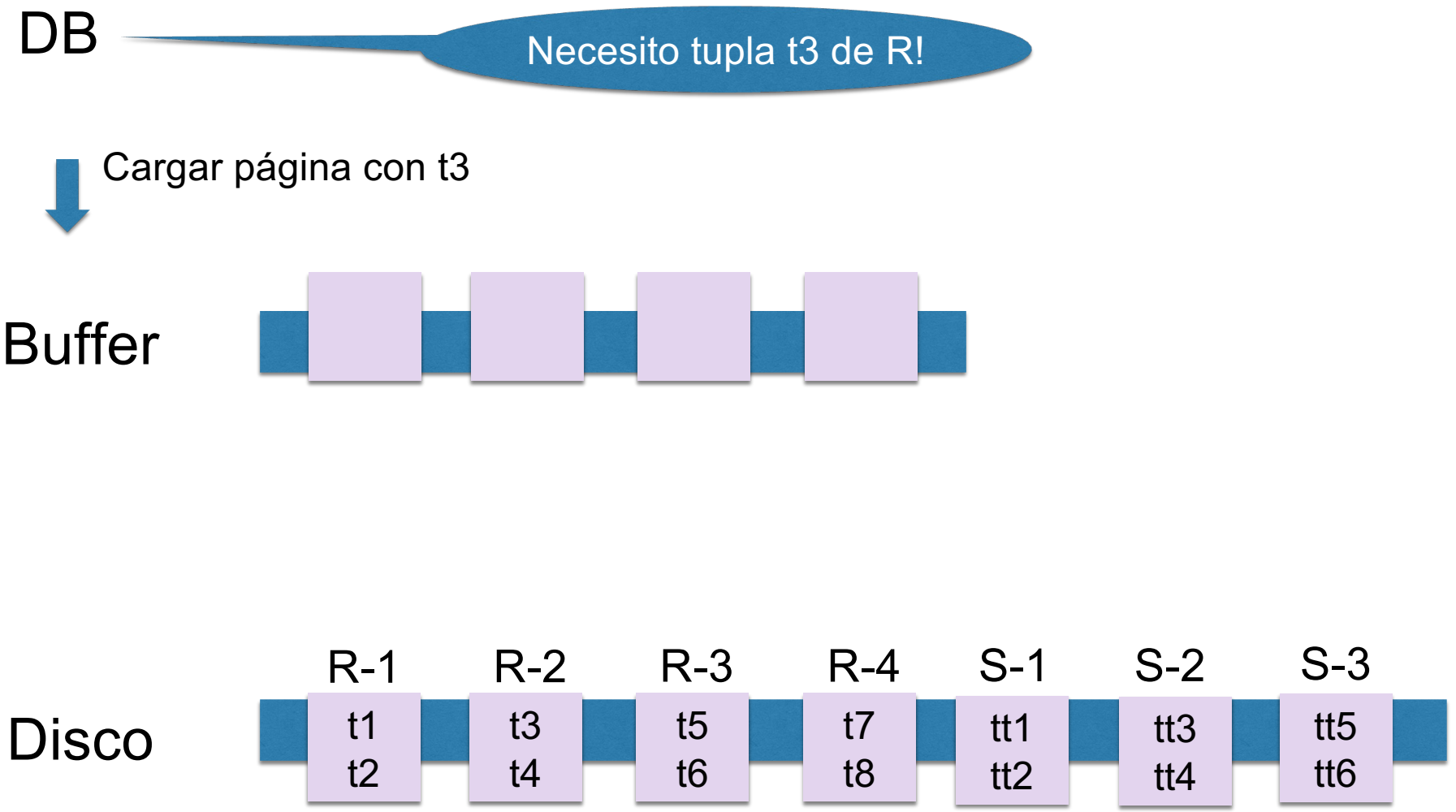
Páginas, disco y buffer



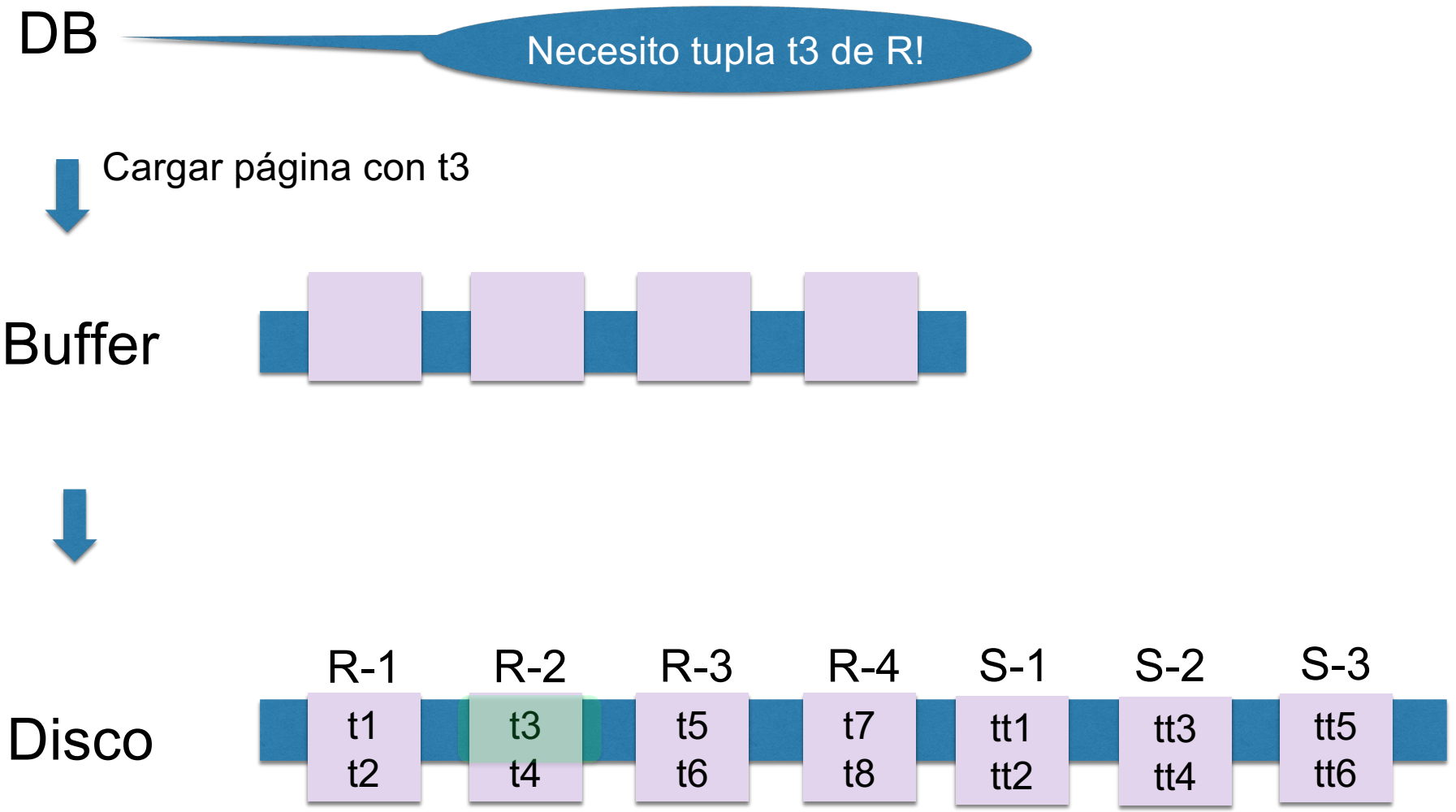
Páginas, disco y buffer



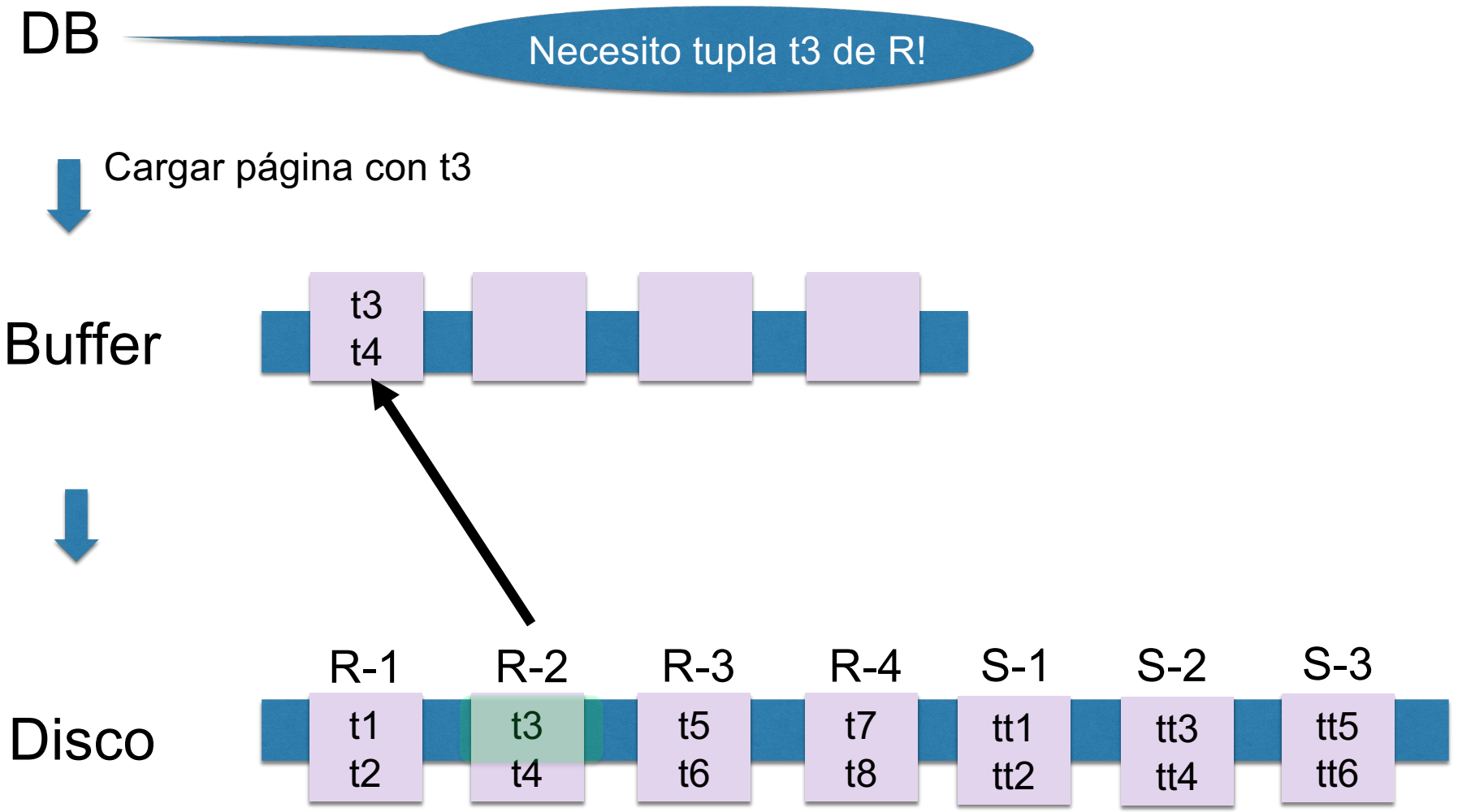
Páginas, disco y buffer



Páginas, disco y buffer



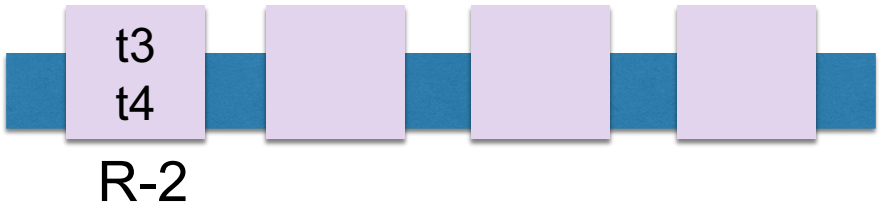
Páginas, disco y buffer



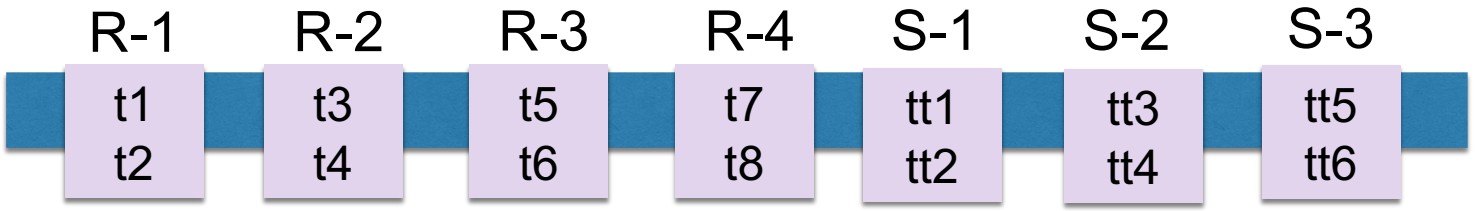
Páginas, disco y buffer

DB

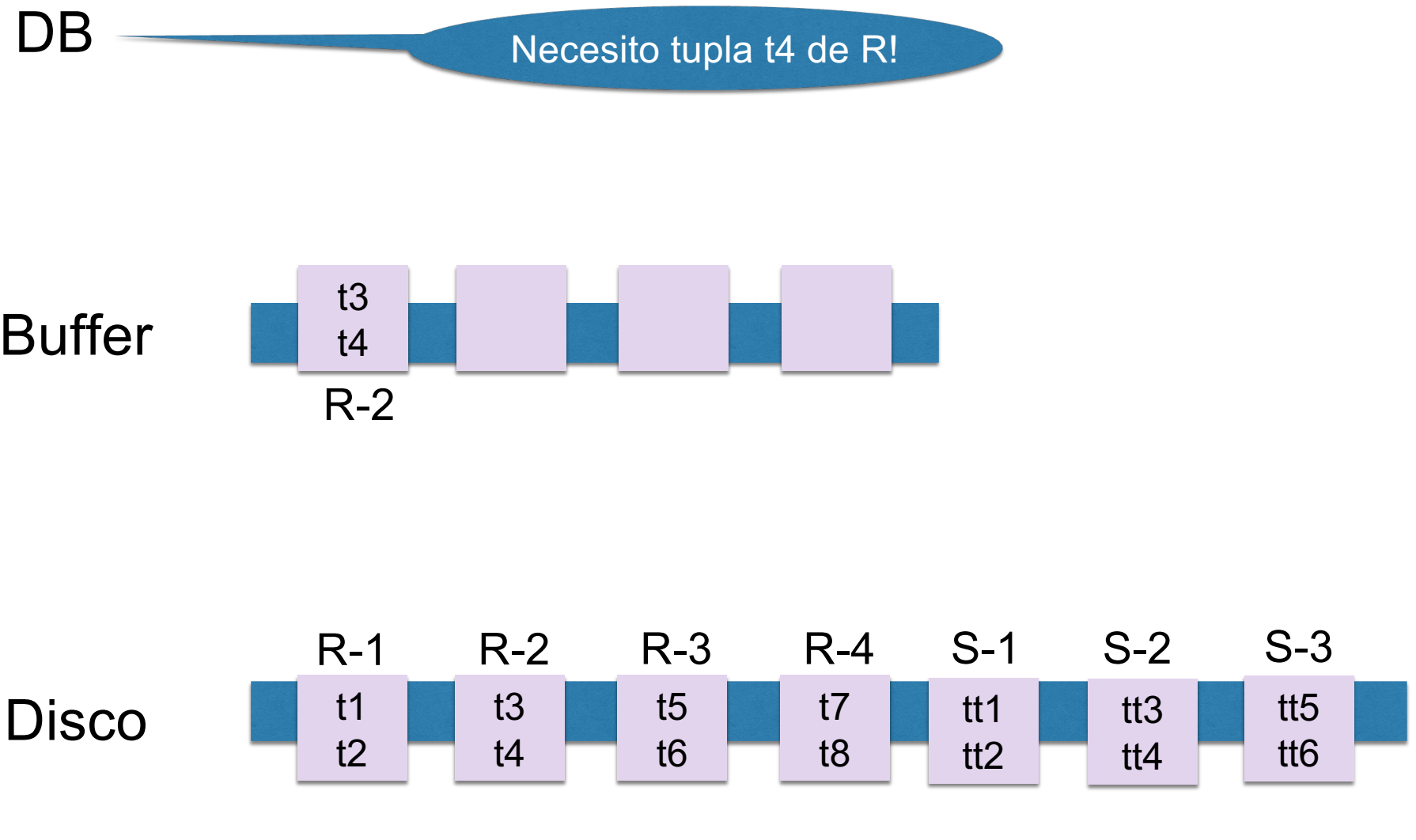
Buffer



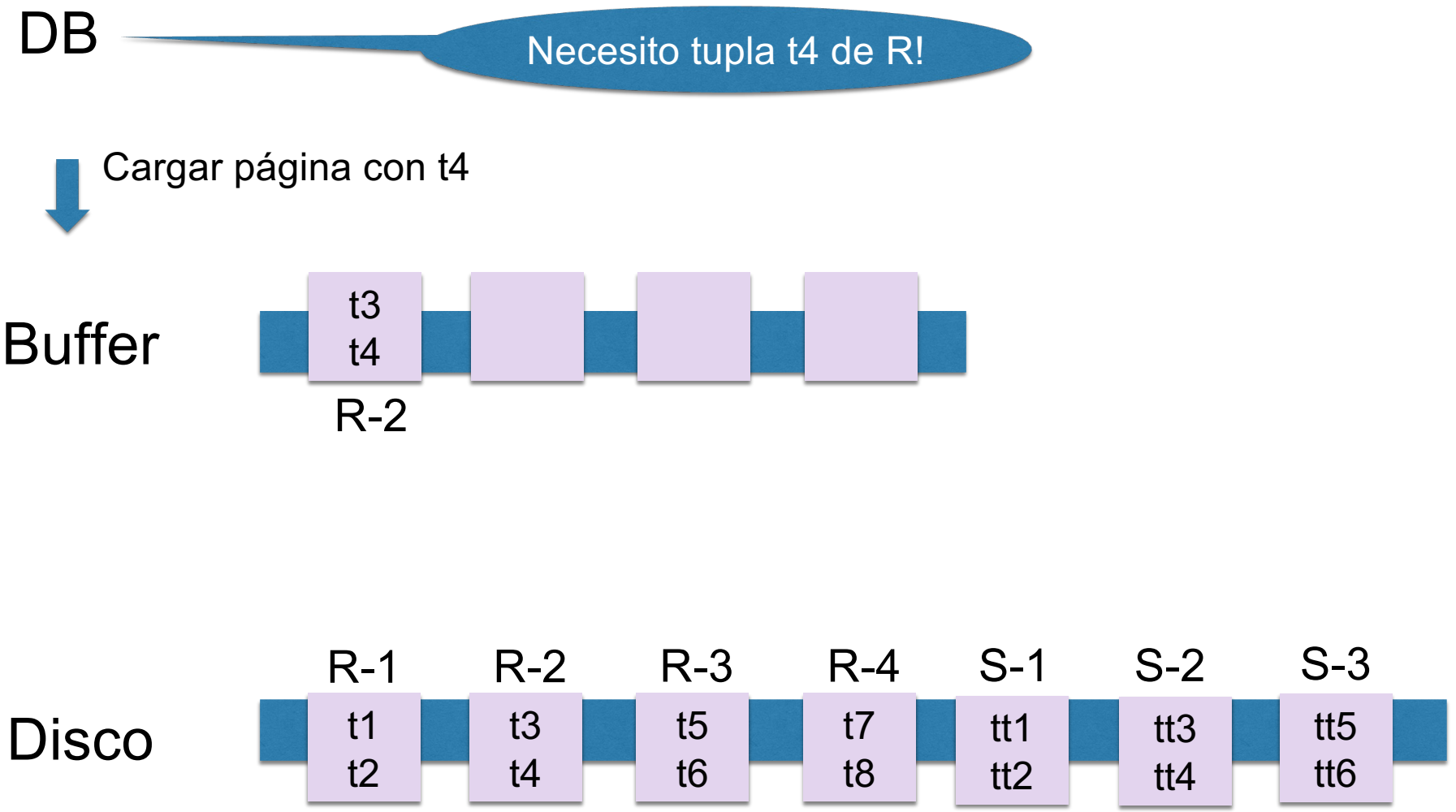
Disco



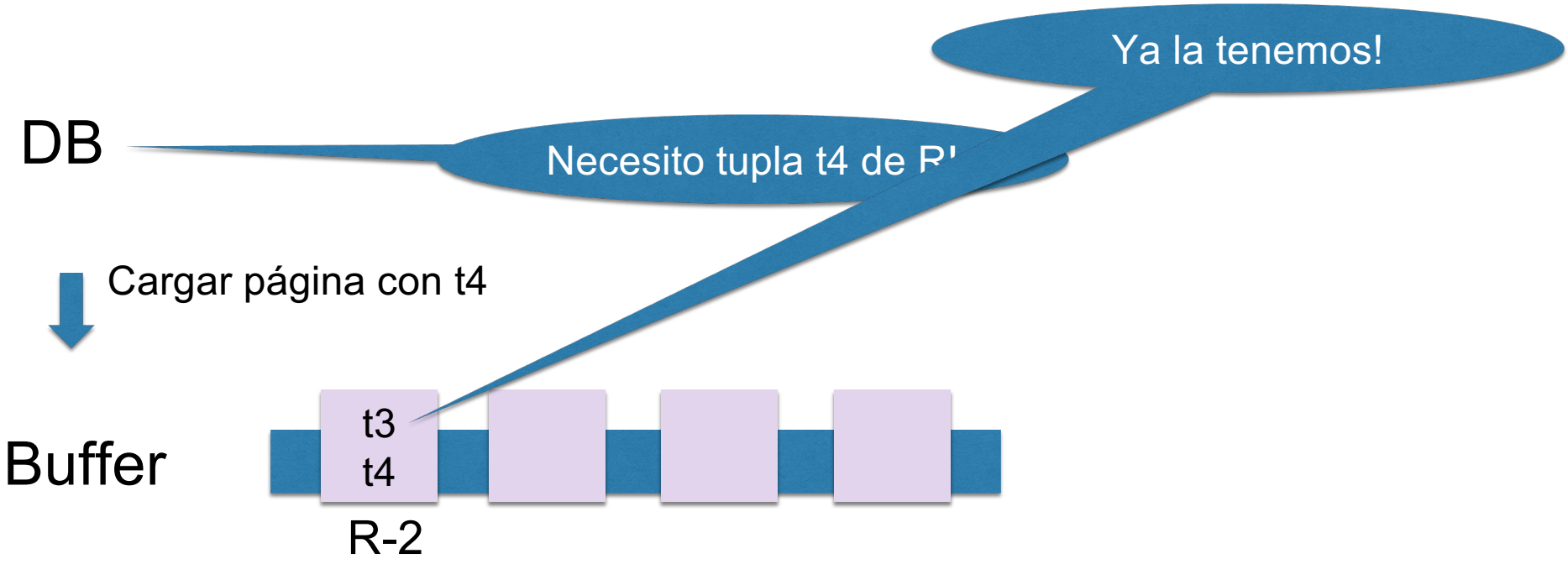
Páginas, disco y buffer

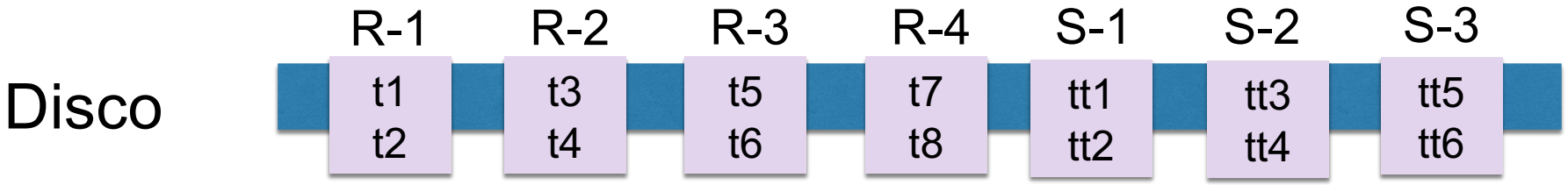


Páginas, disco y buffer



Páginas, disco y buffer

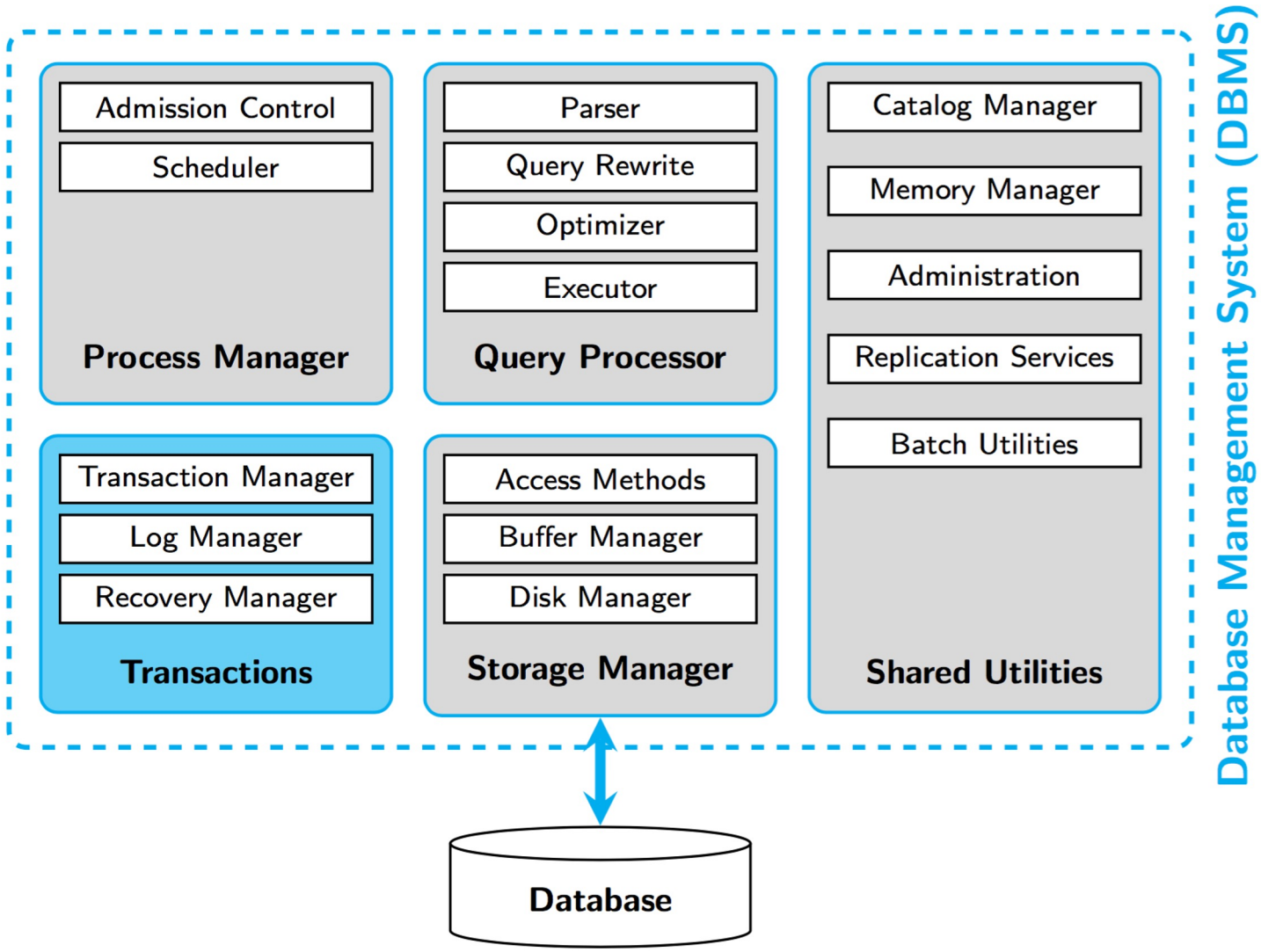




Fallas en Transacciones


EL SERVIDOR HA FALLADO
¿DÓNDE ESTÁ LA COPIA DE SEGURIDAD?
EN EL SERVIDOR....

Repaso: Transacciones



Transacciones

Es un componente que asegura las propiedades **ACID**



Atomicity
Consistency
Isolation
Durability

Atomicidad
Consistencia
a**I**slamiento
Durabilidad

ACID 

Atomicity: O se ejecutan todas las operaciones de la transacción, o no se ejecuta ninguna.

Consistency: Cada transacción preserva la consistencia de la BD (restricciones de integridad, etc.).

Isolation: Cada transacción debe ejecutarse como si se estuviese ejecutando sola, de forma aislada.

Durability: Los cambios que hace cada transacción son permanentes en el tiempo, independiente de cualquier tipo de falla.

En esta clase nos centraremos en Atomicity y Durability.

Transacciones



Transaction Manager se encarga de asegurar Isolation y Consistency

Log y Recovery Manager se encargan de asegurar **Atomicity** y **Durability**

Tipos de fallas en Transacciones

Fallas en la ejecucion de transacciones

1. Datos Erróneos: pueden ingresar en el sistema debido a errores humanos, fallas en la validación de los datos o problemas en la integración de sistemas externos. Soluciones:

- **Restricciones de integridad:** Implementar restricciones a nivel de base de datos como claves primarias, claves foráneas, y restricciones de unicidad y verificación.

- **Limpieza de datos (Data Cleaning):** Usar procesos para detectar y corregir registros erróneos o incompletos. Esto incluye la normalización de datos, la eliminación de duplicados y la corrección de formatos incorrectos.

Fallas en la ejecucion de transacciones

2. Fallas en el Almacenamiento: pueden ocurrir debido a fallos físicos en los dispositivos de almacenamiento, lo que puede resultar en la pérdida de datos críticos. Soluciones:

- **RAID (Redundant Array of Independent Disks):** Utilizar diferentes configuraciones de RAID para proteger los datos. Por ejemplo, RAID 1 duplica los datos en dos discos duros, mientras que RAID 5 distribuye los datos junto con la paridad a través de tres o más discos.

- **Copias Redundantes**: Mantener copias de seguridad regulares en dispositivos de almacenamiento separados.

Fallas en la ejecucion de transacciones

3. Catástrofes: Eventos imprevistos y graves como incendios, inundaciones, terremotos o ataques cibernéticos que pueden destruir los equipos físicos y los datos almacenados. Soluciones:

- **Copias Distribuidas**: Implementar un sistema de copias de datos distribuidas geográficamente. Esto asegura que, si una ubicación es afectada por una catástrofe, los datos pueden ser recuperados desde otra ubicación intacta.

- **Respaldos Totales e Incrementales**: Realizar copias de seguridad completas de toda la base de datos a intervalos regulares. También se deben tener respaldos incrementales, los cuales solo guardan los cambios desde el último respaldo, sea total o incremental.

Fallas en la ejecucion de transacciones

4. Fallas del Sistema: Interrupciones en el funcionamiento normal del sistema de base de datos, ya sea por fallos de software, hardware o errores humanos que afectan la integridad de las transacciones. Solución:

- **Log y Recovery Manager**



Log Manager

Ejemplo de log

2023-10-24 02:39:13.320 UTC [1125037] grupo3@grupo3e2 ERROR: type "varcahr" does not exist at character 60
2023-10-24 02:39:13.320 UTC [1125037] grupo3@grupo3e2 STATEMENT: create table genero_en_pelicula(id INT PRIMARY KEY, genero VARCAHR(40), pid INT, titulo VARCHAR(60));

2023-10-24 03:42:54.934 UTC [1134253] grupo3@grupo3e2 ERROR: column "pid" of relation "visualizaciones_series" does not exist
2023-10-24 03:42:54.934 UTC [1134253] grupo3@grupo3e2 STATEMENT: COPY visualizaciones_series (id_visualizacion, uid, pid, fecha) FROM STDIN DELIMITER ',' CSV HEADER;

2023-10-24 04:09:19.361 UTC [1136729] grupo3@grupo3e2 ERROR: column "id" of relation "cancelaciones" does not exist
2023-10-24 04:09:19.361 UTC [1136729] grupo3@grupo3e2 STATEMENT: COPY cancelaciones (id,estado, fecha_inicio, pro_id,uid, proveedor) FROM STDIN DELIMITER ',' CSV HEADER;

2023-10-24 04:09:35.542 UTC [1136729] grupo3@grupo3e2 ERROR: relation "subscripciones" does not exist
2023-10-24 04:09:35.542 UTC [1136729] grupo3@grupo3e2 STATEMENT: COPY subscripciones (id,estado, fecha_inicio, pro_id,uid, proveedor) FROM STDIN DELIMITER ',' CSV HEADER;

2023-10-24 04:12:00.863 UTC [1136729] grupo3@grupo3e2 ERROR: relation "subscripciones" does not exist
2023-10-24 04:12:00.863 UTC [1136729] grupo3@grupo3e2 STATEMENT: COPY subscripciones (id,estado, fecha_inicio, pro_id,uid, proveedor) FROM STDIN DELIMITER ',' CSV HEADER;

Log Manager

Registra todas las acciones de las transacciones

Una página se va llenando secuencialmente con *logs*
Cuando la página se llena, se almacena en disco

Todas las transacciones escriben el *log* de manera concurrente

Log Records

Los *logs* comunes son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T** UPDATE>

¿Cómo los usamos?, con **Loggins**.

Undo Logging

Undo Logging

Forma de escribir los *logs* para poder hacer *recovery* del sistema

Undo Logging

Los *logs* son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T**, X, t> donde t es el valor **antiguo** de X

Undo Logging - Reglas

Regla 1: si **T** modifica X, todos *logs* <**T**, X, t> deben ser escritos antes que el valor X sea escrito en disco

Regla 2: si **T** hace *commit*, el log <COMMIT **T**> debe ser escrito justo después de que todos los datos modificados por **T** estén almacenados en disco

Undo Logging

En resumen:

- **T** cambia el valor del X (t valor antiguo)
- Generar el log <**T**, X, t>

} repeat

- Escribir todos logs <**T**, X, t> al disco
- Escribir valor nuevo de X a disco

} **write-ahead logging**

- Escribir <COMMIT **T**>

Ejemplo de log de transacción

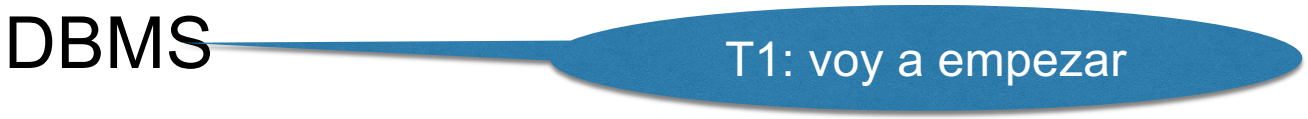
Start transaction T1

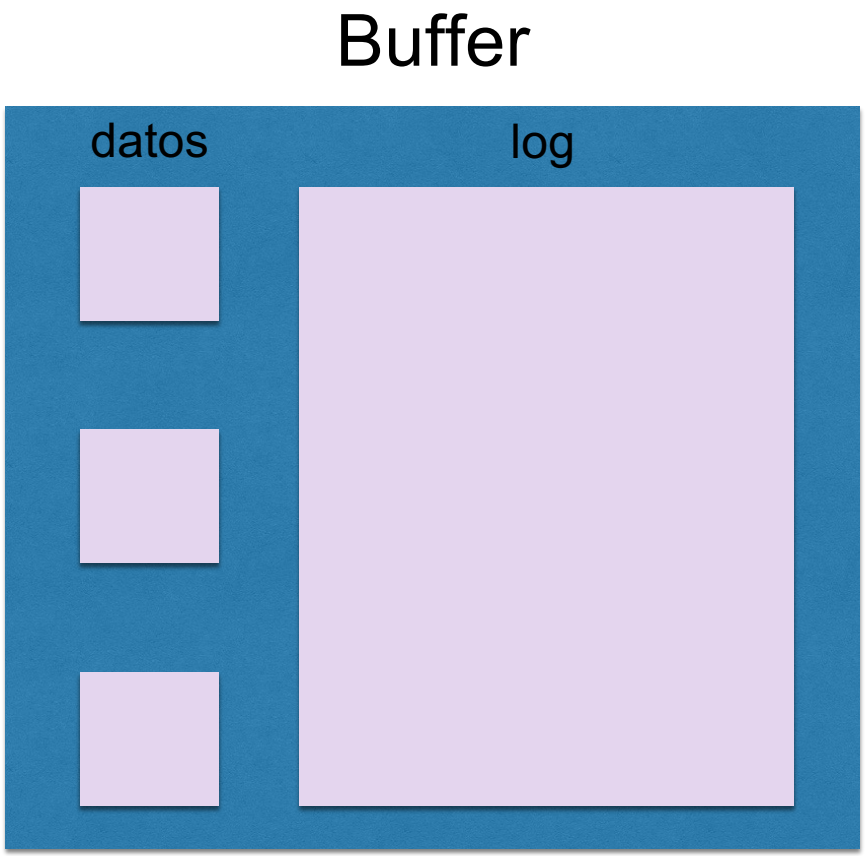
Reemplace el valor 31 de R.a por 99

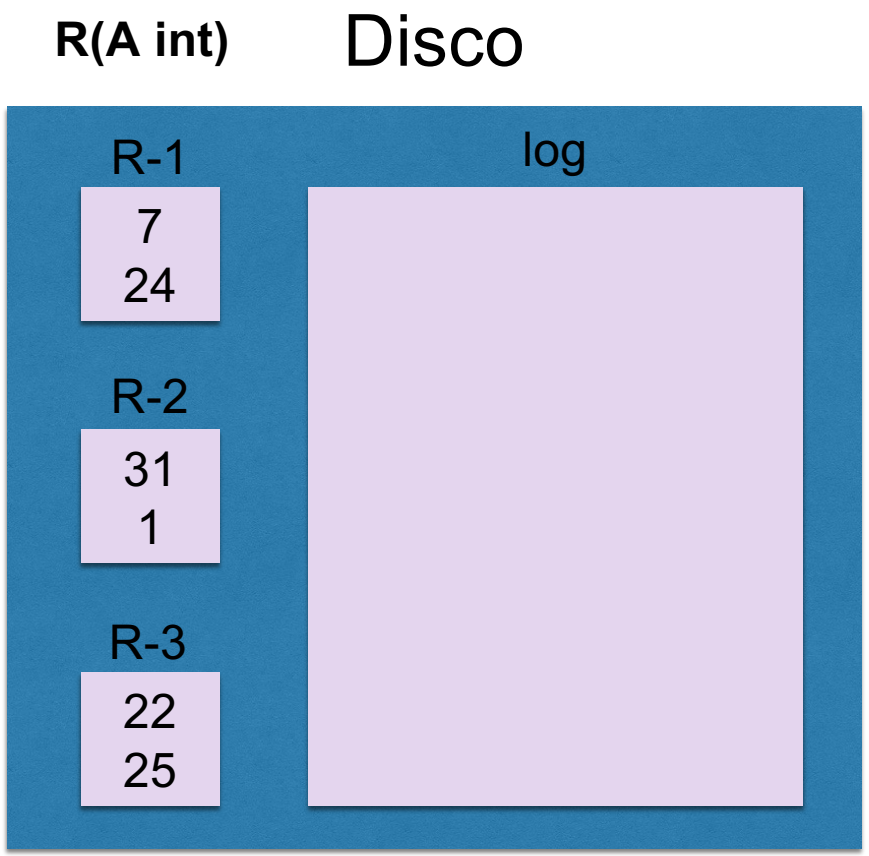
Reemplace el valor 99 de R.a por 23

Commit T1

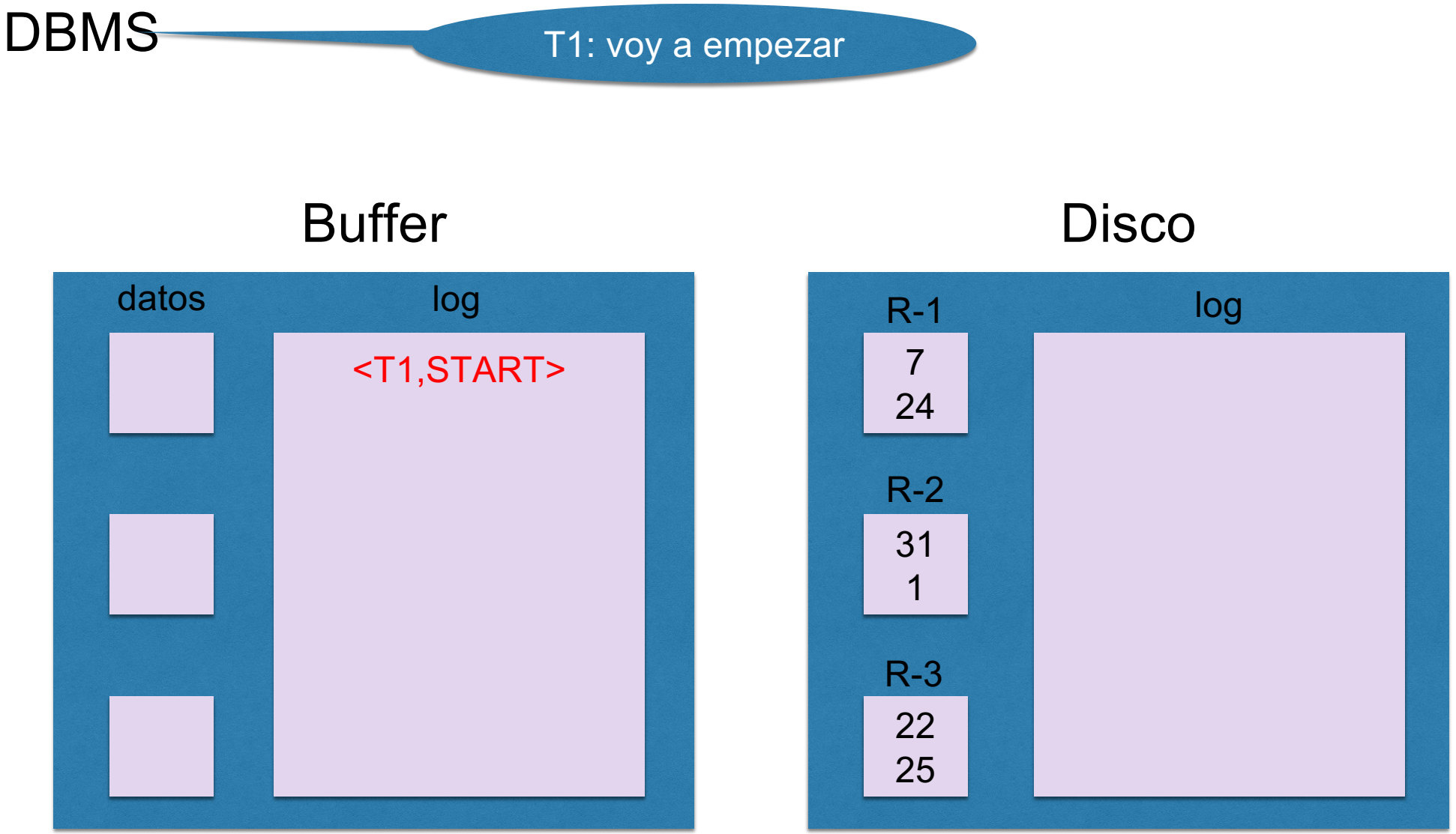
Undo Logging – en la BD



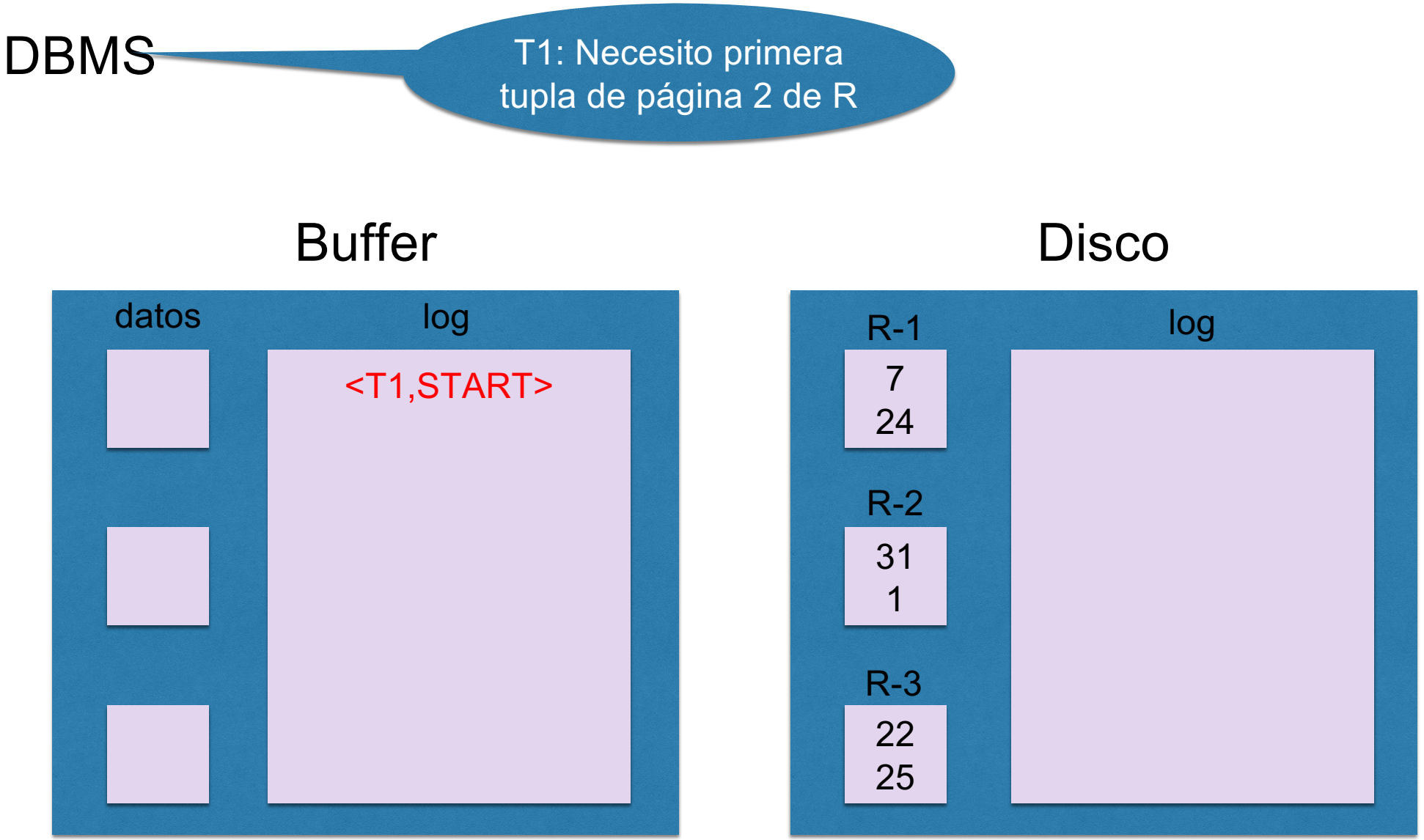




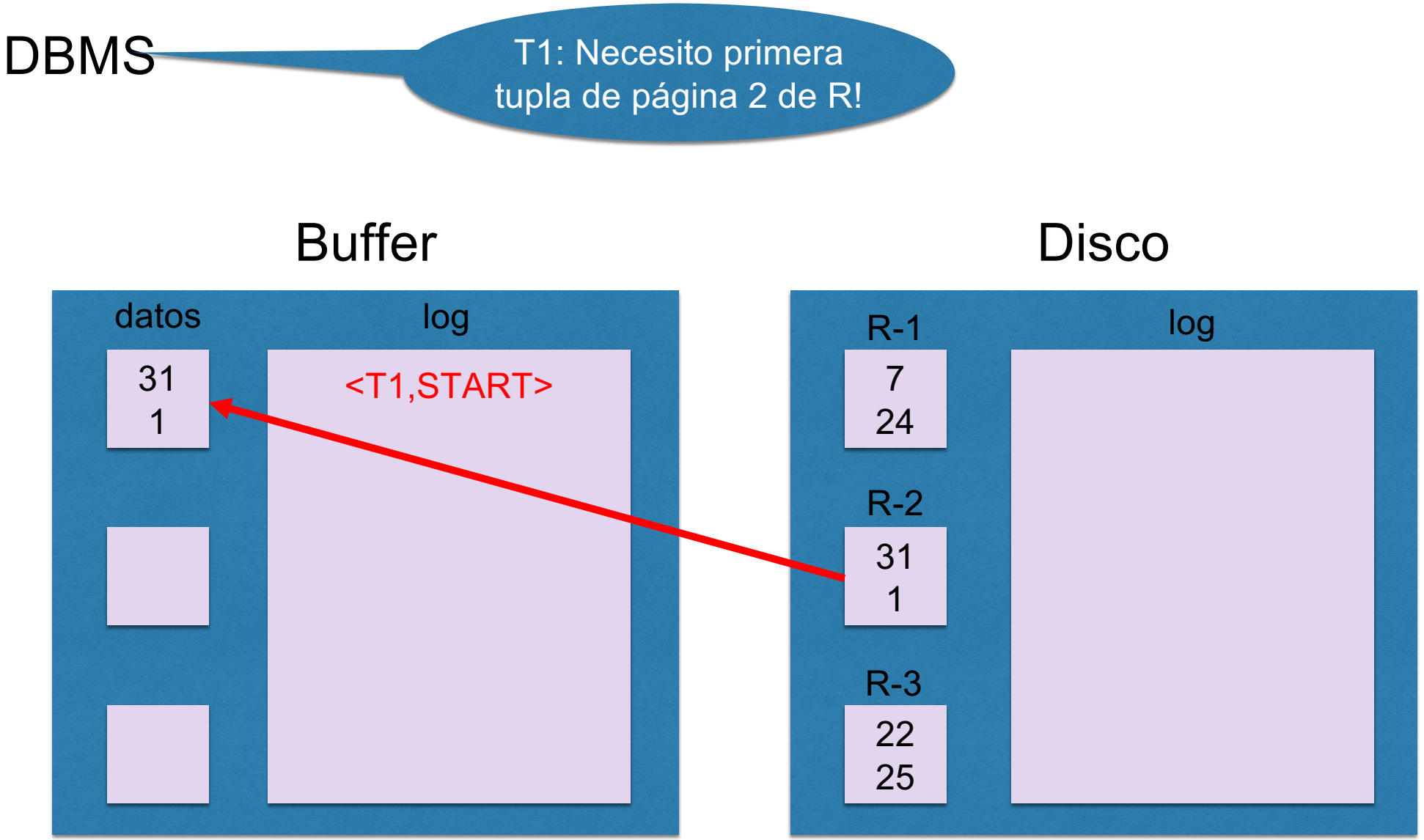
Undo Logging – en la BD



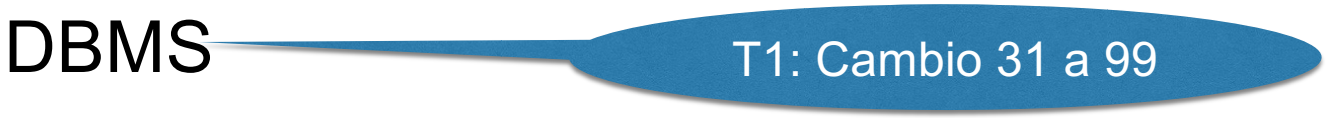
Undo Logging – en la BD

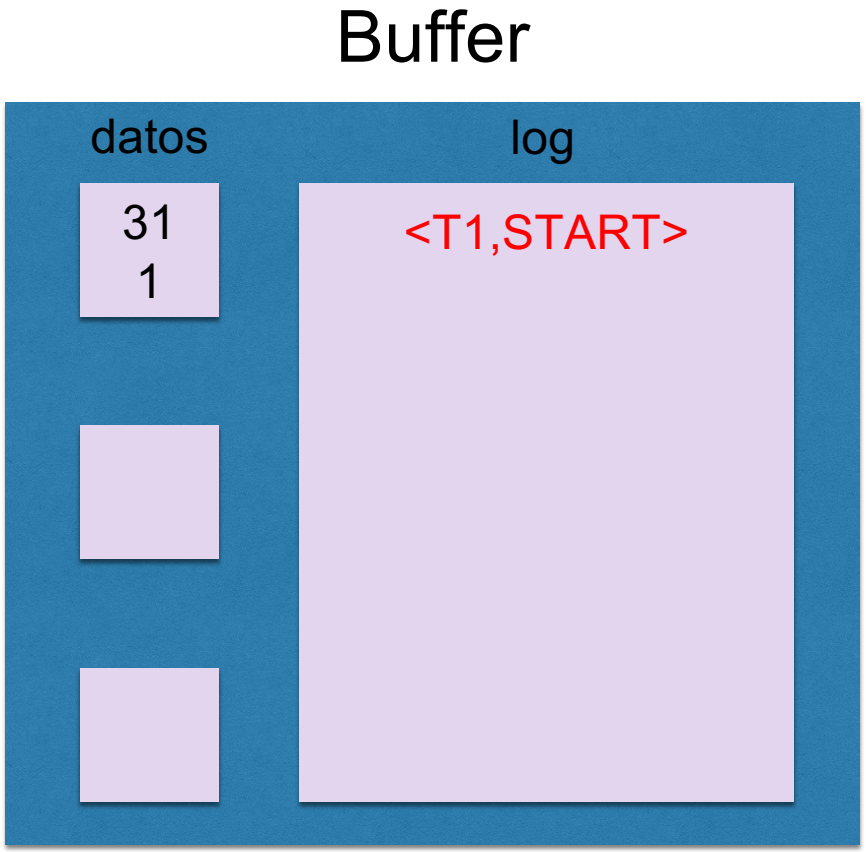


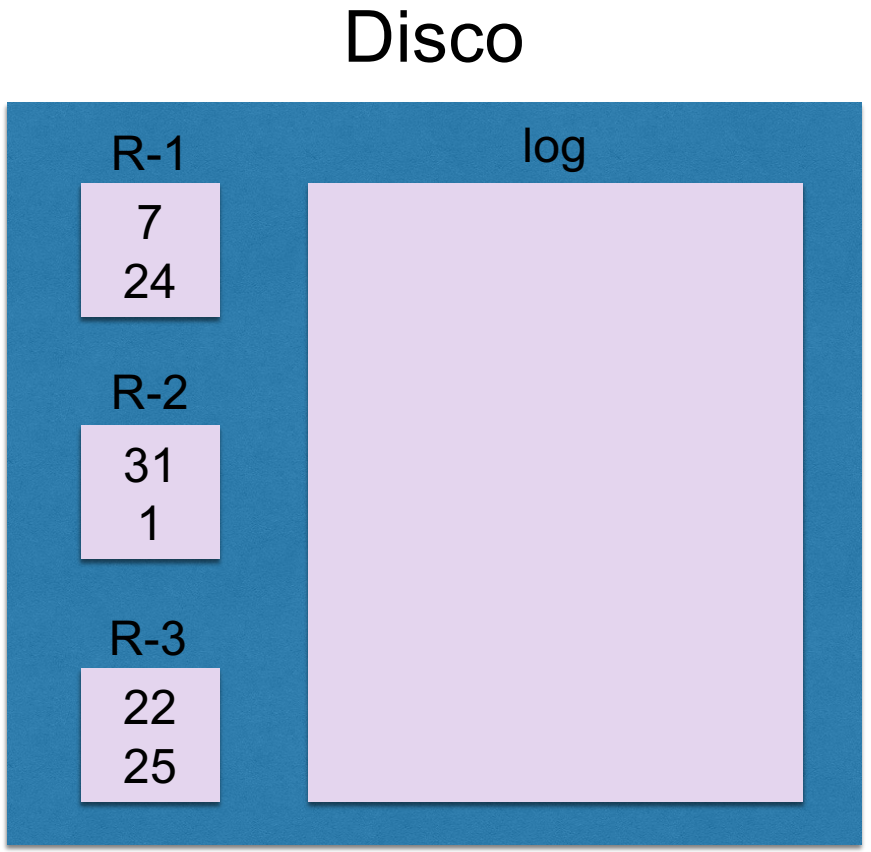
Undo Logging – en la BD



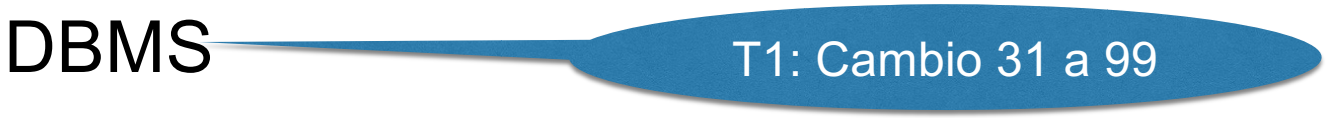
Undo Logging – en la BD

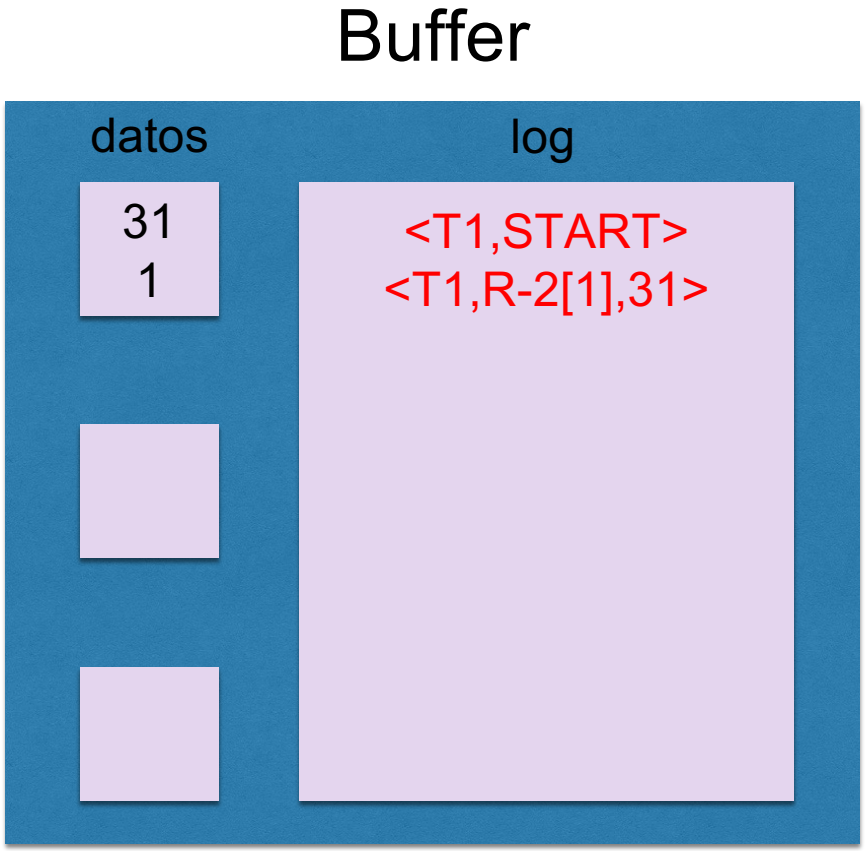


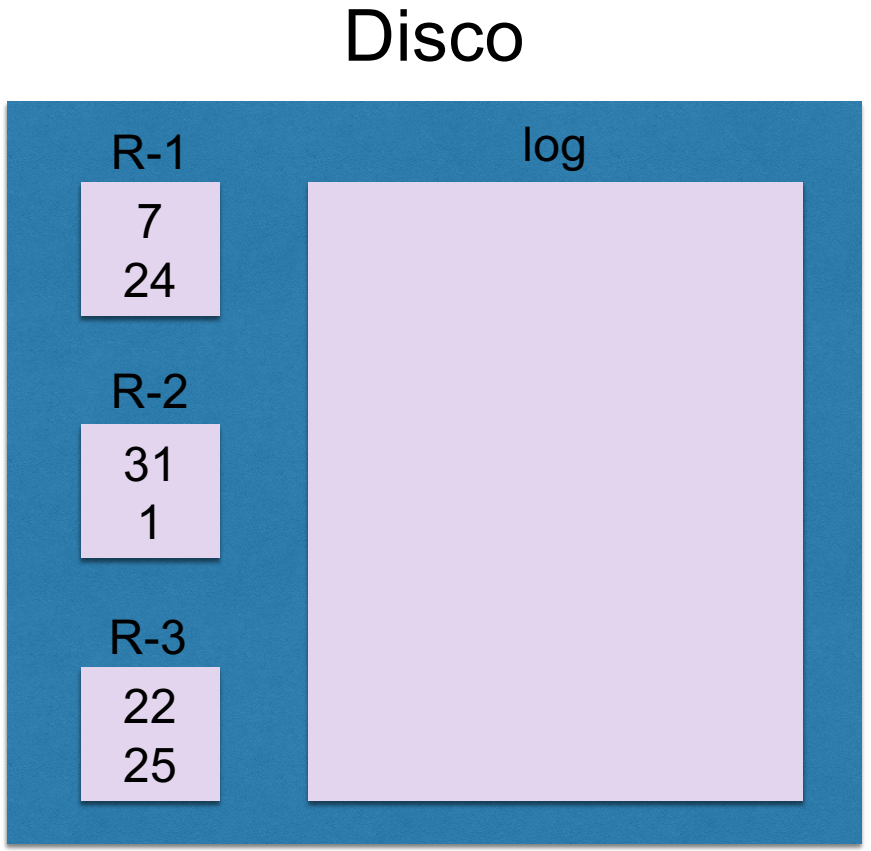




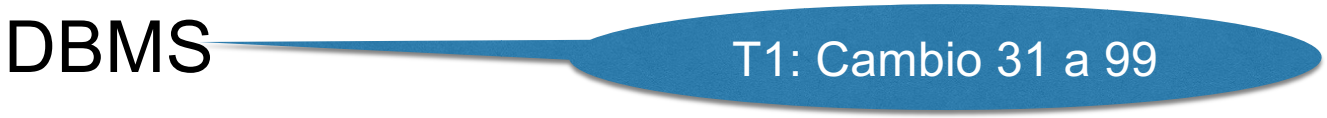
Undo Logging – en la BD

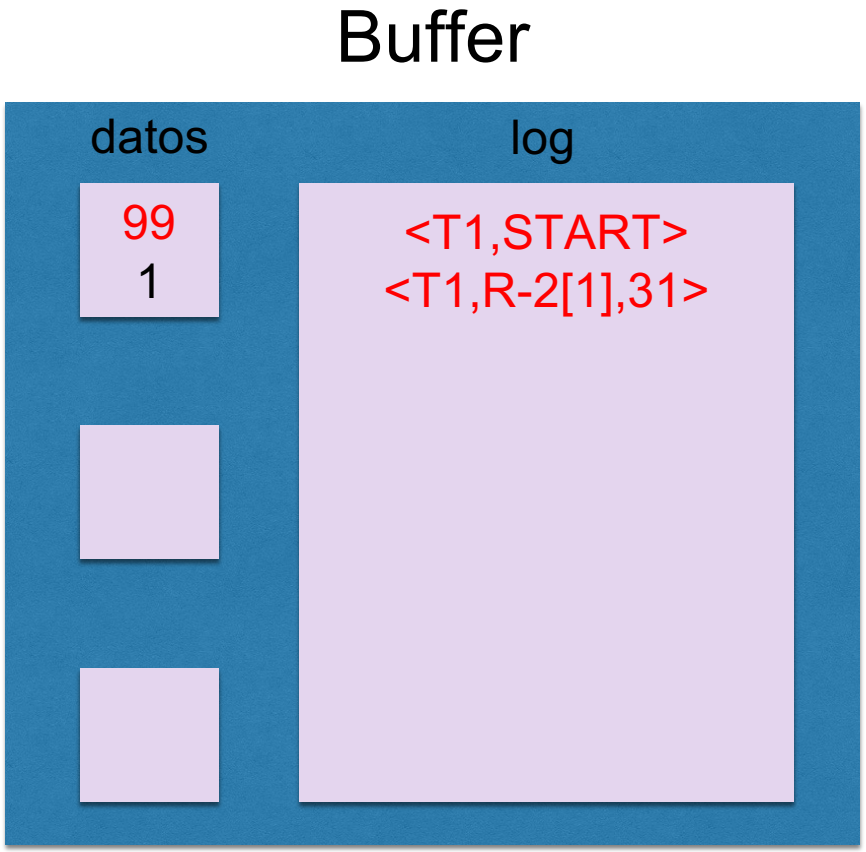


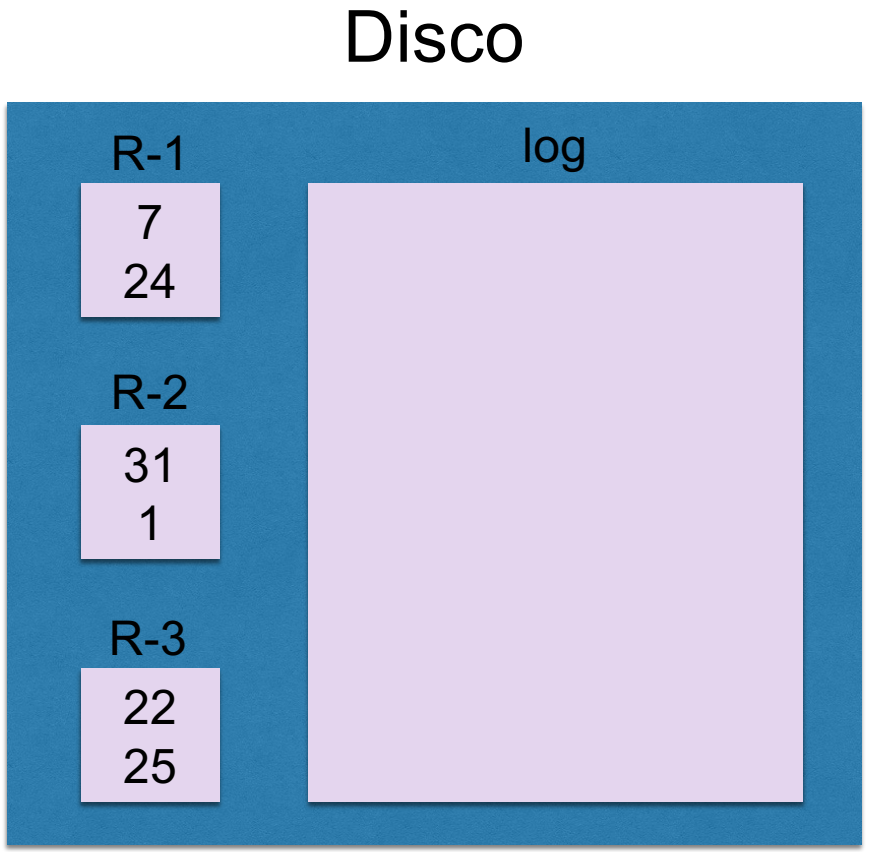




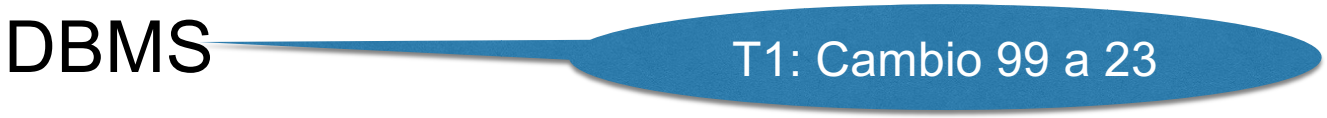
Undo Logging – en la BD

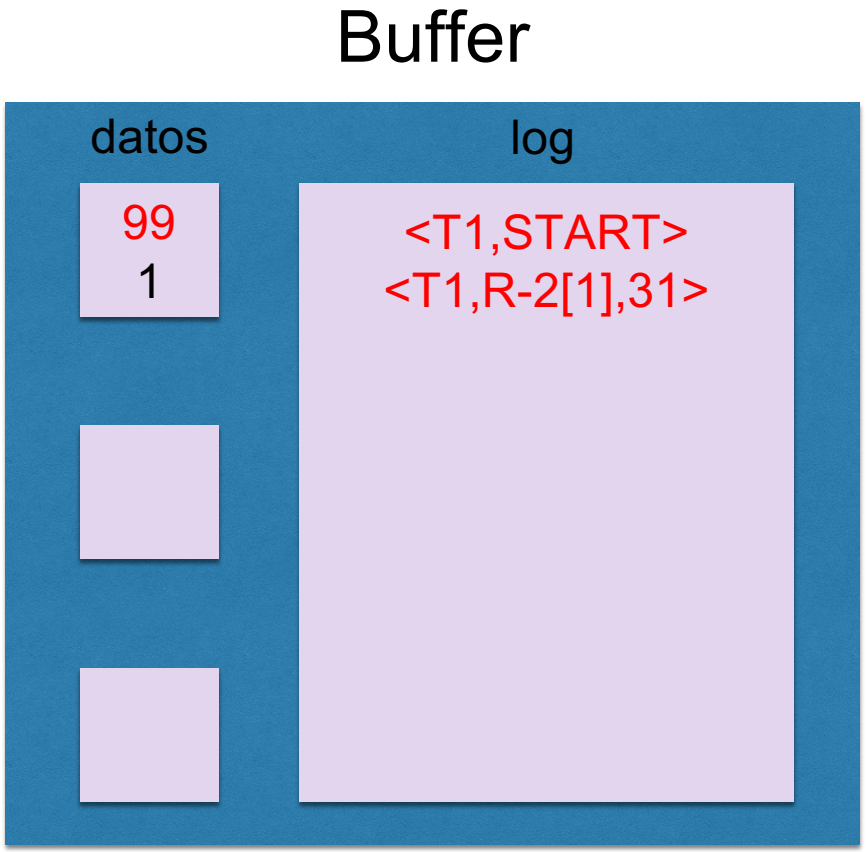


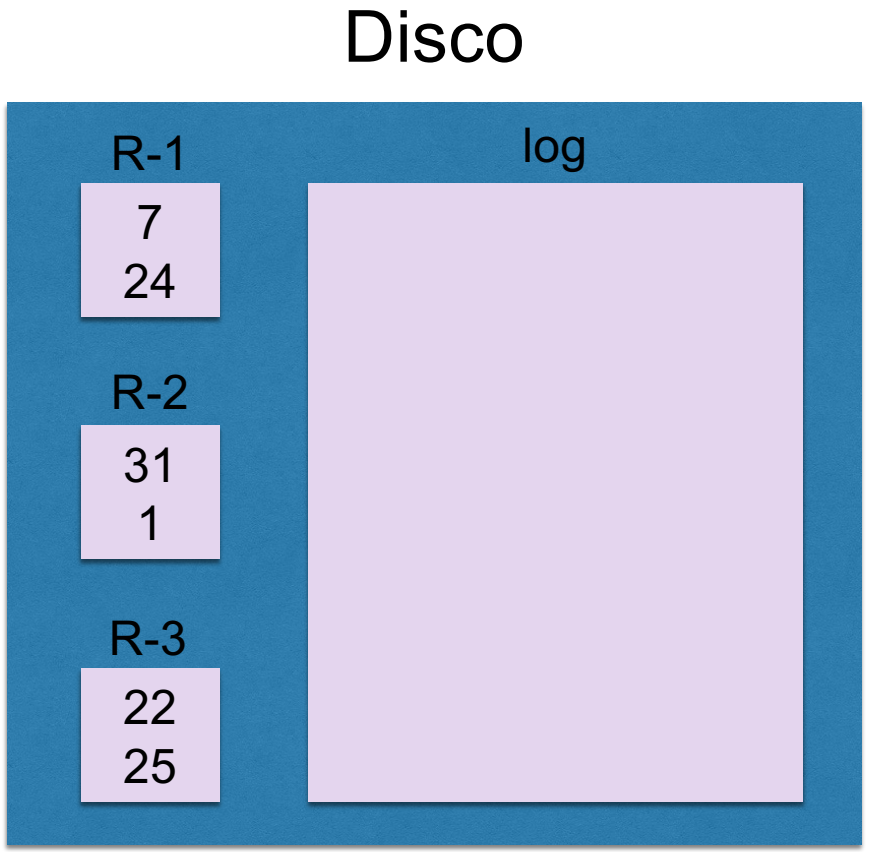




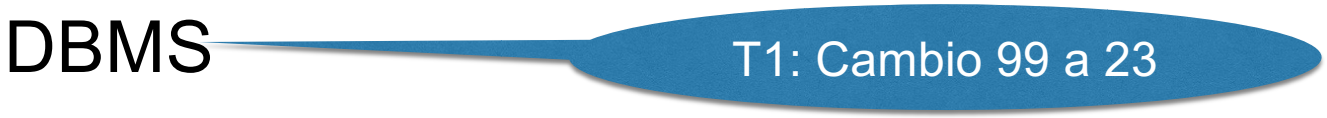
Undo Logging – en la BD

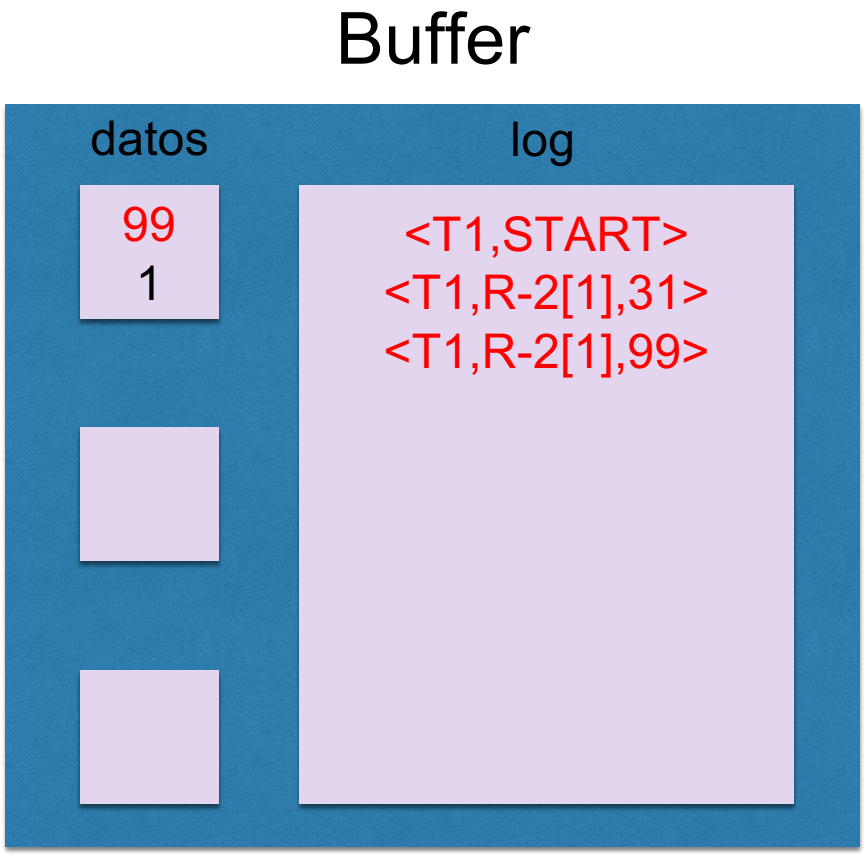


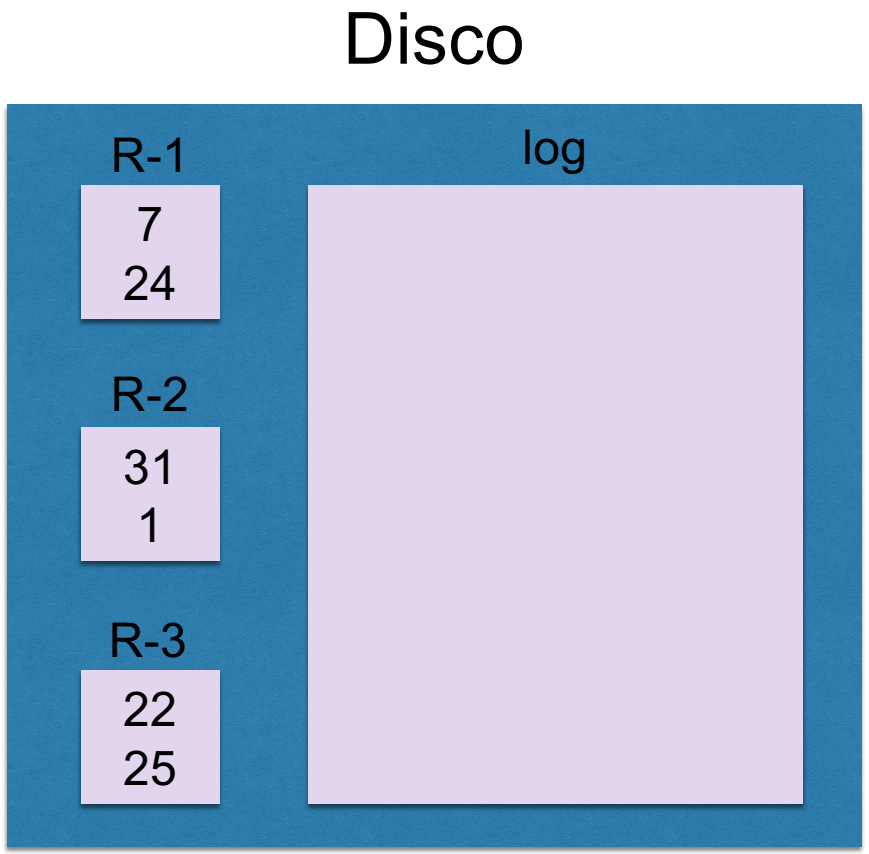




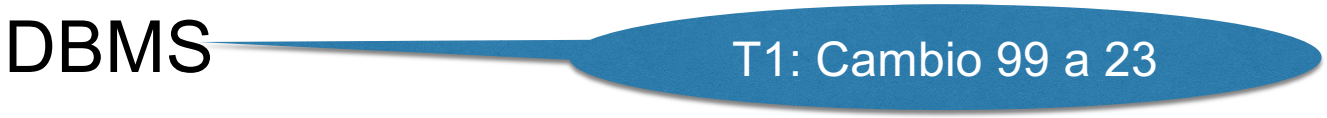
Undo Logging – en la BD

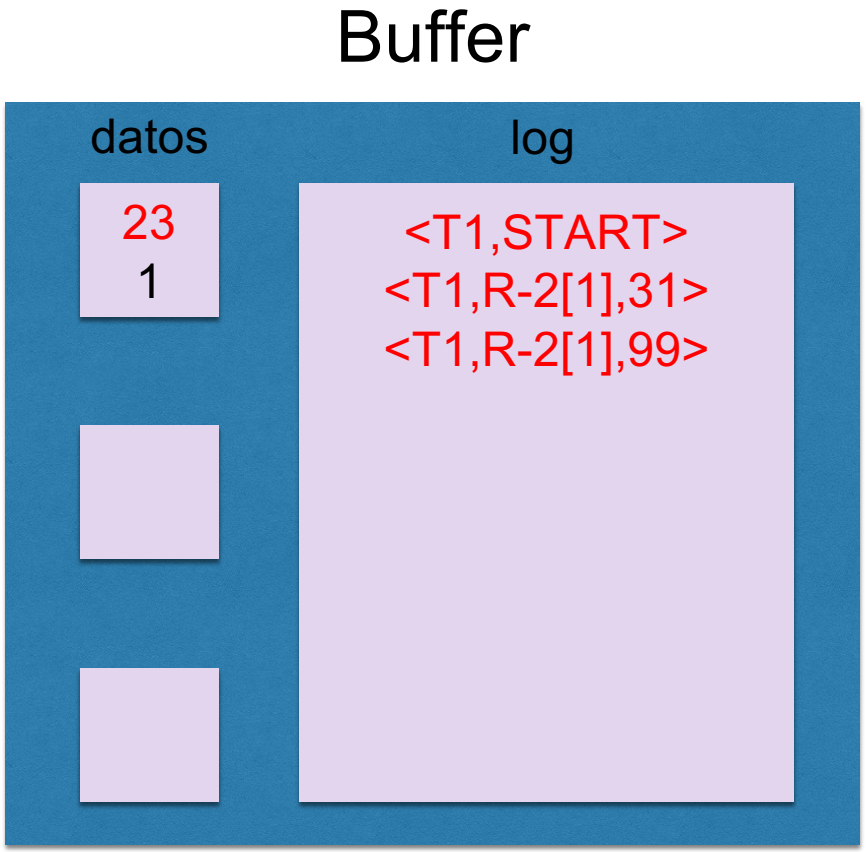


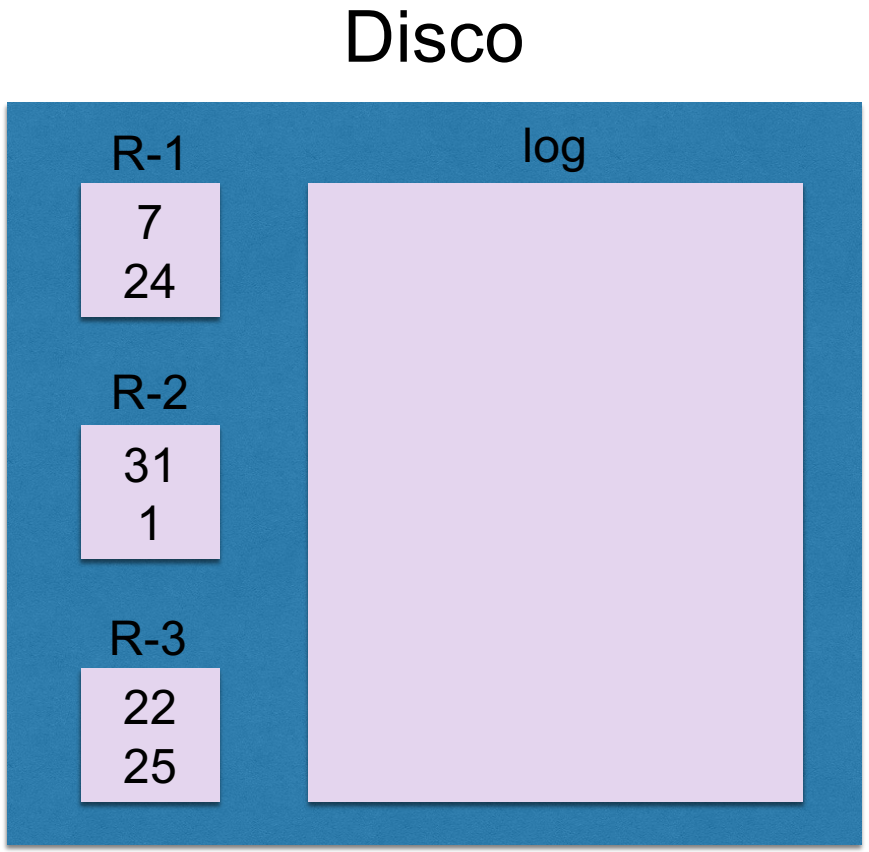




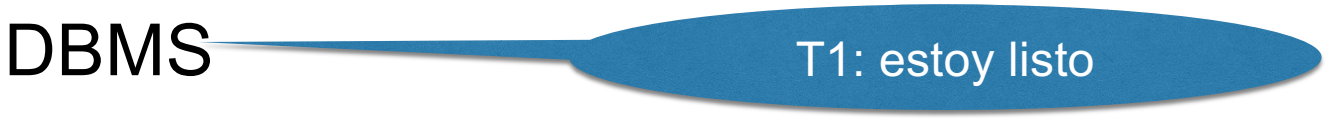
Undo Logging – en la BD

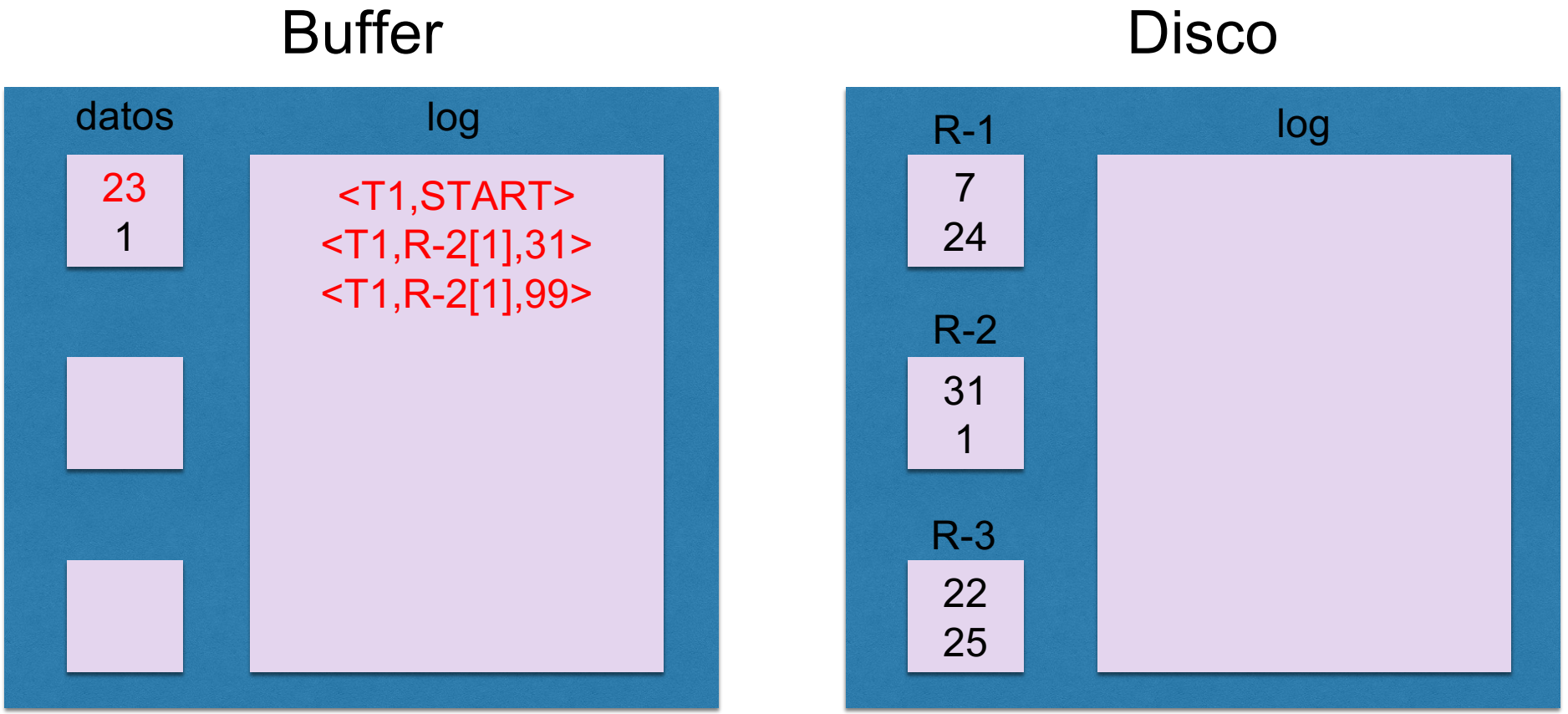




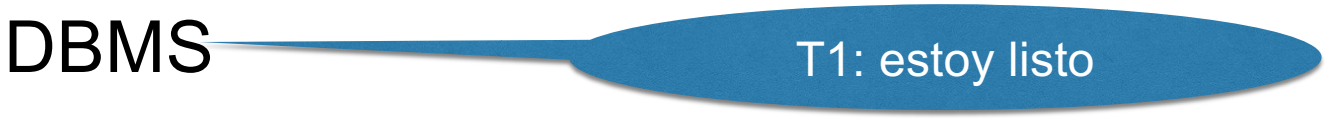


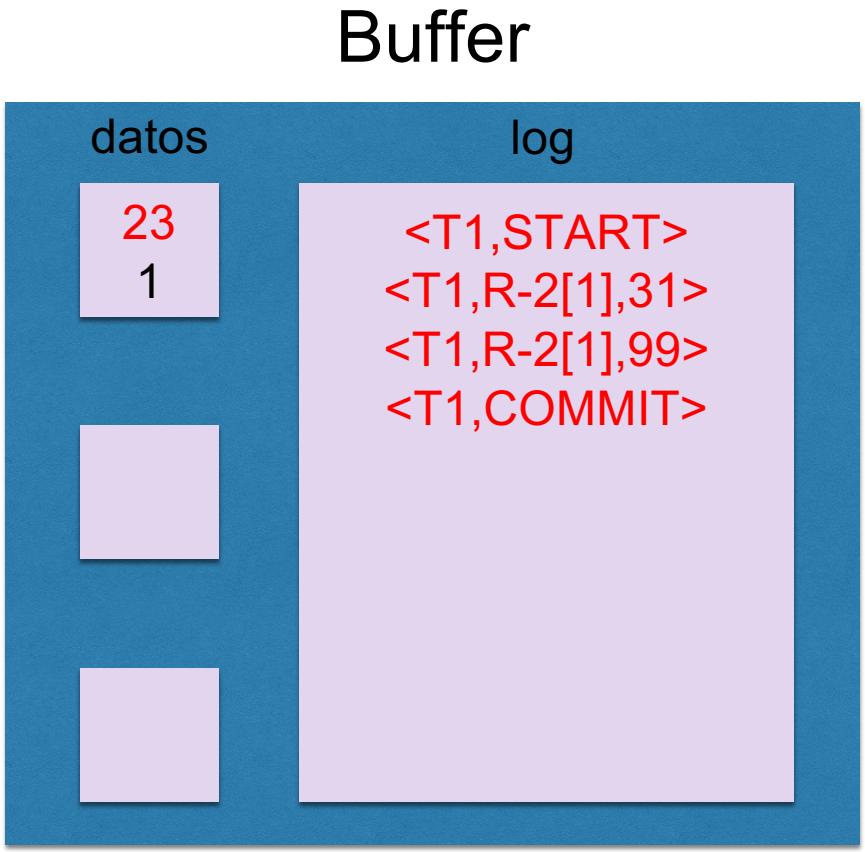
Undo Logging – en la BD

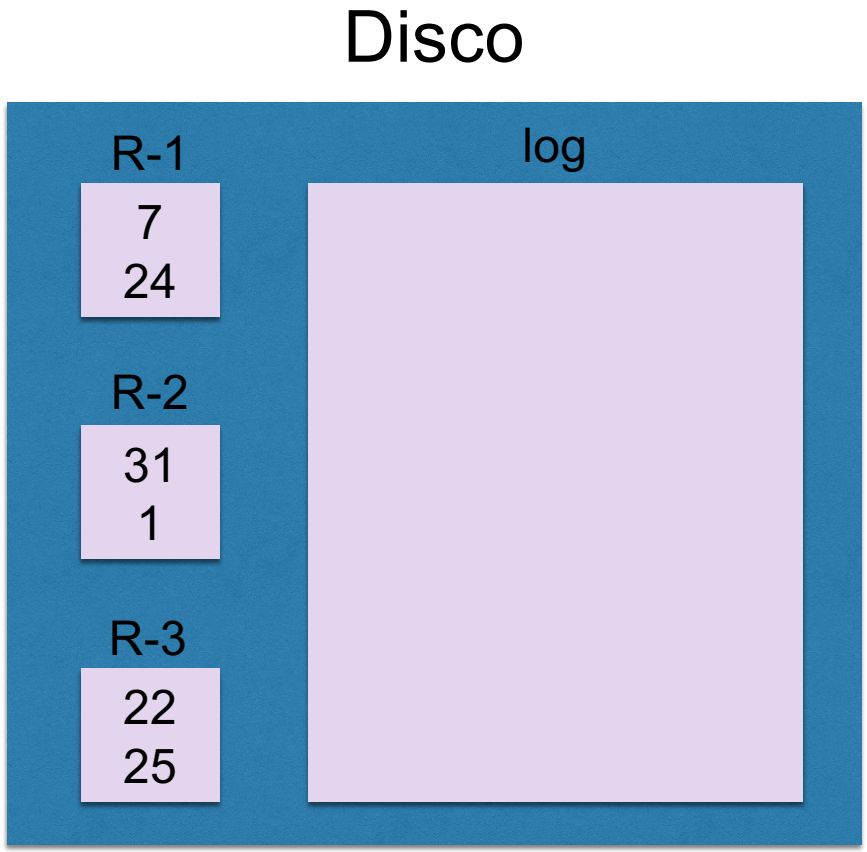




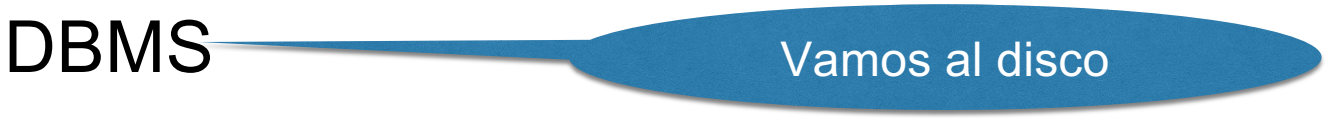
Undo Logging – en la BD

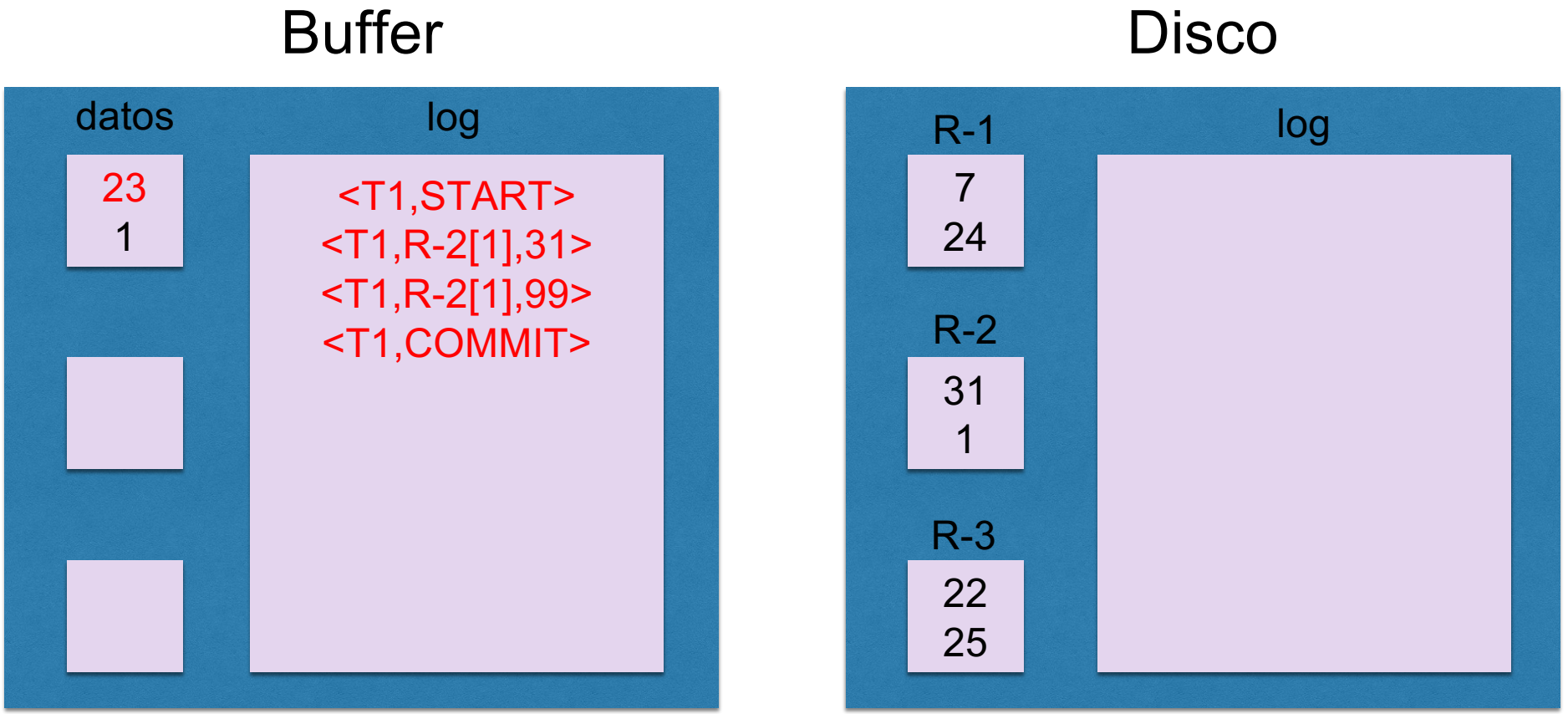




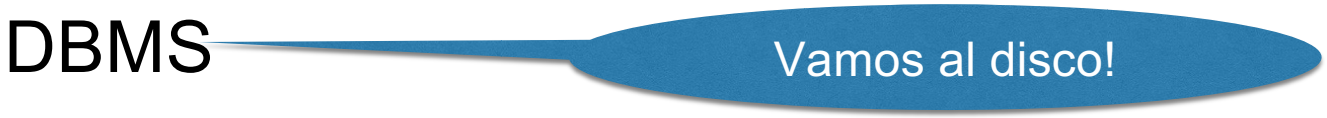


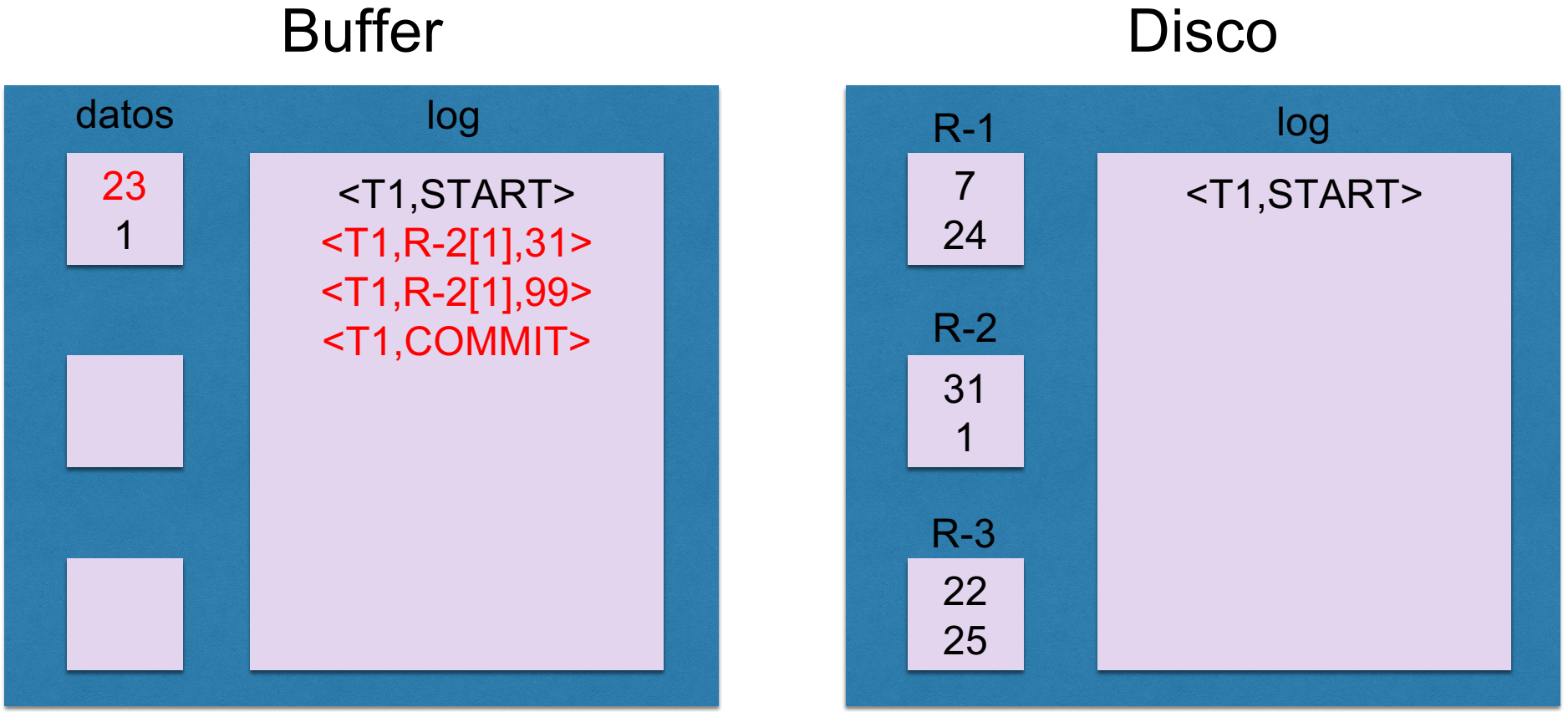
Undo Logging – en la BD





Undo Logging – en la BD

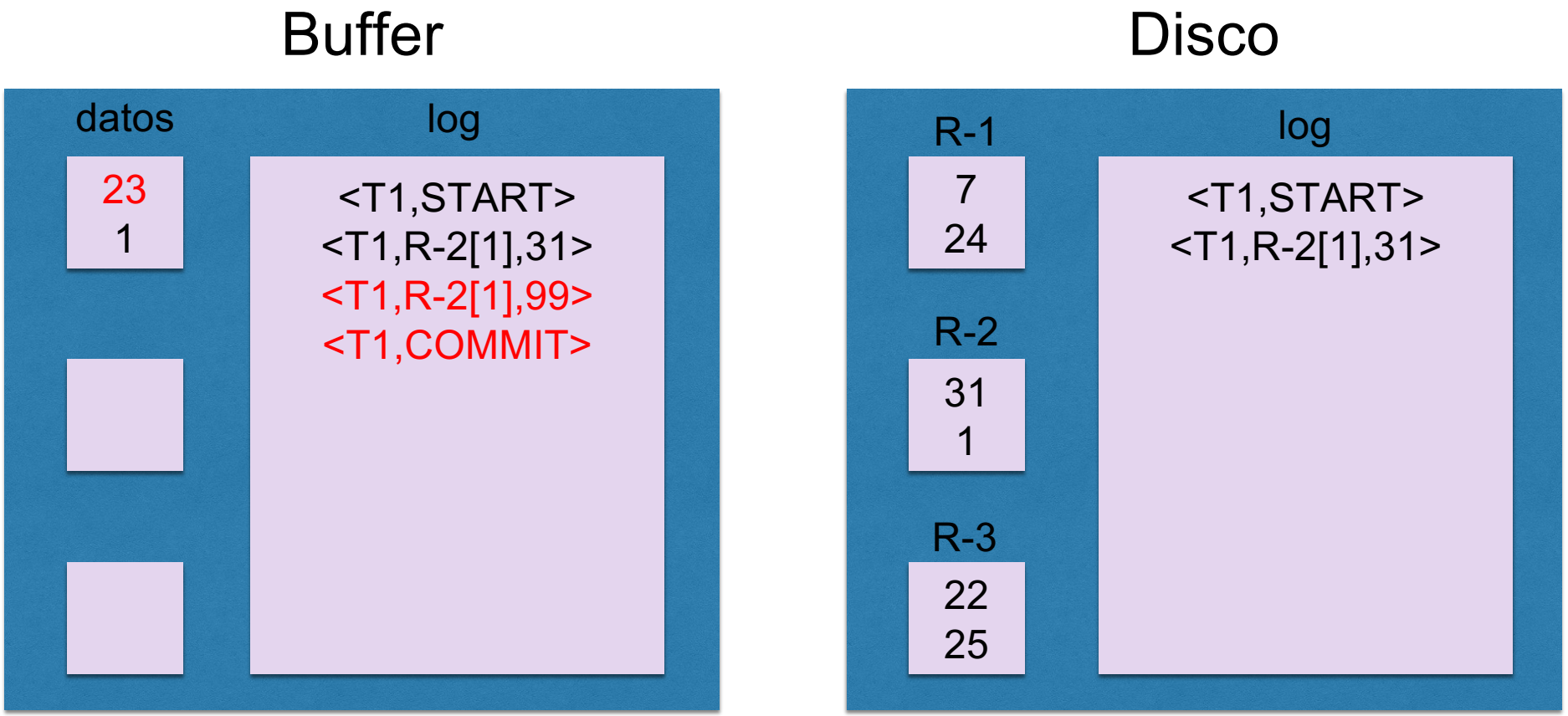




Undo Logging – en la BD

DBMS

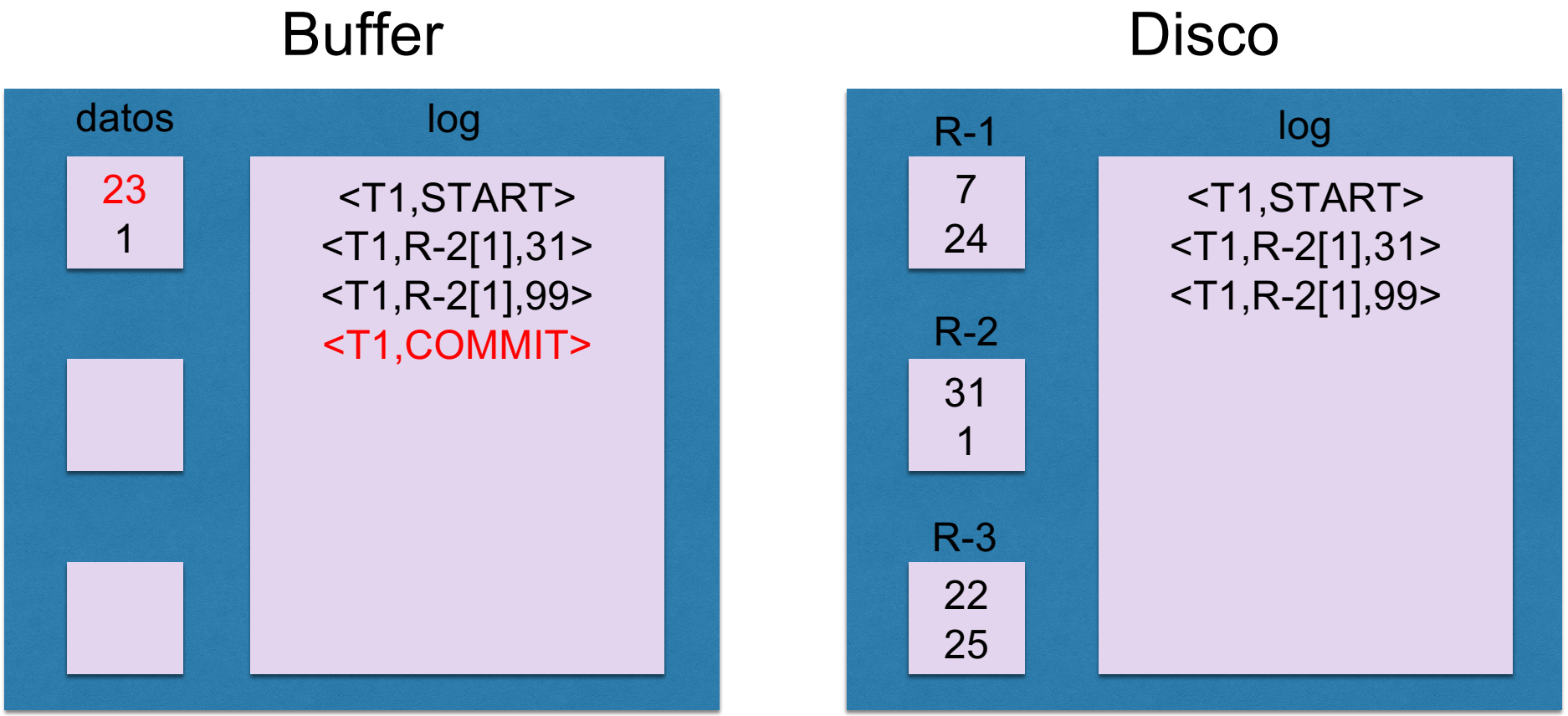
Vamos al disco!



Undo Logging – en la BD

DBMS

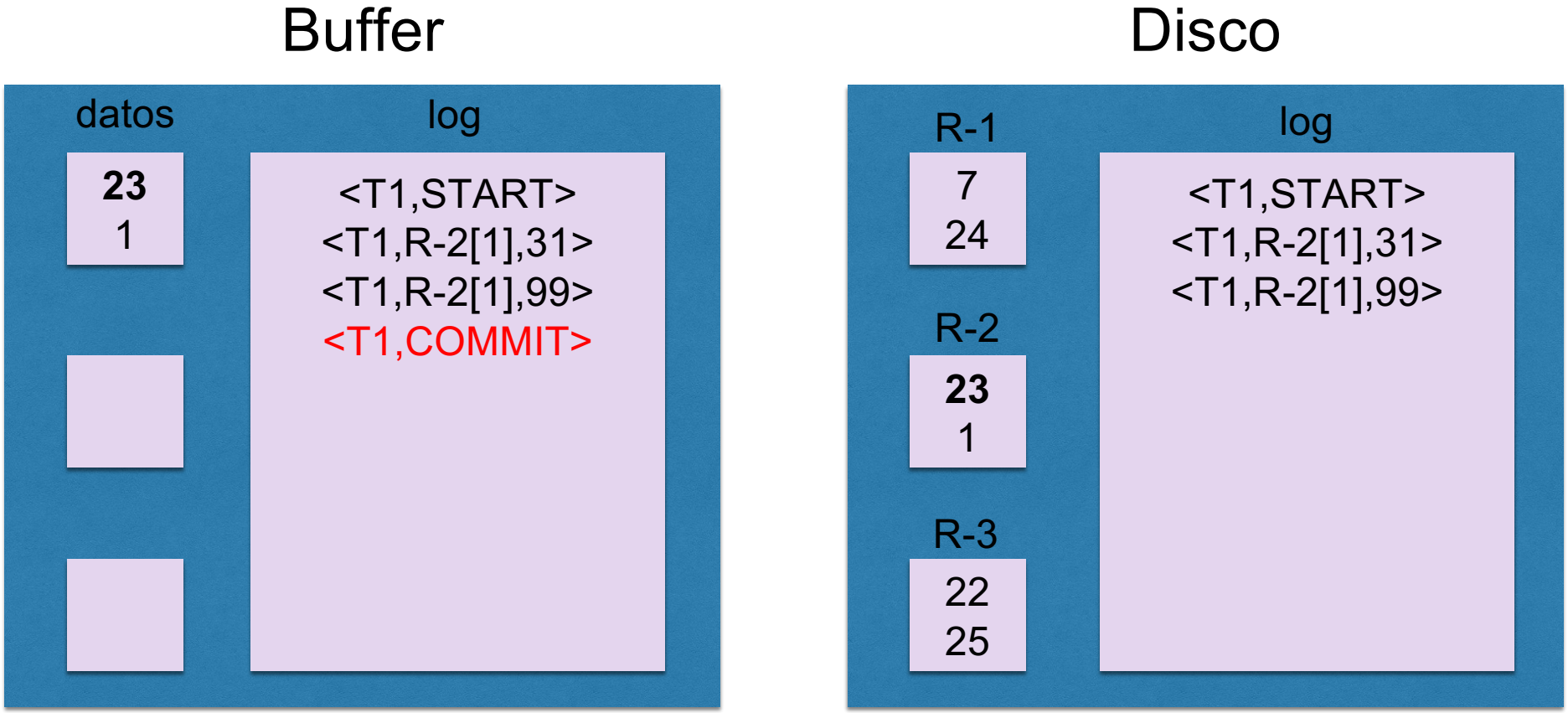
Vamos al disco!



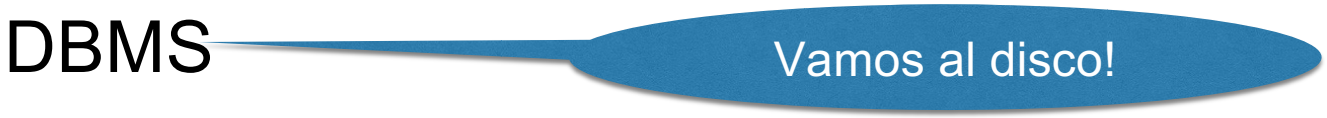
Undo Logging – en la BD

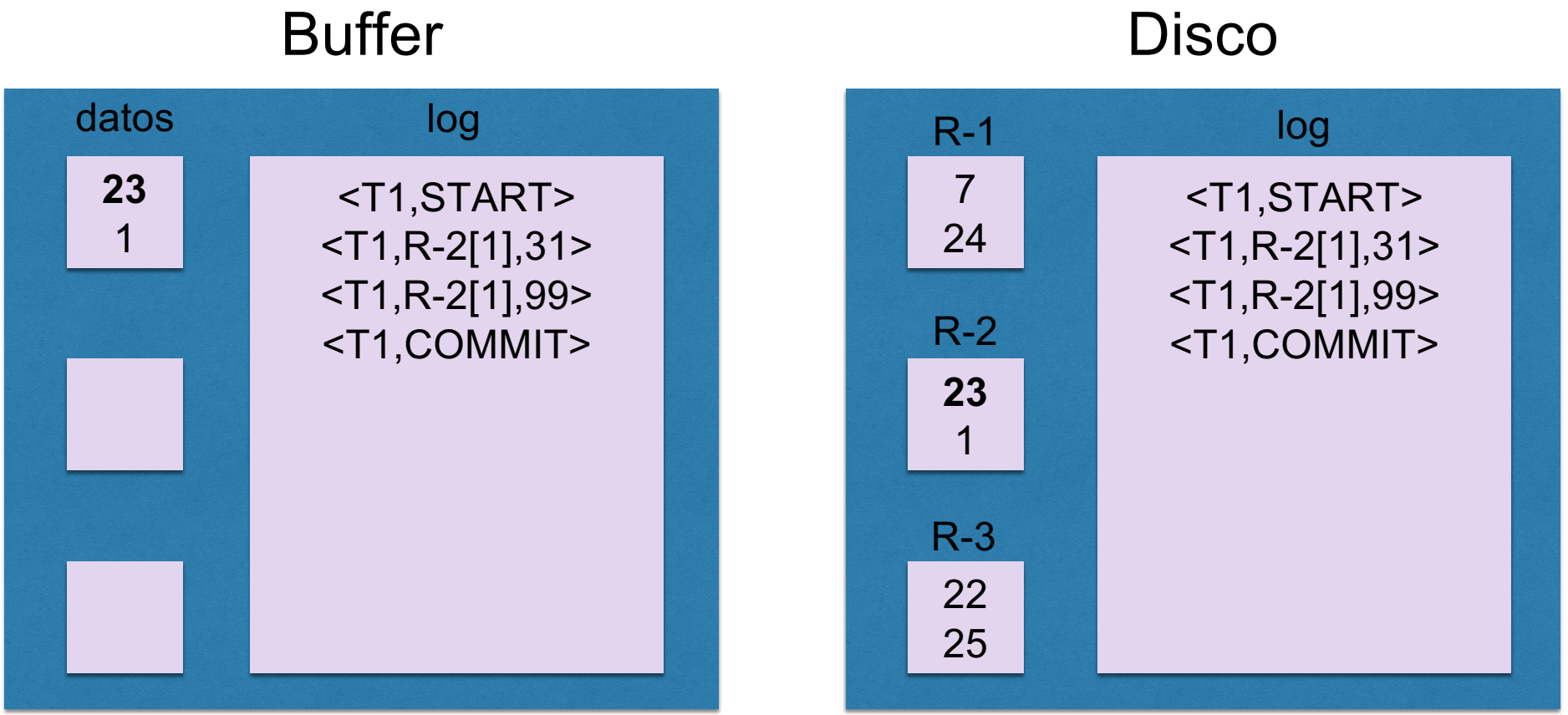
DBMS

Vamos al disco!

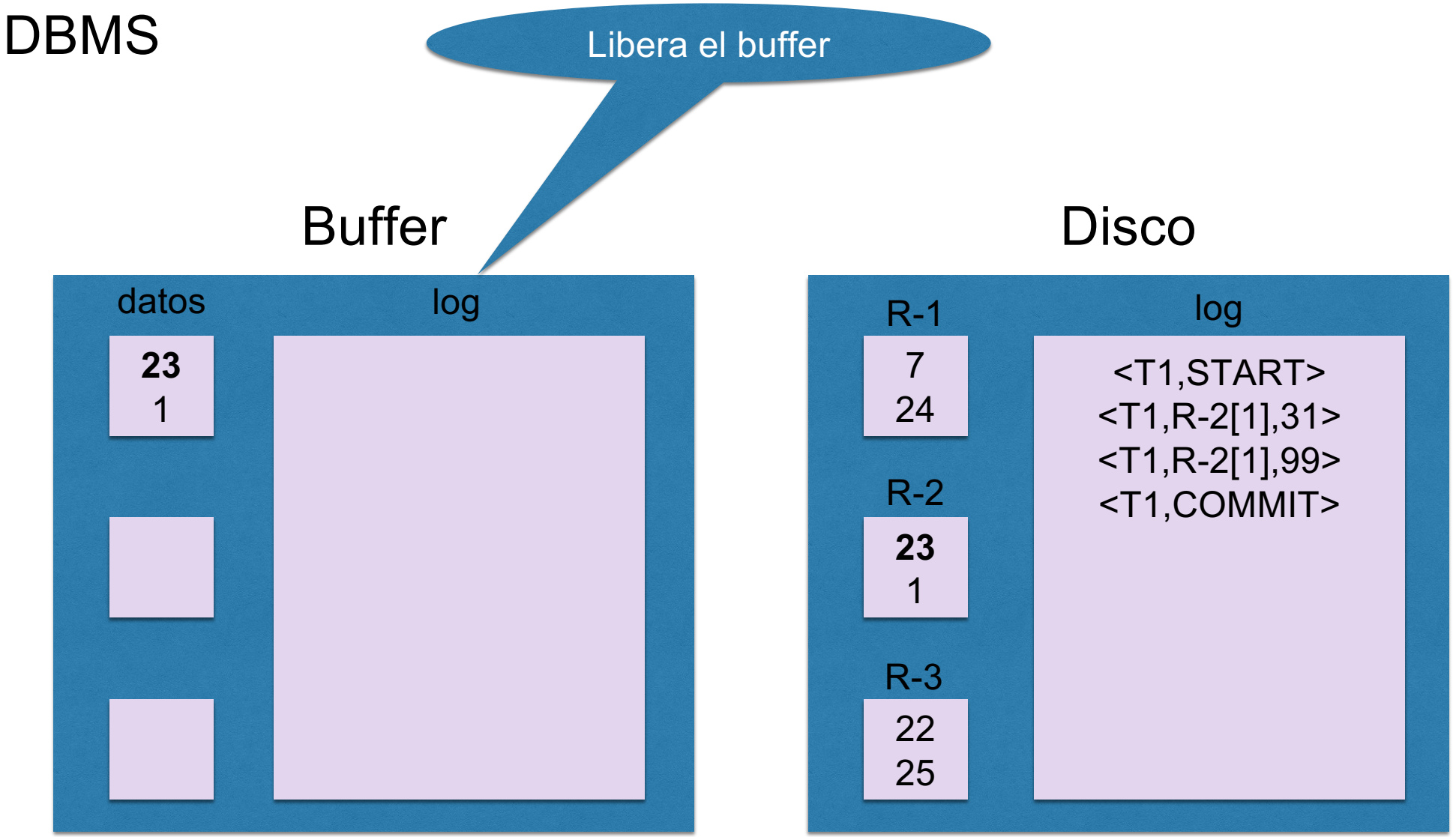


Undo Logging – en la BD

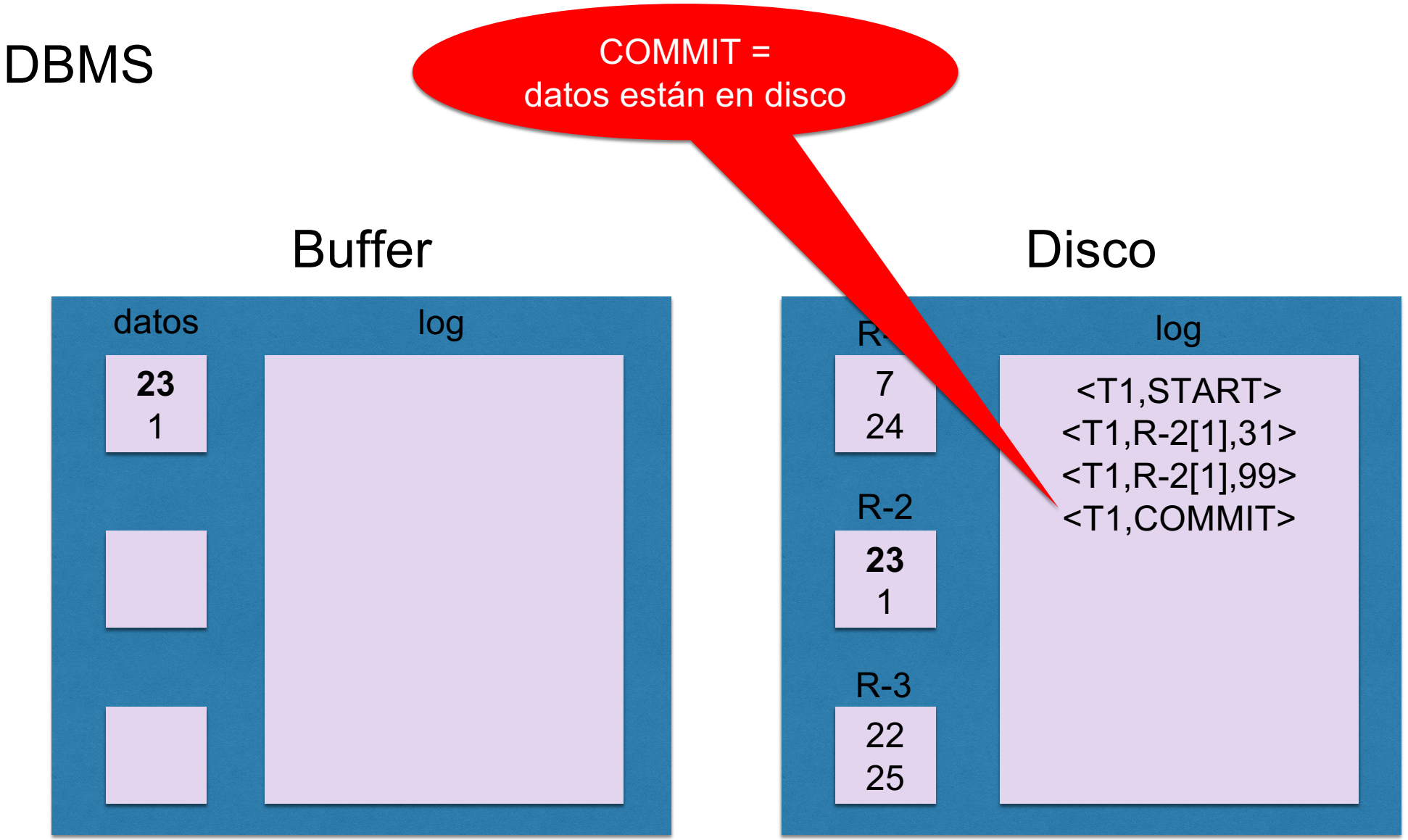




Undo Logging – en la BD



Undo Logging – en la BD



Recuperación con Undo Logging

Detectando fallas en el *log*:

... <START T> ... <COMMIT T> ...



Veamos un caso donde ocurre una falla…

Ejemplo de log de transacción abortada

Start transaction T1

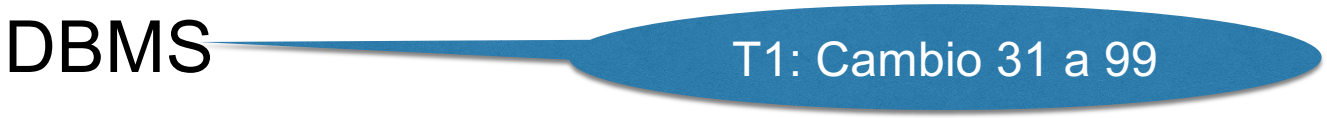
Reemplace el valor 31 de R.a por 99

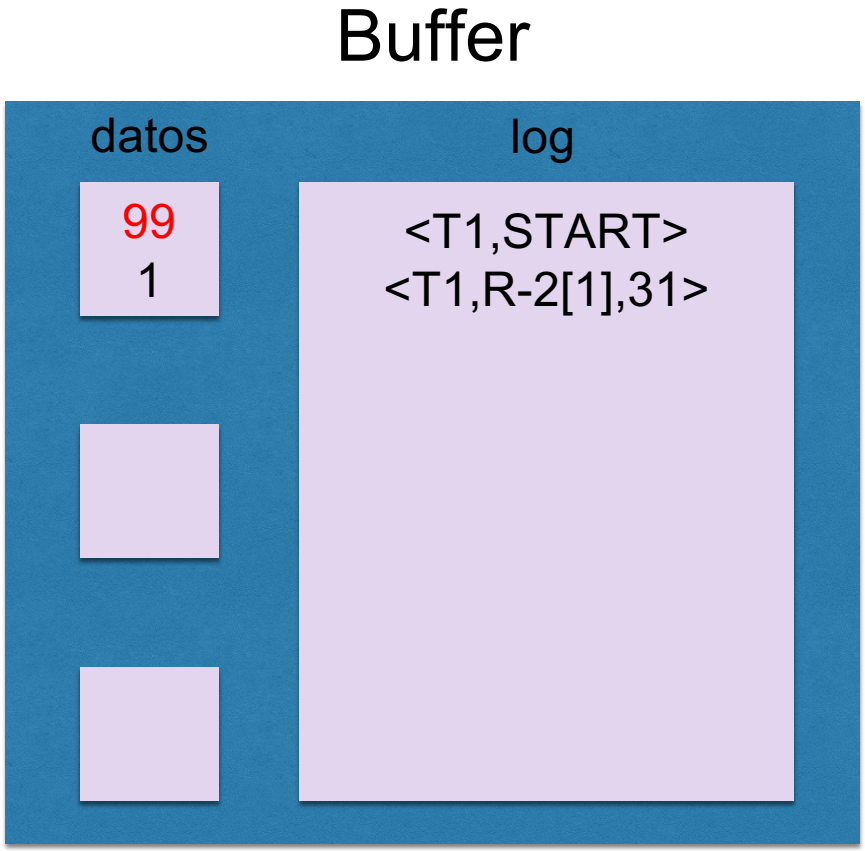
T1 se cae

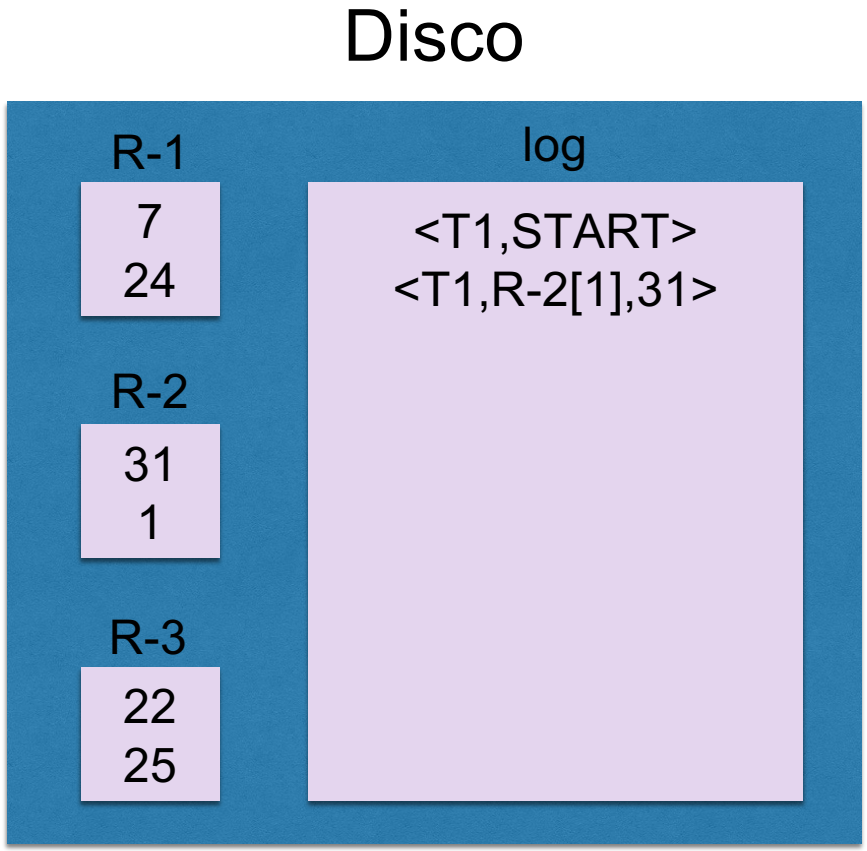
Reemplace el valor 99 de R.a por 23

Commit T1

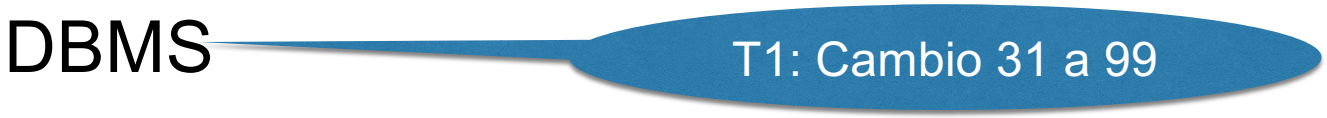
Undo Logging – en la BD

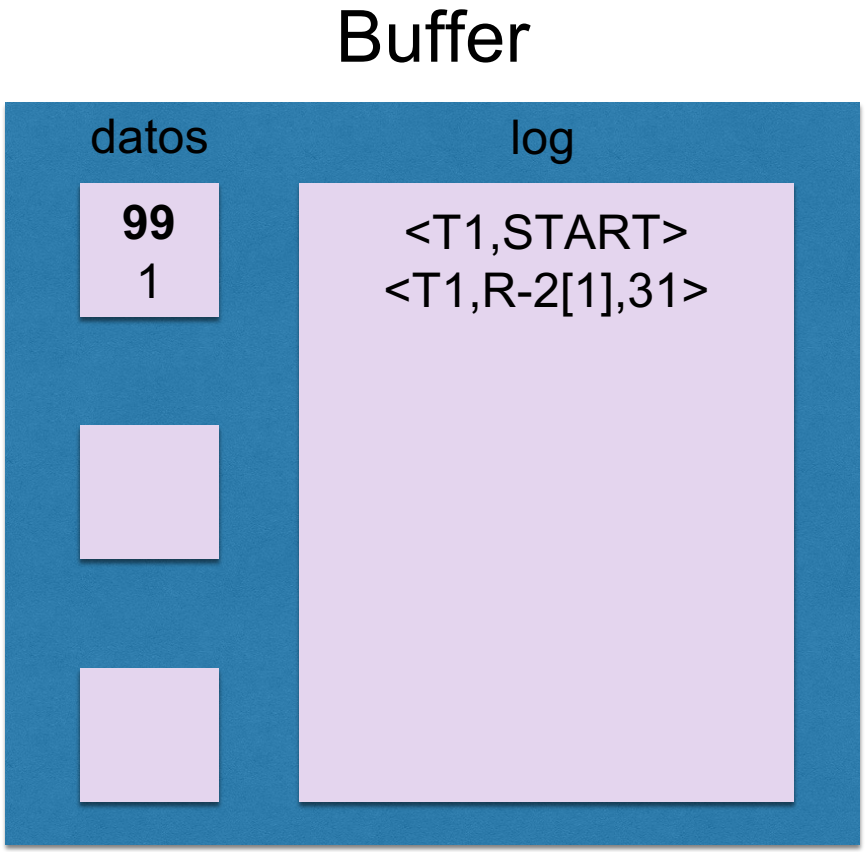


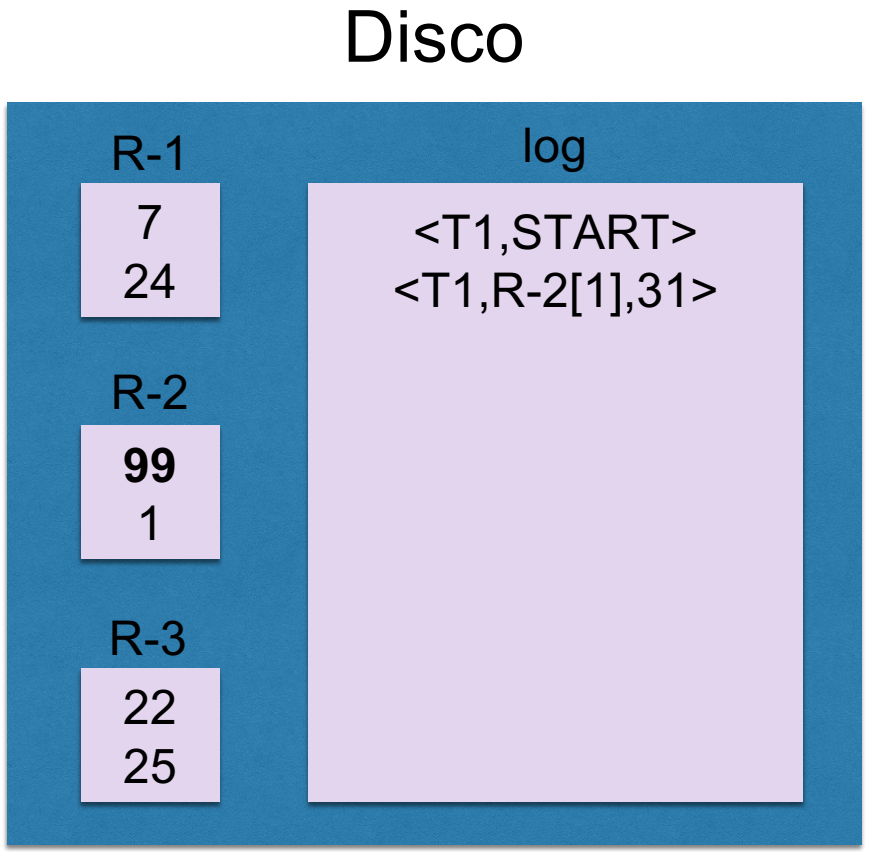




Undo Logging – en la BD

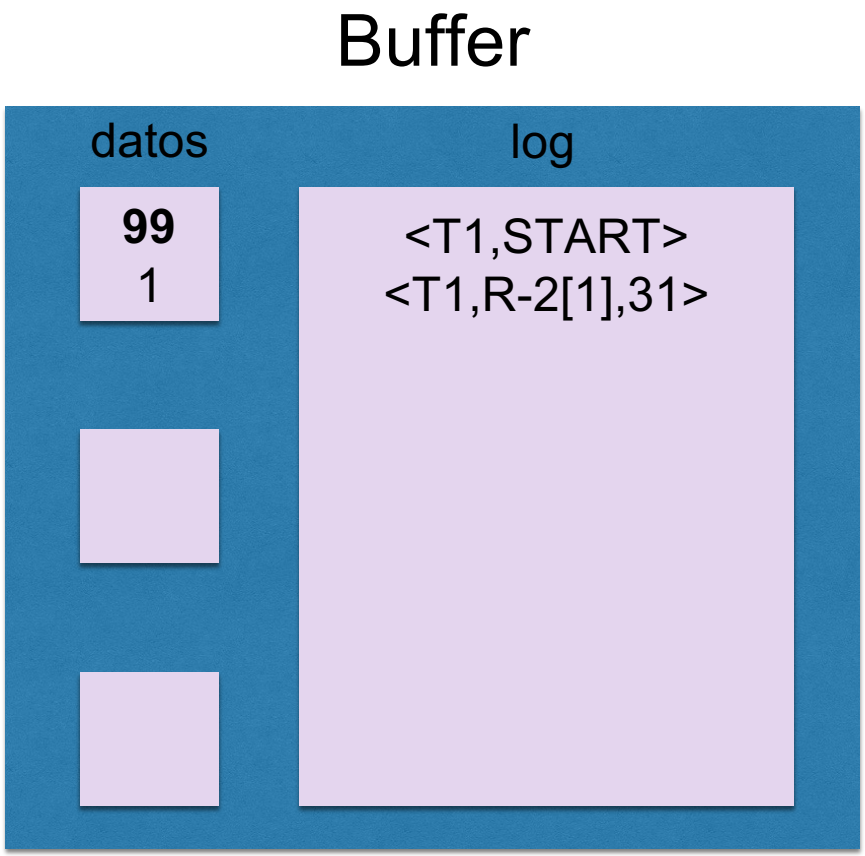


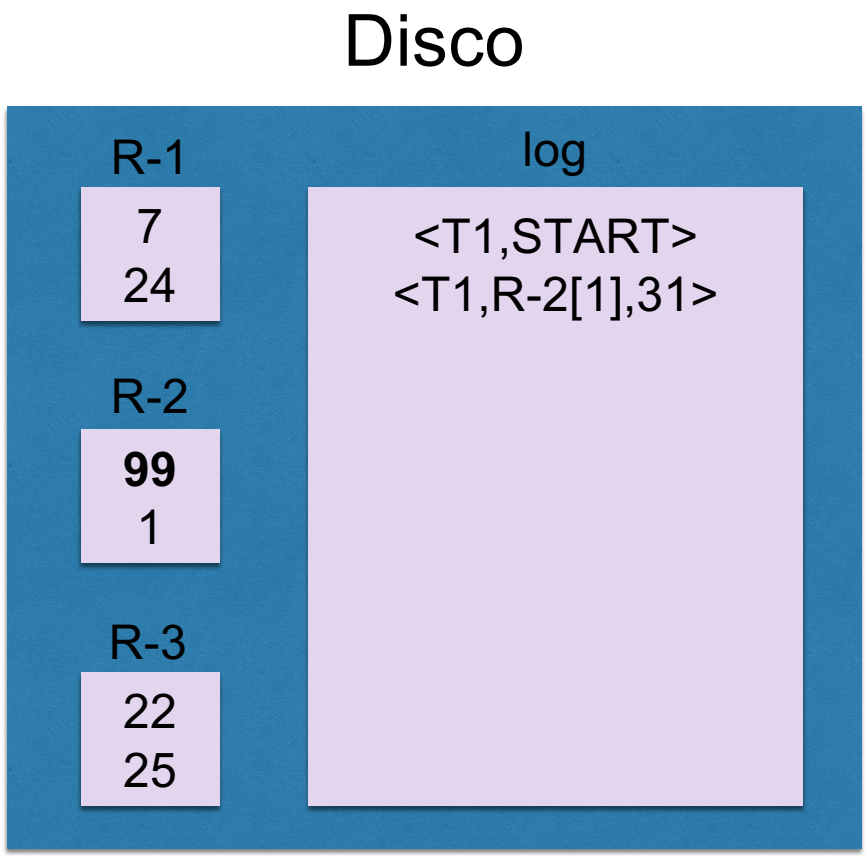




Undo Logging – en la BD

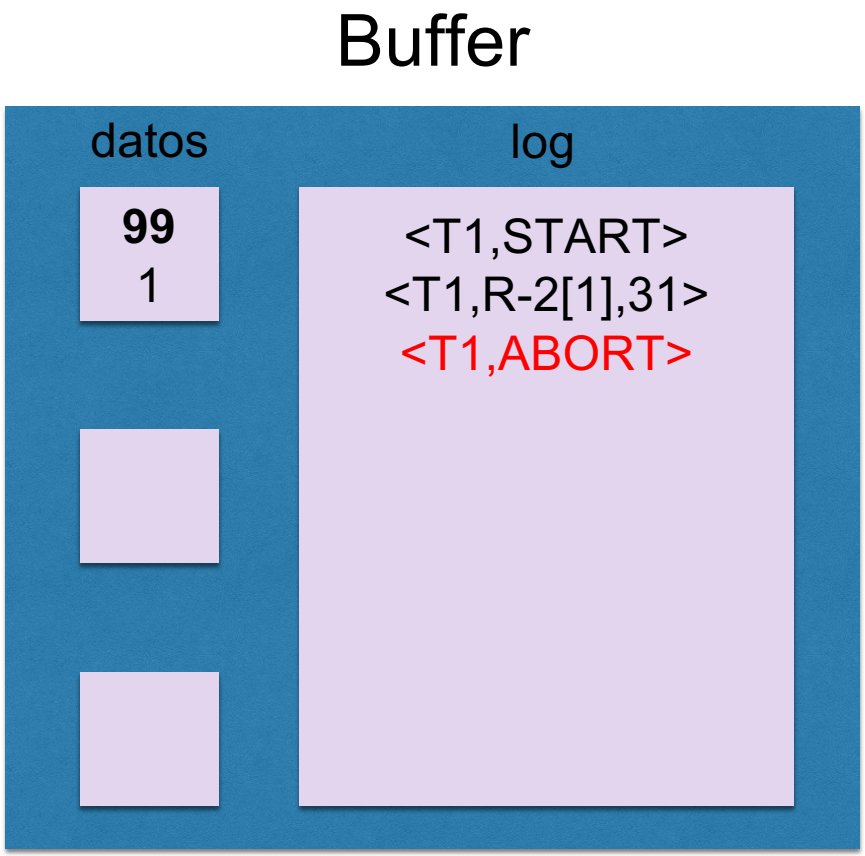


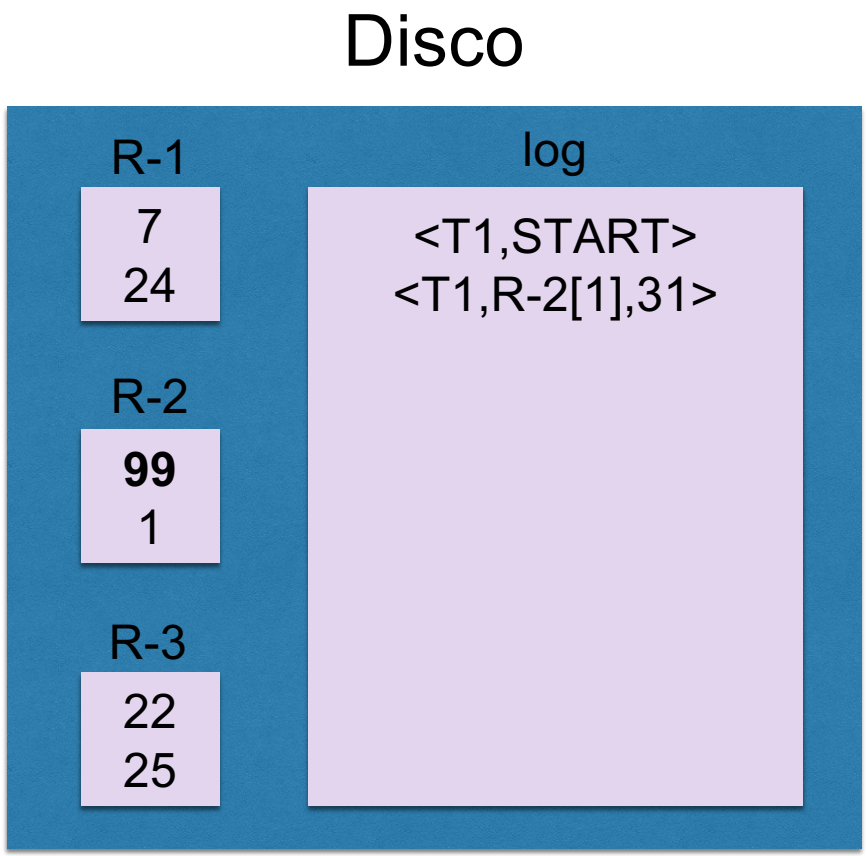




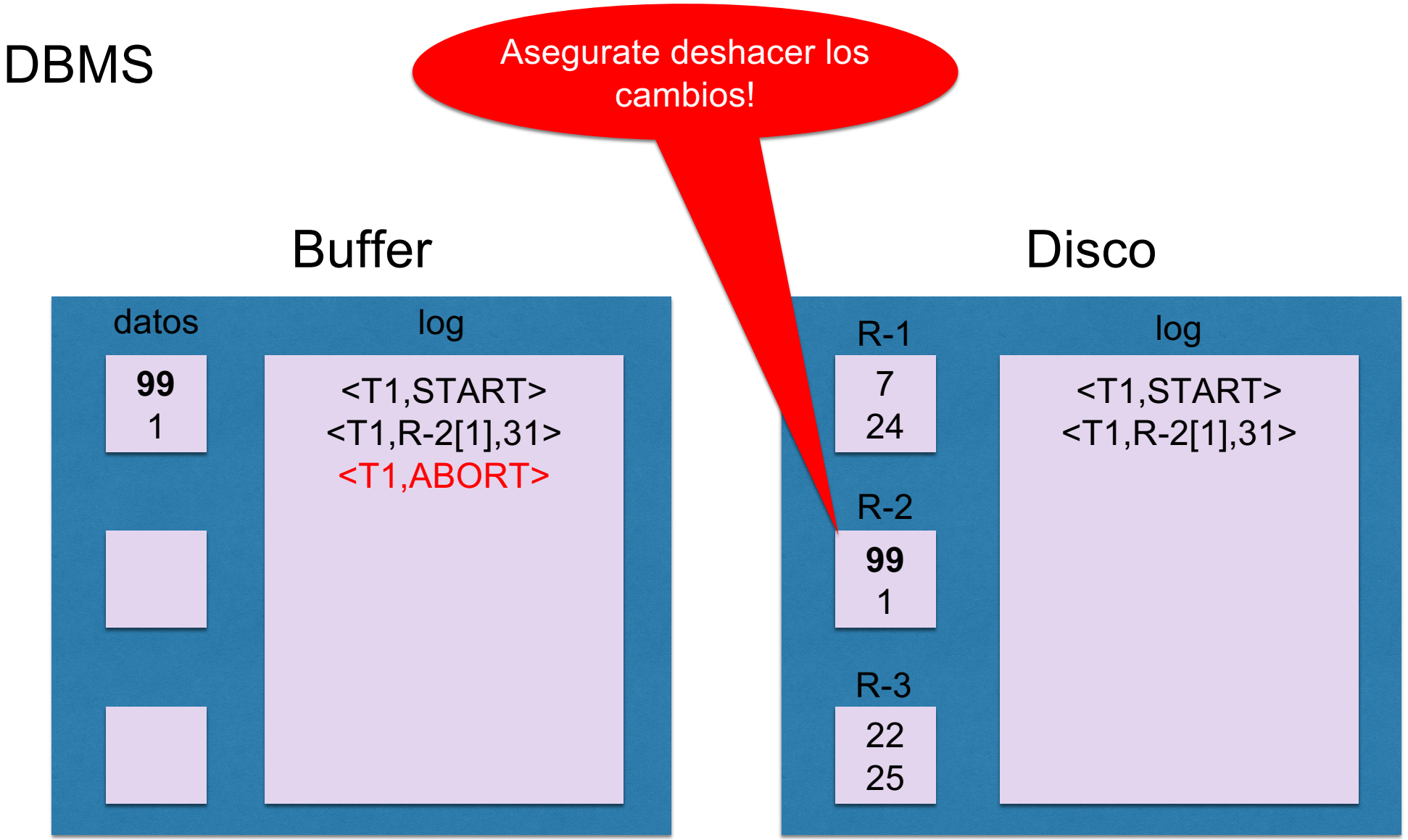
Undo Logging – en la BD



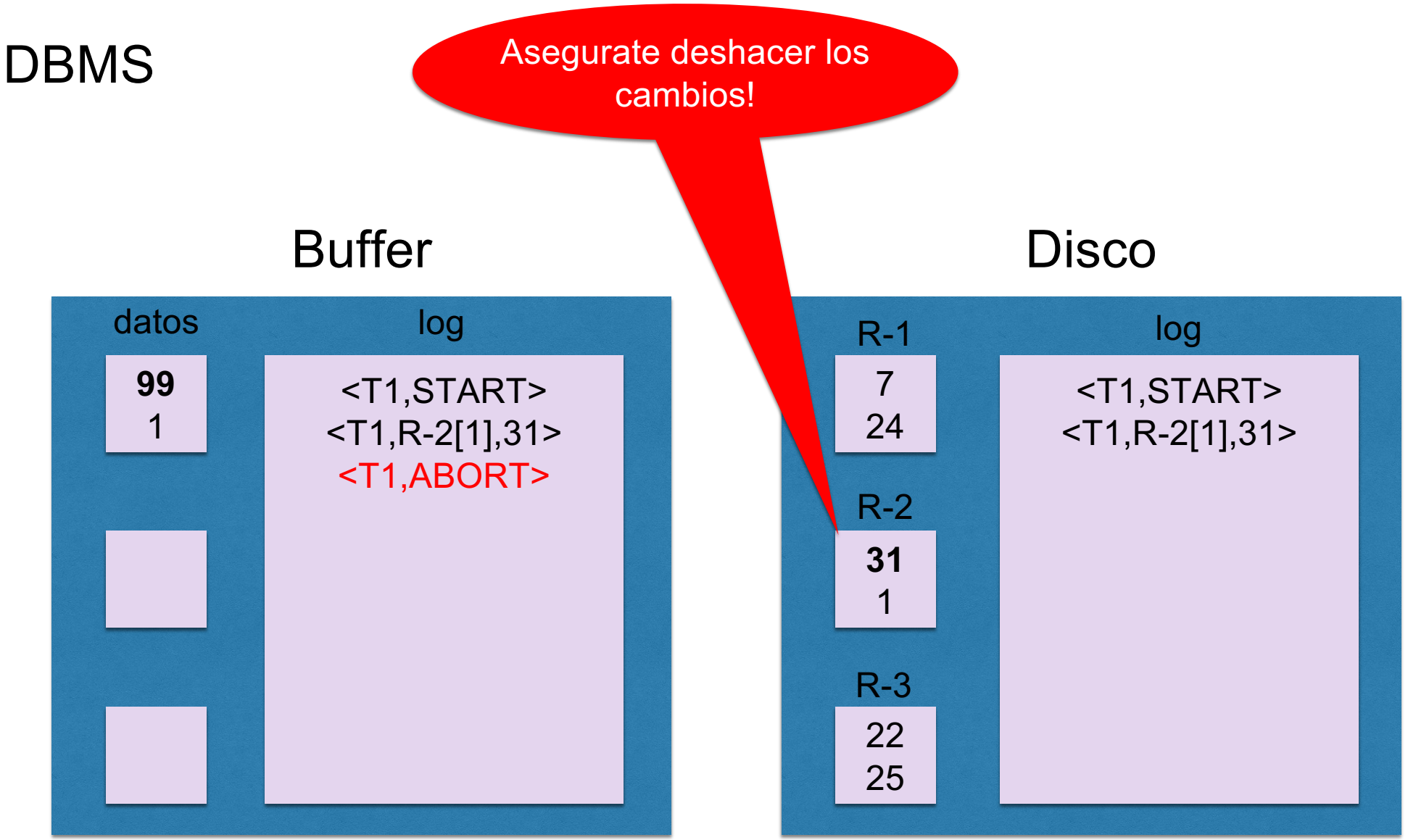




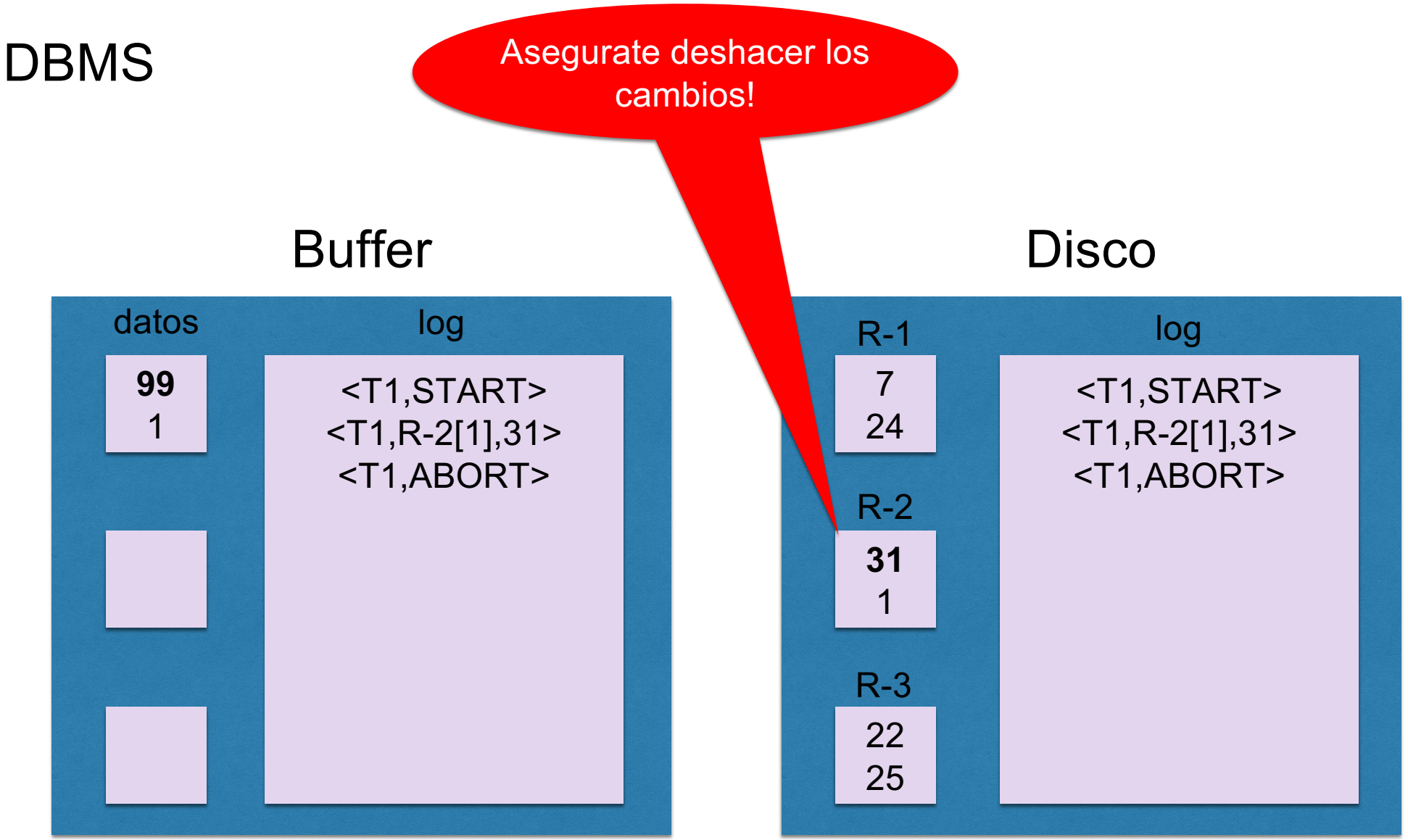
Undo Logging – en la BD



Undo Logging – en la BD



Undo Logging – en la BD



Recuperación con Undo Logging

Detectando fallas en el *log*:

... <START T> ... <ABORT T> ...



Recuperación con Undo Logging

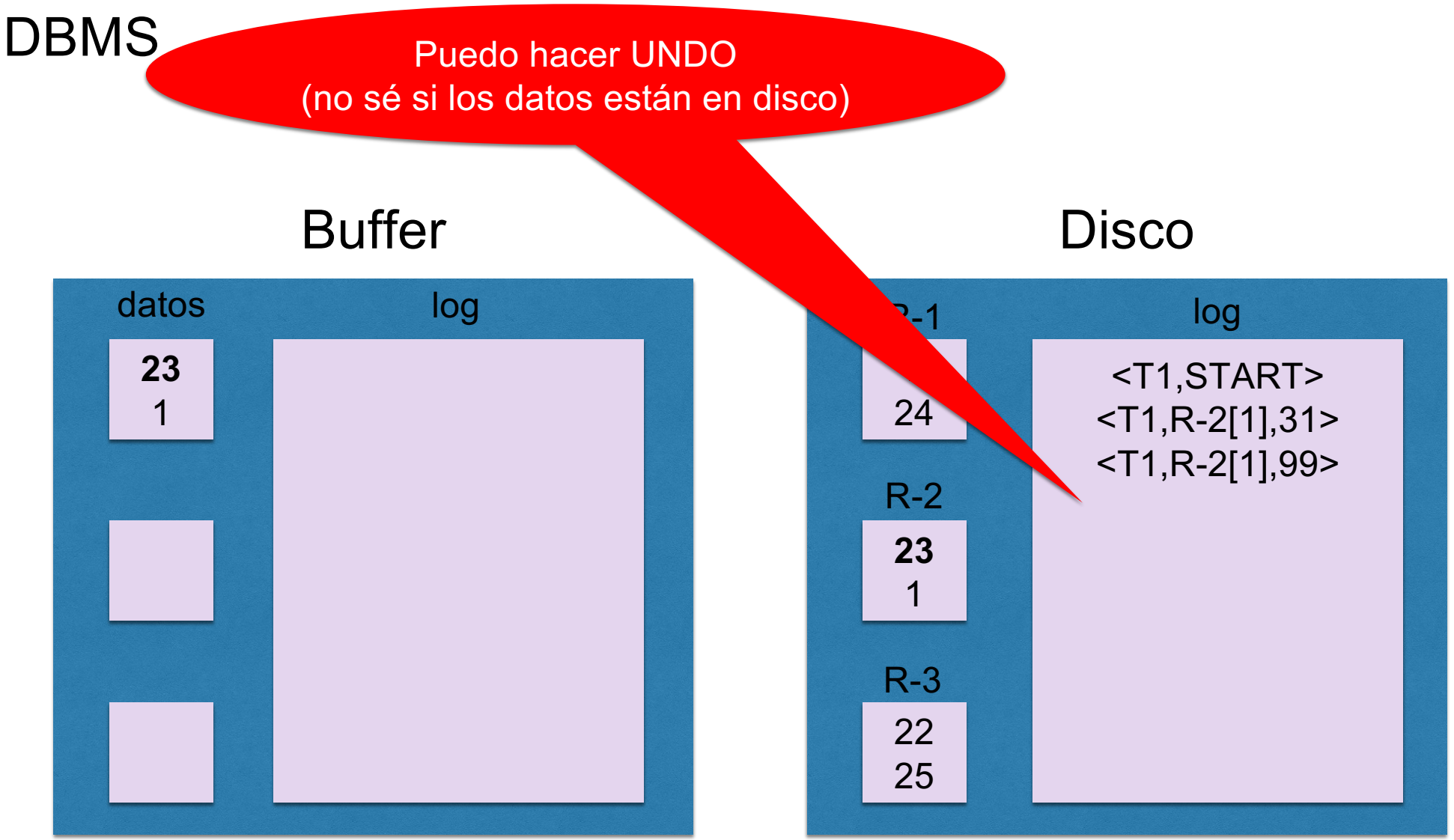
Detectando fallas en el *log*:

... <START T> ...



¡Encontramos una falla! Recuperemos con Undo Logging:

Undo Logging – en la BD



Recuperación con Undo Logging

Supongamos que mientras usamos nuestro sistema, se apagó de forma imprevista

Leyendo el *log* podemos hacer que la base de datos quede en un estado consistente

Recovery

Algoritmo para un *Undo Logging*

Procesamos el log desde el final hasta el principio:

- Si leo <COMMIT **T**>, marco **T** como realizada
- Si leo <ABORT **T**>, marco **T** como realizada
- Si leo <**T**, X, t>, debo restituir X := t en disco, si no fue realizada.
- Si leo <START **T**>, lo ignoro

Recovery

Algoritmo para un *Undo Logging*

- ¿Hasta dónde tenemos que leer el *log*?
- ¿Qué pasa si el sistema falla en plena recuperación?
- ¿Cómo trucamos el *log*?

¡Con **Checkpoints**!

Recovery

Uso de *Checkpoints*

Utilizamos *checkpoints* para no tener que leer el *log* entero y para manejar las fallas mientras se hace *recovery*

Recovery

Uso de *Checkpoints*

- Dejamos de escribir transacciones
- Esperamos a que las transacciones actuales terminen
- Se guarda el *log* en disco
- Escribimos <CKPT> y se guarda en disco
- Se reanudan las transacciones

Recovery

Uso de *Checkpoints*

Ahora hacemos *recovery* hasta leer un <CKPT>

Pero hay un gran problema: es prácticamente necesario apagar el sistema para guardar un *checkpoint.*

Para solucionarlo, usaremos

Nonquiescent Checkpoints

Recovery

Uso de *Nonquiescent Checkpoints*

Nonquiescent Checkpoints son un tipo de *checkpoint* que no requiere "apagar" el sistema

Recovery

Uso de *Nonquiescent Checkpoints*

- Escribimos un *log* <START CKPT (T_1, ..., T_n)>, donde T_1, ..., T_n son transacciones activas
- Esperamos hasta que T_1, ..., T_n terminen, sin restringir nuevas transacciones
- Cuando T_1, ..., T_n hayan terminado, escribimos <END CKPT>

Undo Recovery

Uso de *Nonquiescent Checkpoints*

- Avanzamos desde el final al inicio
- Si encontramos un <END CKPT>, hacemos *undo* de todo lo que haya *empezado* después del inicio del *checkpoint*
- Si encontramos un <START CKPT (T_1, ..., T_n)> sin su <END CKPT>, debemos analizar el log desde el inicio de la transacción más antigua entre T_1, ..., T_n

Ejemplo

Uso de *Checkpoints* en *Undo Logging*

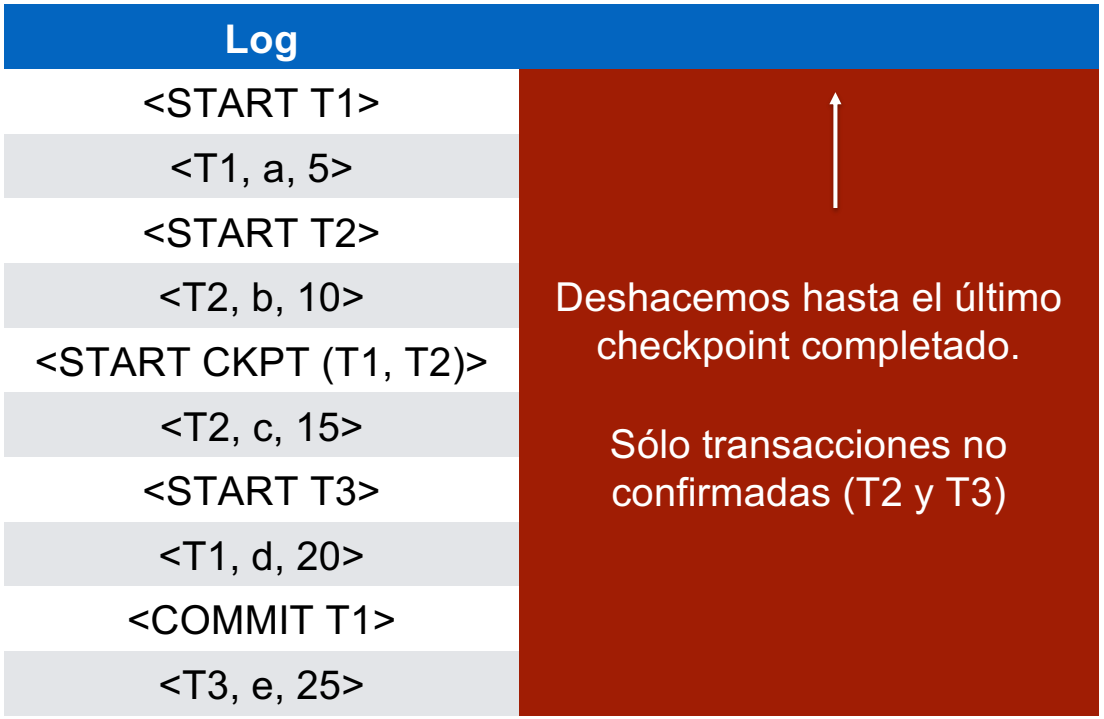
Considere este *log* después de una falla:

Log	
<START T1>	Podemos truncar esta parte
<T1, a, 5>	
<START T2>	
<T2, b, 10>	
<START CKPT (T1, T2)>	T1 y T2 activas
<T2, c, 15>	Deshacemos solo esta parte
<START T3>	
<T1, d, 20>	
<COMMIT T1>	Noten que T3 partió después del checkpoint
<T3, e, 25>	
<COMMIT T2>	
<END CKPT>	

Ejemplo

Uso de *Checkpoints* en *Undo Logging*

Ahora considere este *log* después de una falla:



Undo Logging

Pero tenemos un problema: con Undo logging no es posible hacer COMMIT antes de almacenar los datos en disco.

Por lo tanto las transacciones se toman más tiempo en terminar.

¿Existe alguna manera de hacer commit antes de almacenar los datos en el disco? ¡Si!, con **Redo Logging.**

Redo Logging

Redo Logging

Los *logs* son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T**, X, v> donde v es el valor **nuevo** de X
- <**T**,END>

Redo Logging - Reglas

Regla 1: Antes de modificar cualquier elemento X en disco, es necesario que todos los *logs* estén almacenados en disco, *incluido* el COMMIT

Esto es al revés respecto a *Undo Logging*

Redo Logging

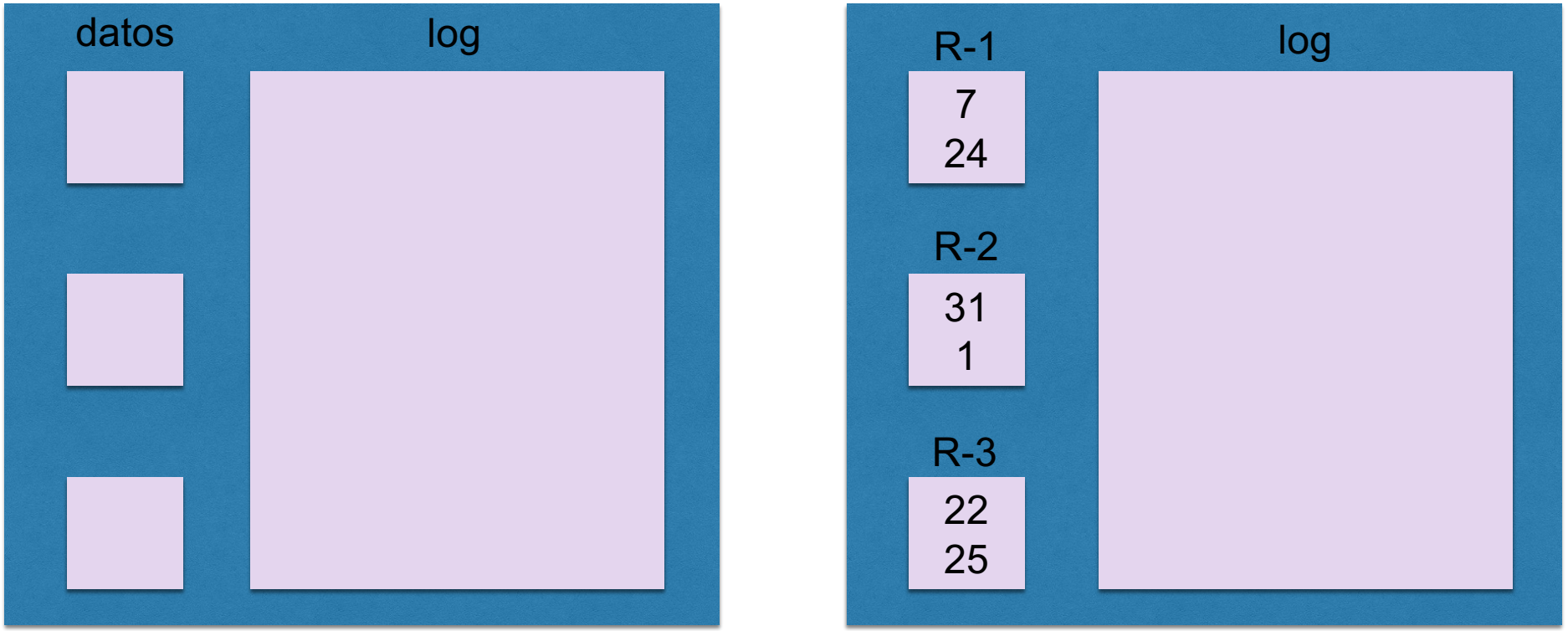
En resumen:

- Escribir el log <**T**, X, v>
- Escribir <COMMIT **T**>
- Escribir los datos en disco
- Escribir <**T**,END> en log (en disco)

Redo Logging – en la BD

DBMS

T1: voy a empezar

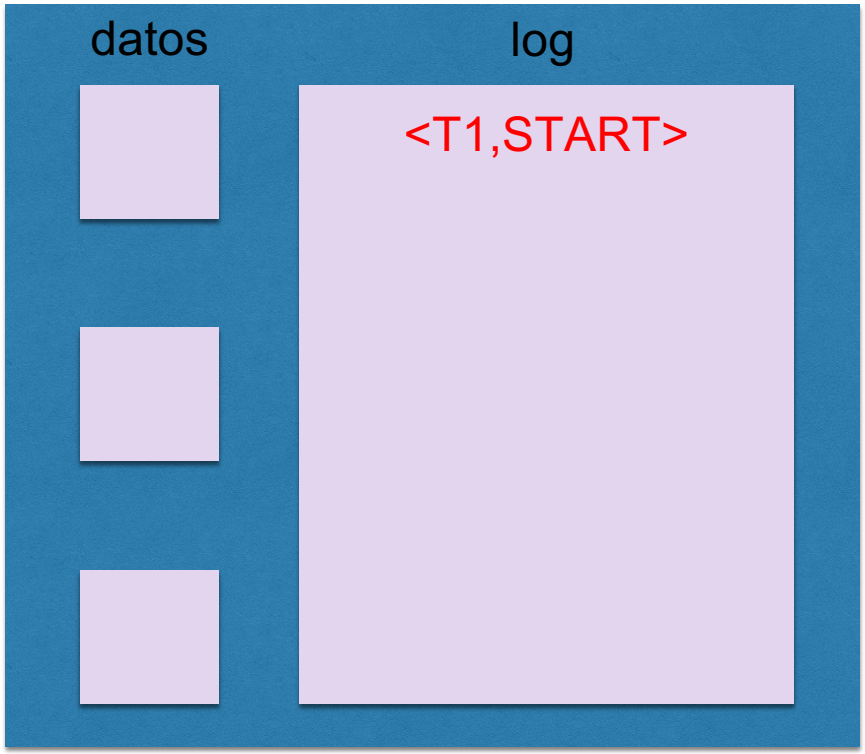


Redo Logging – en la BD

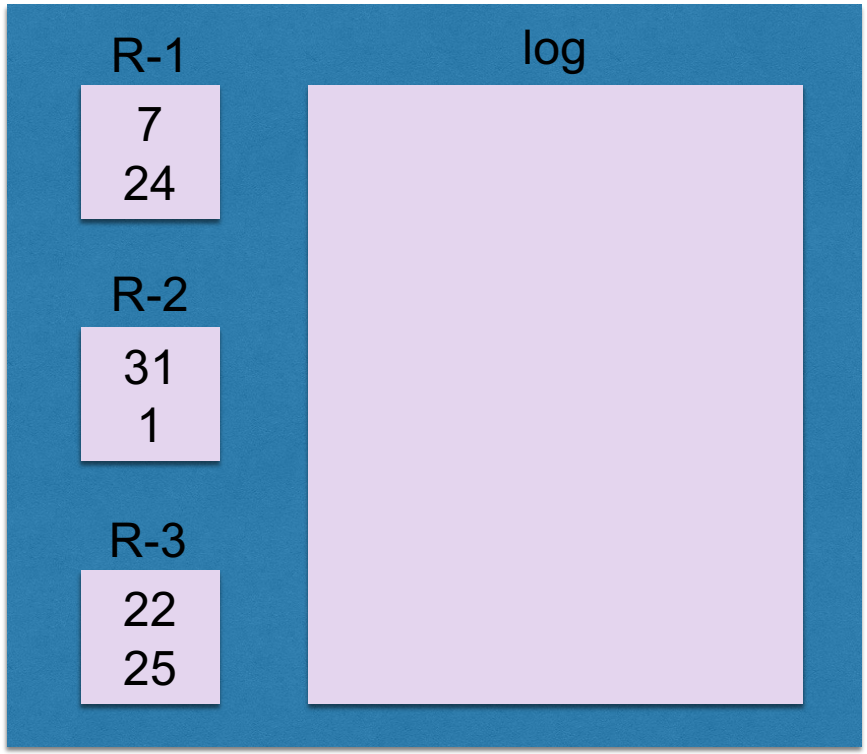
DBMS

T1: voy a empezar

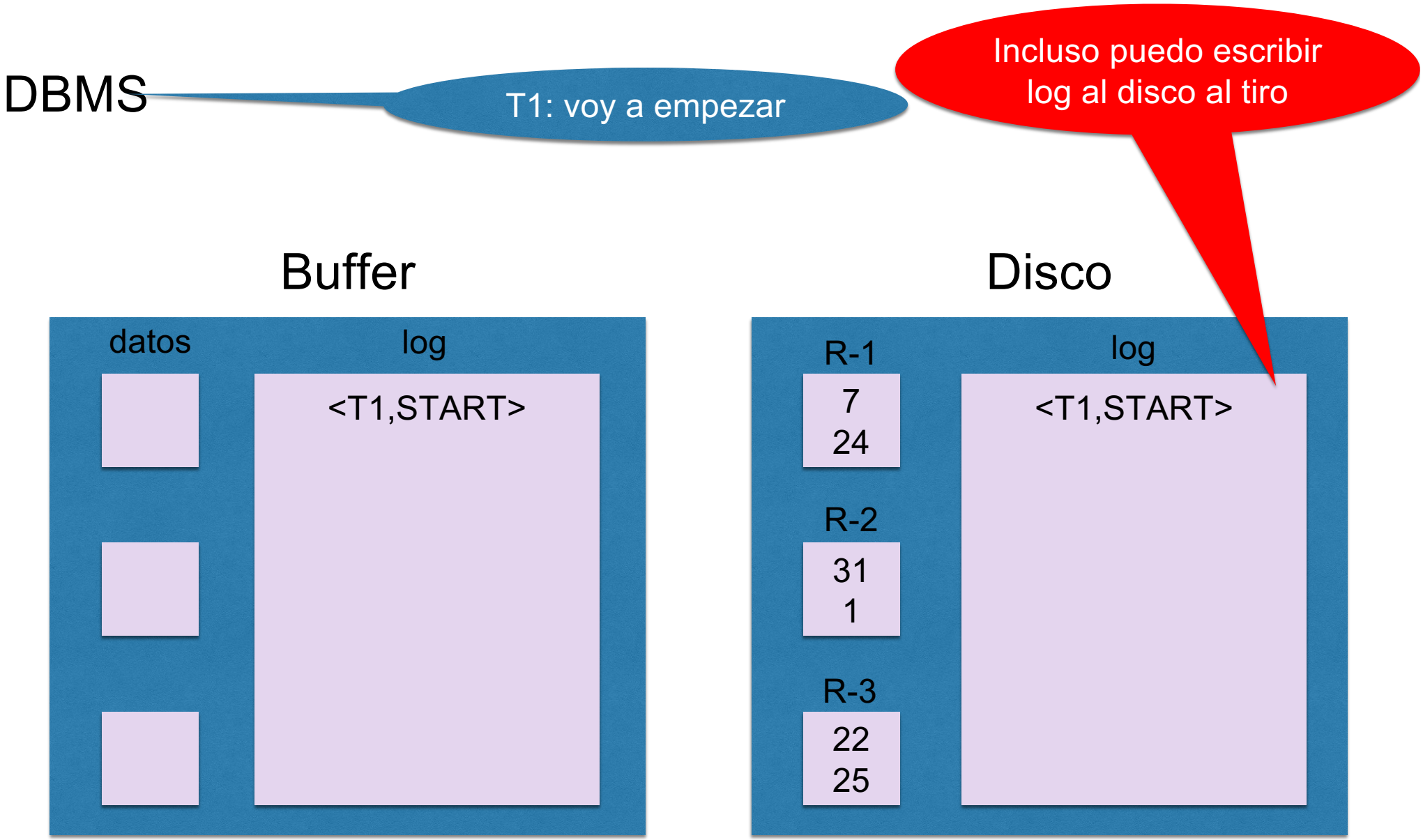
Buffer



Disco



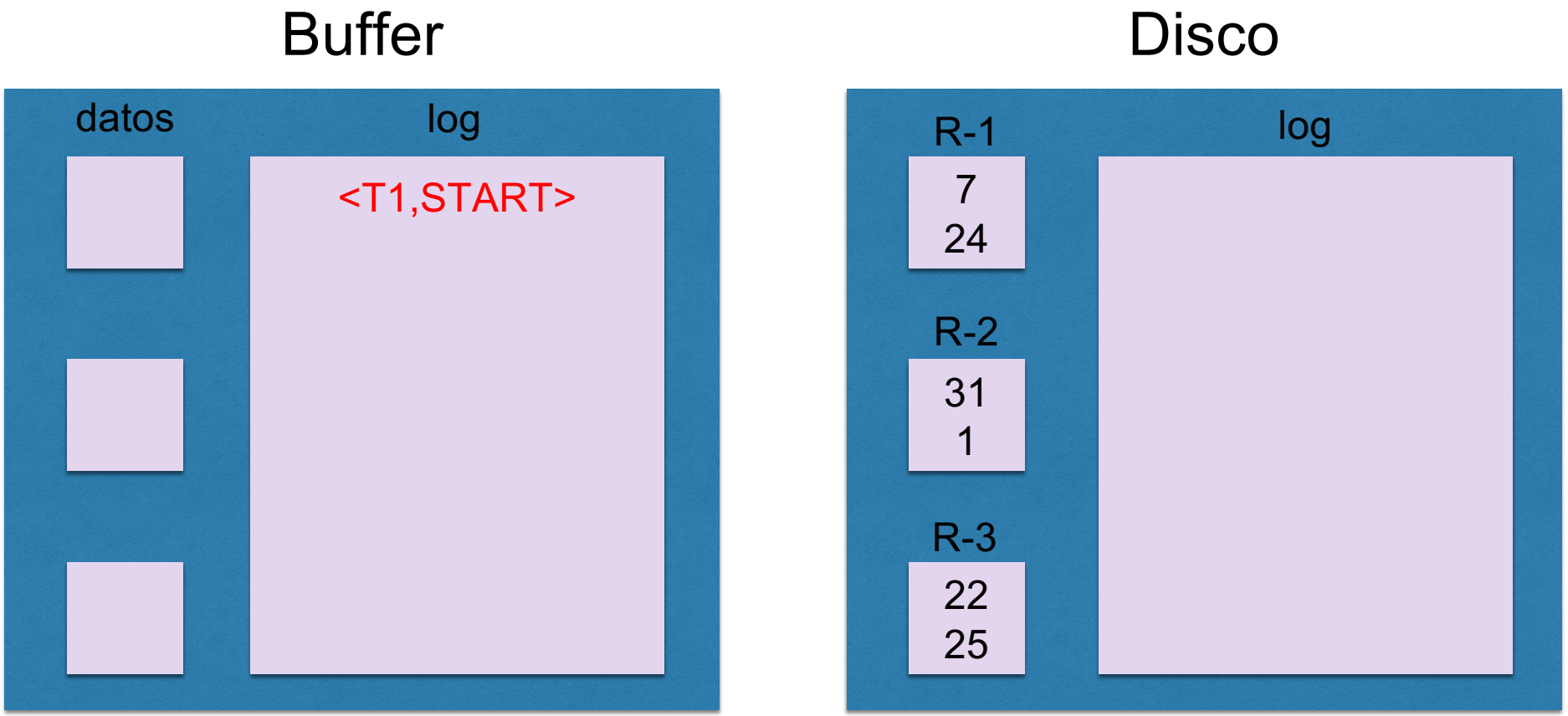
Redo Logging – en la BD



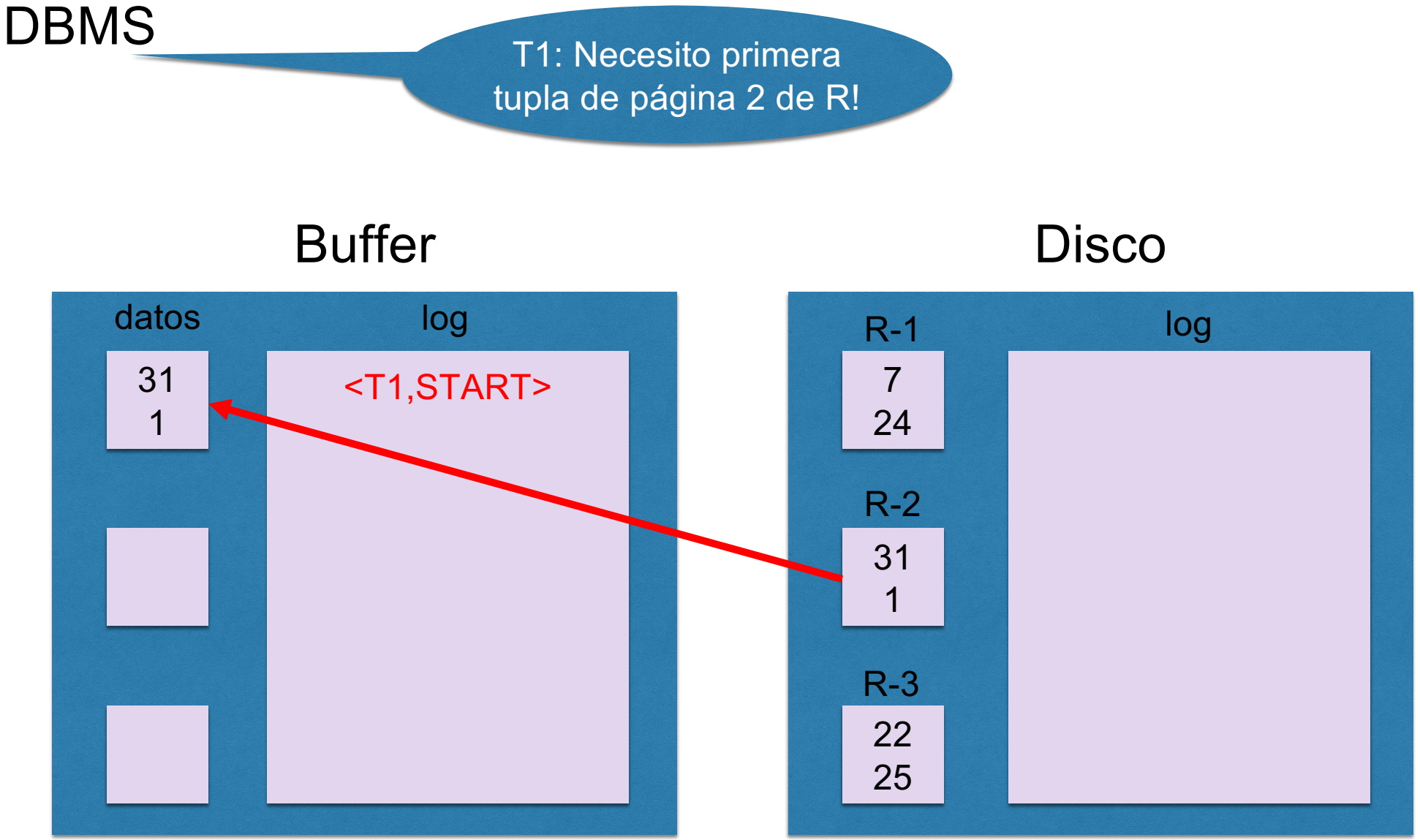
Redo Logging – en la BD

DBMS

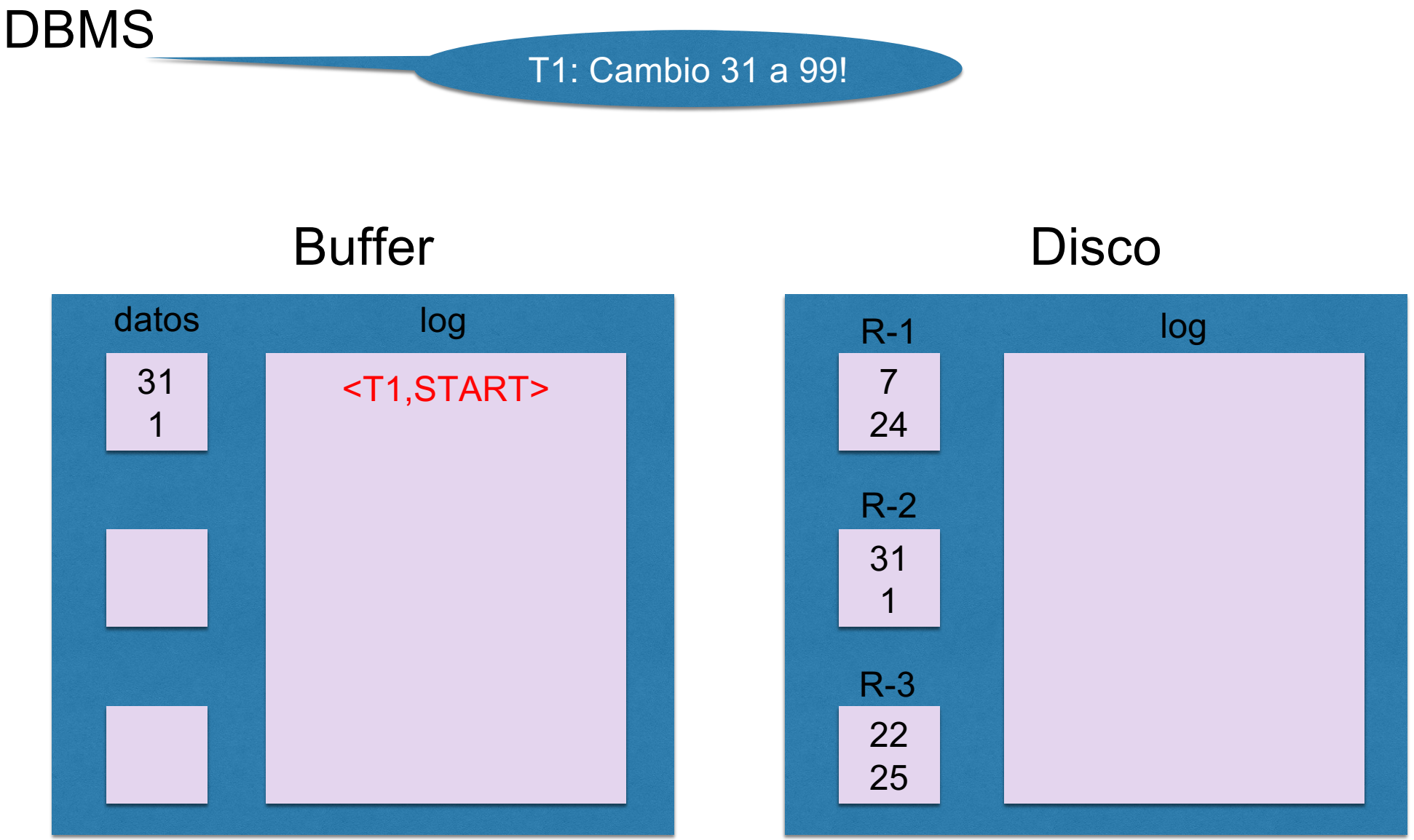
T1: Necesito primera tupla de página 2 de R!



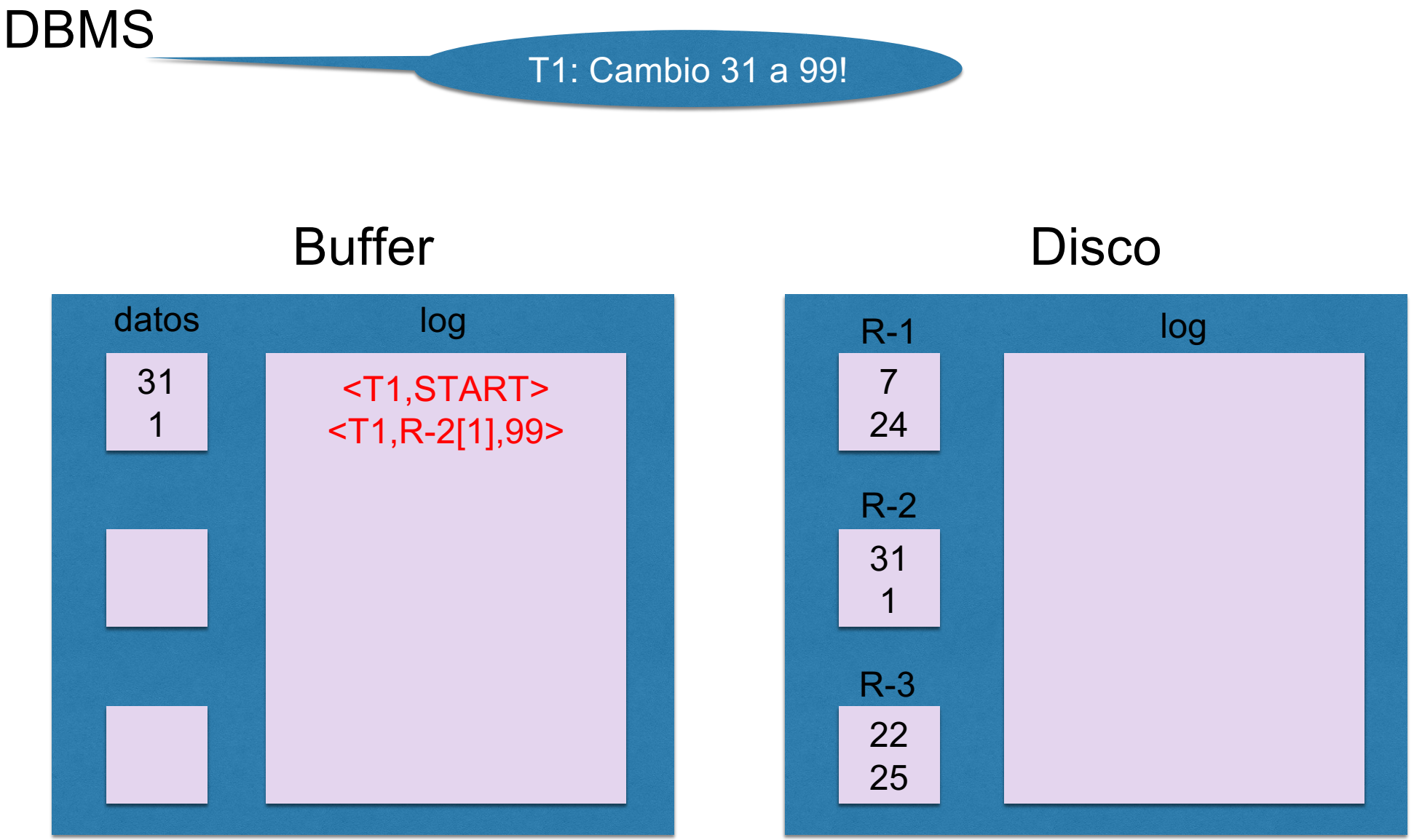
Redo Logging – en la BD



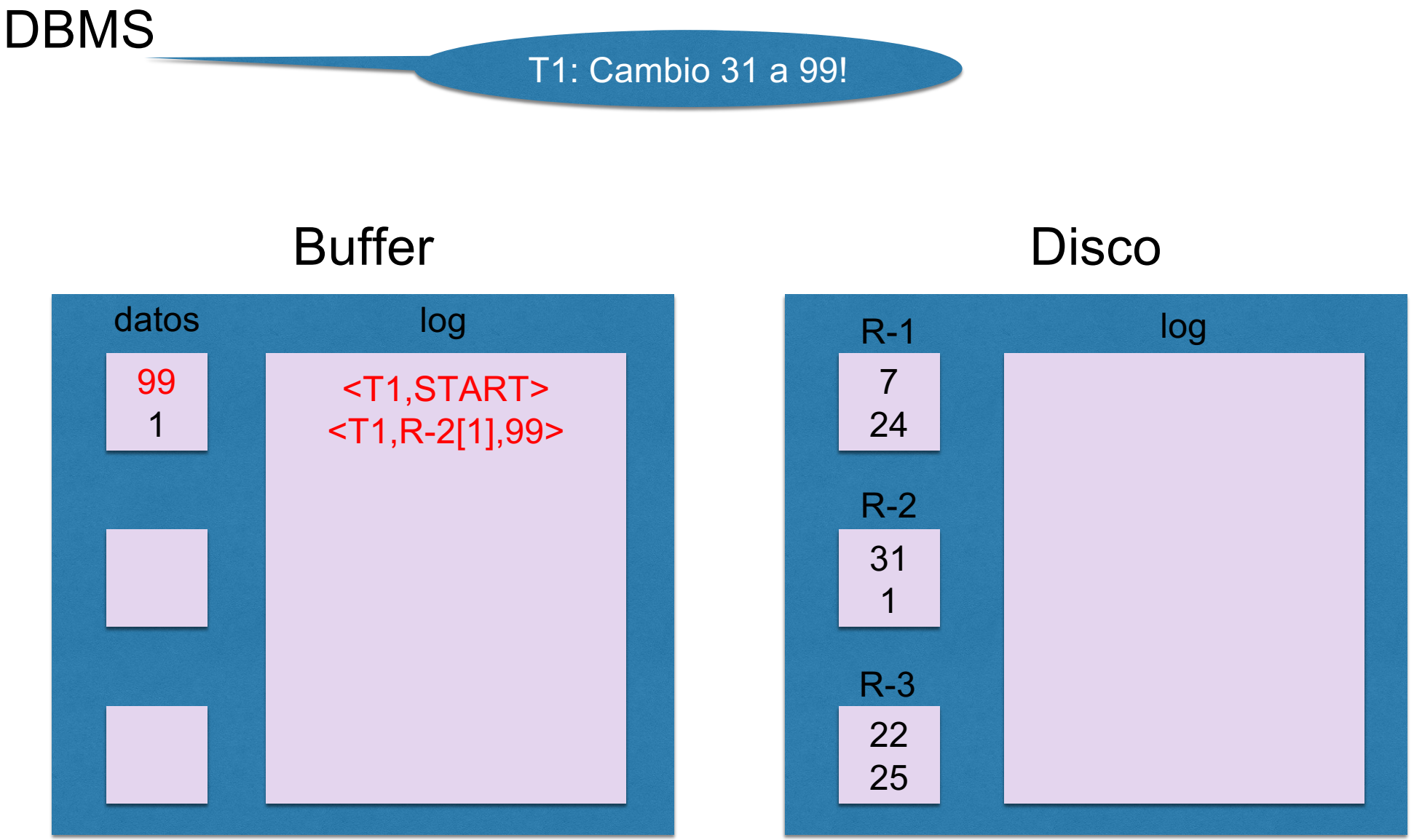
Redo Logging – en la BD



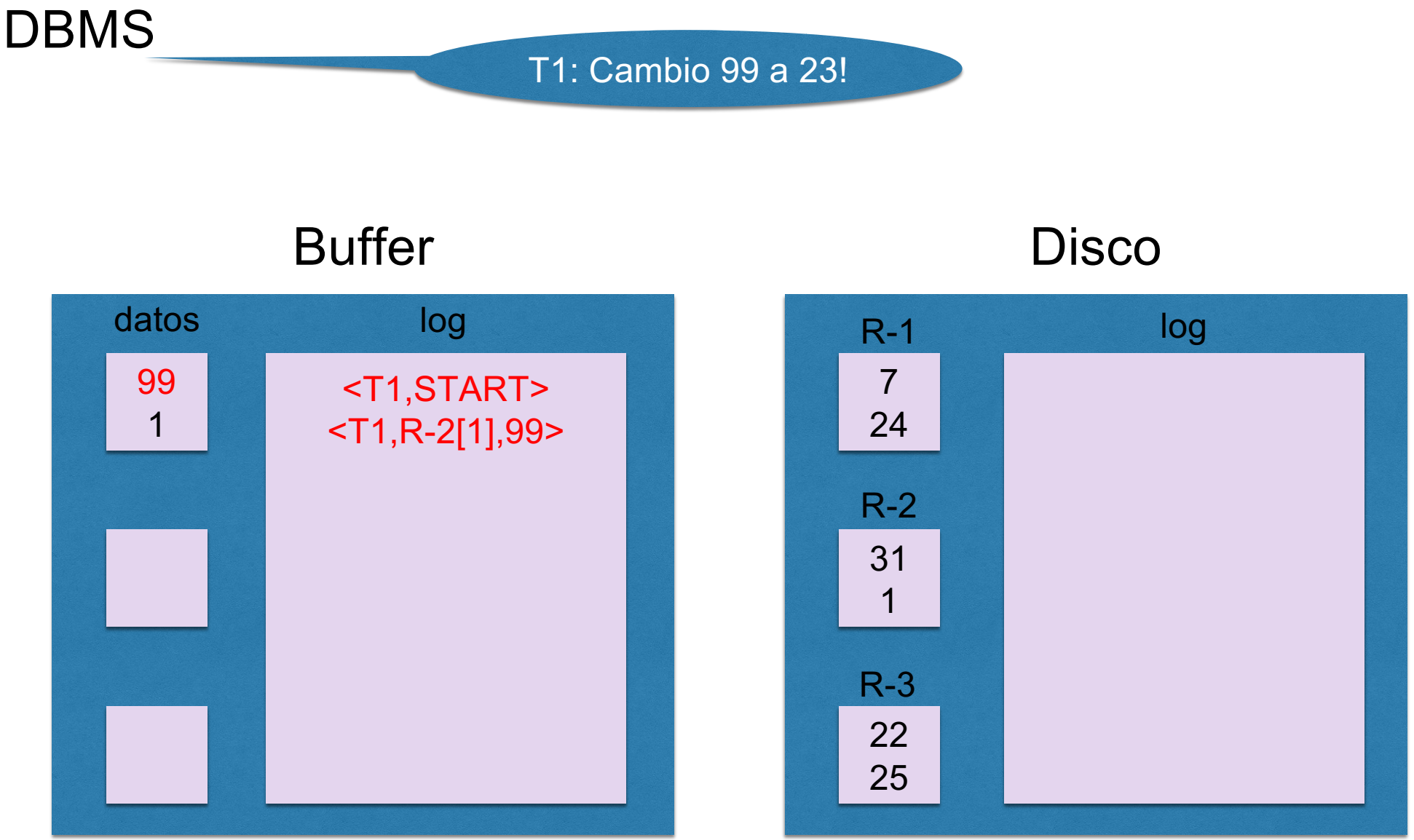
Redo Logging – en la BD



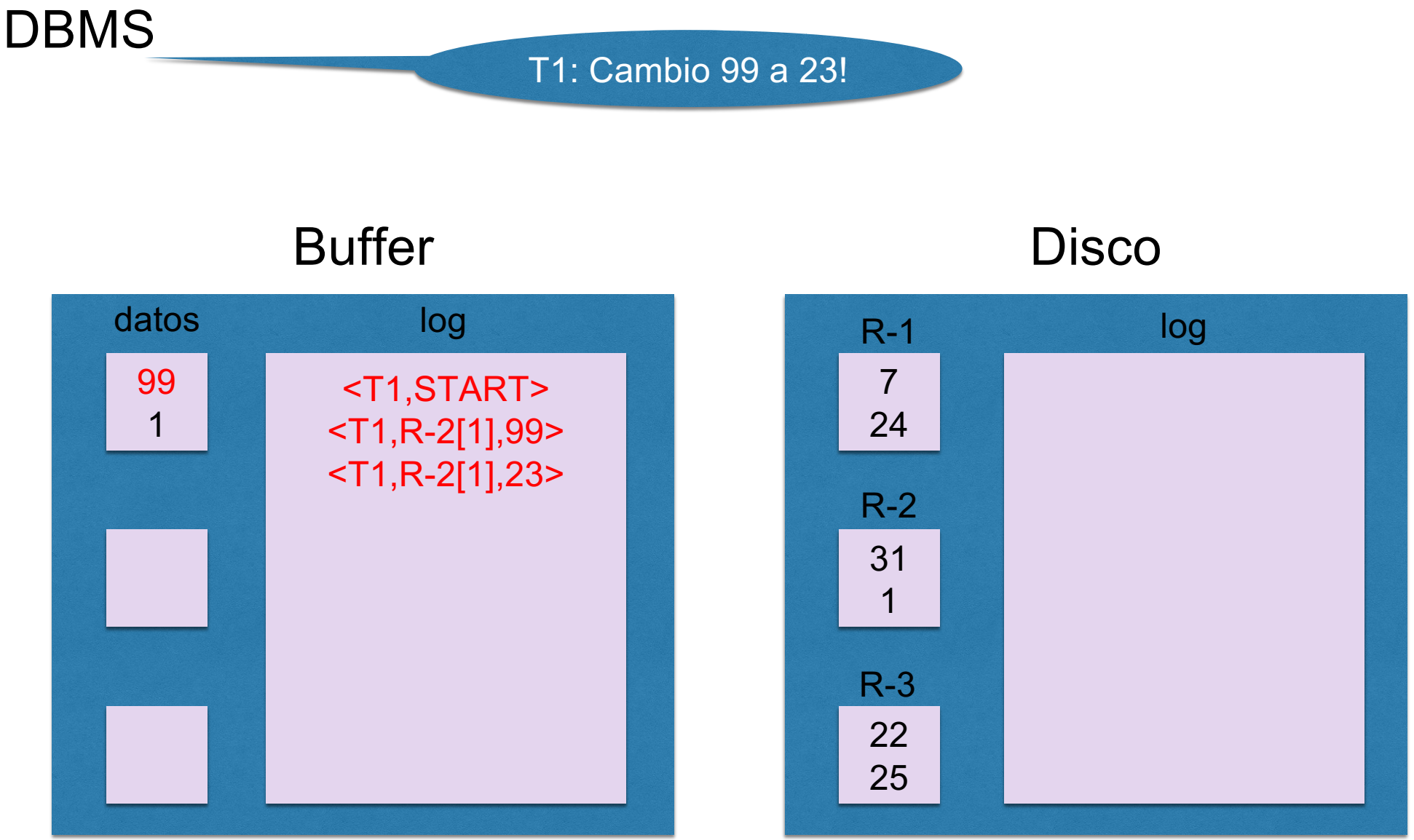
Redo Logging – en la BD



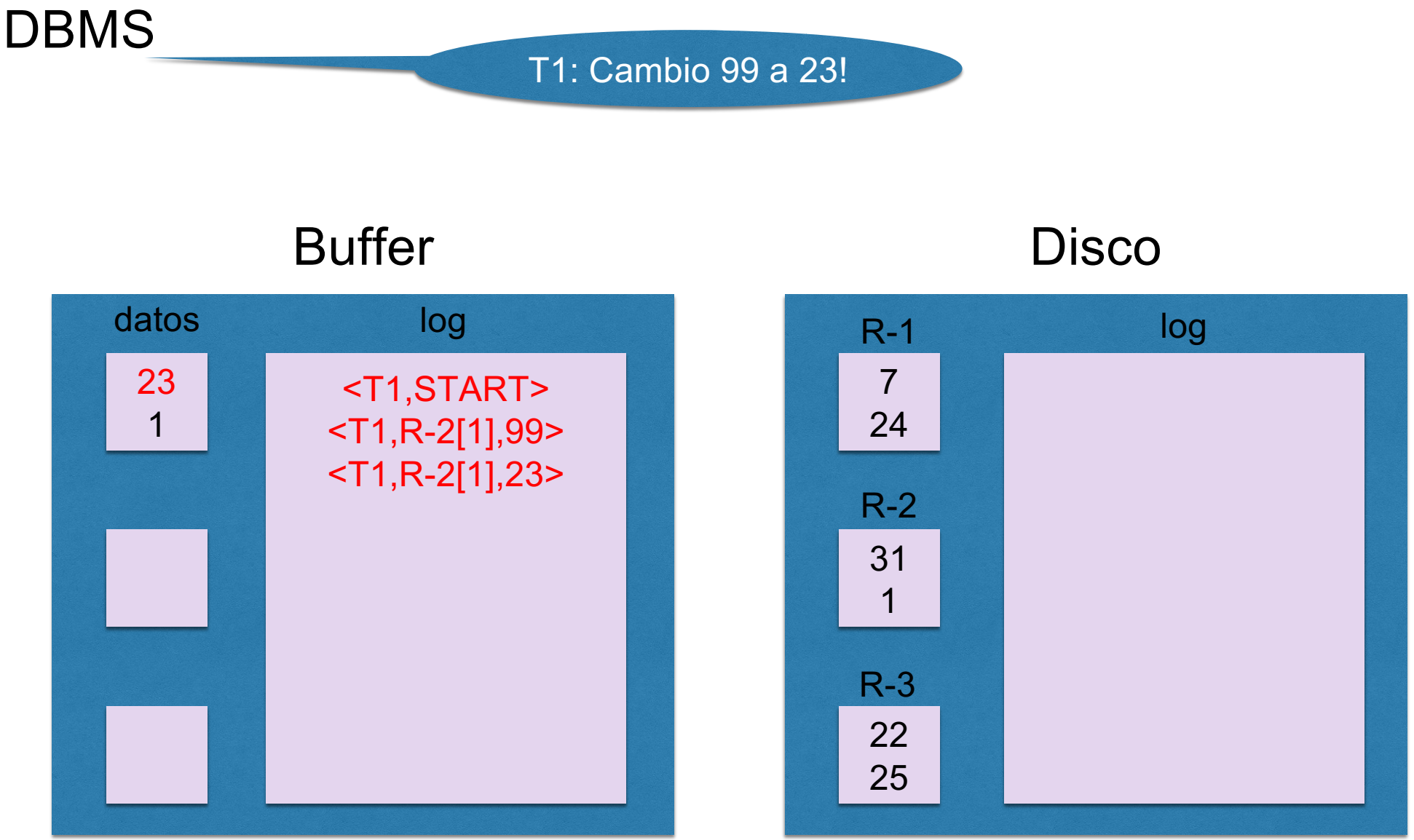
Redo Logging – en la BD



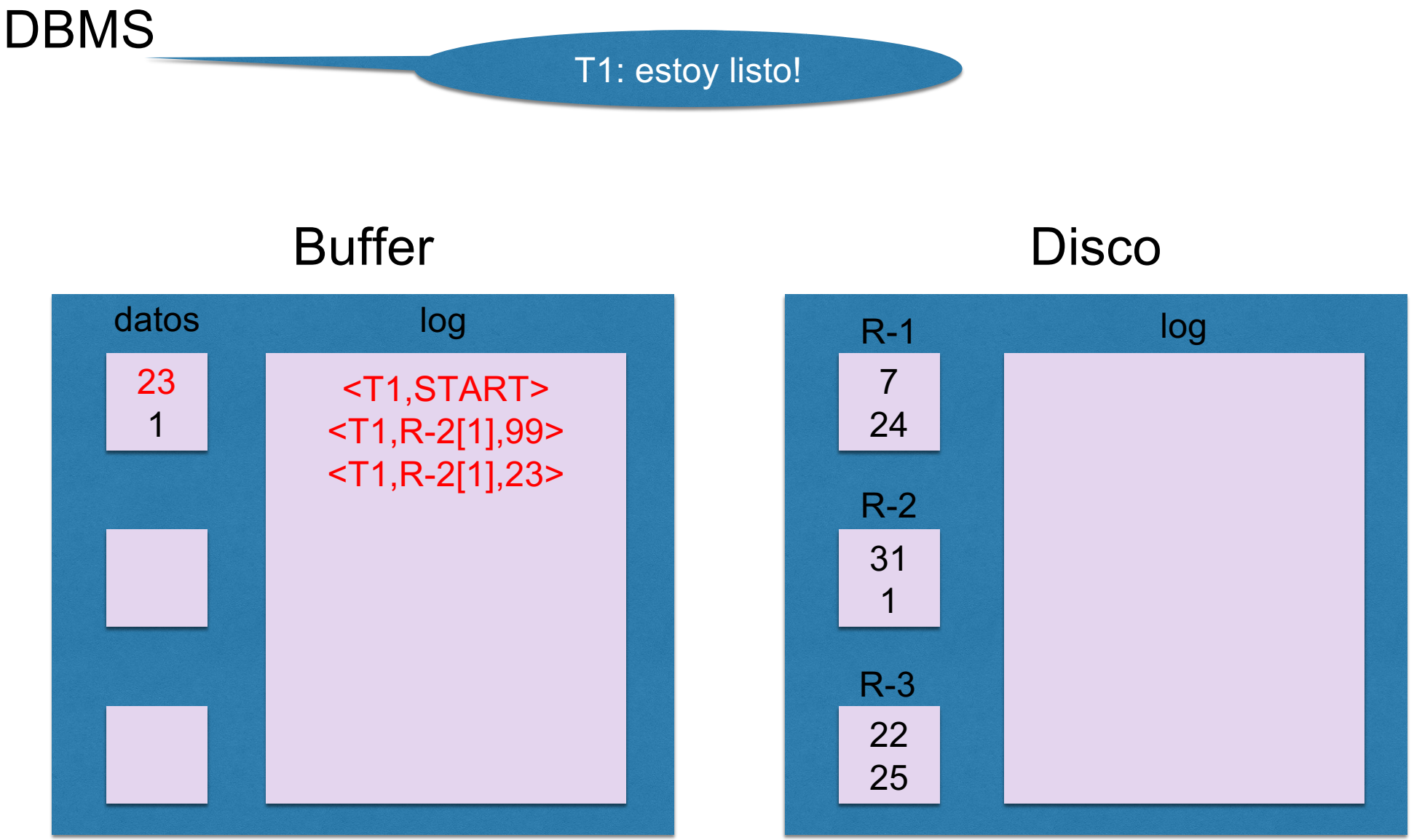
Redo Logging – en la BD



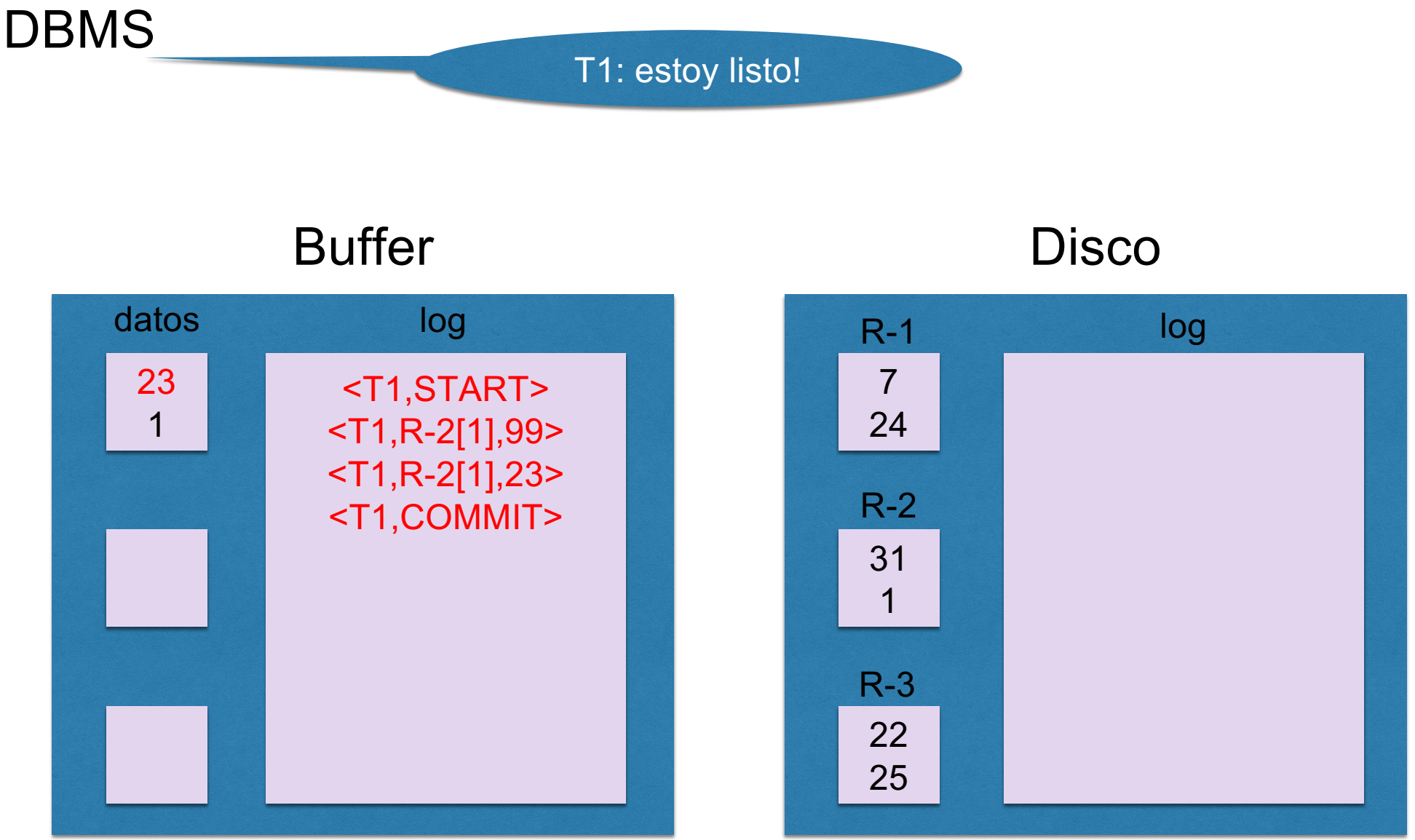
Redo Logging – en la BD



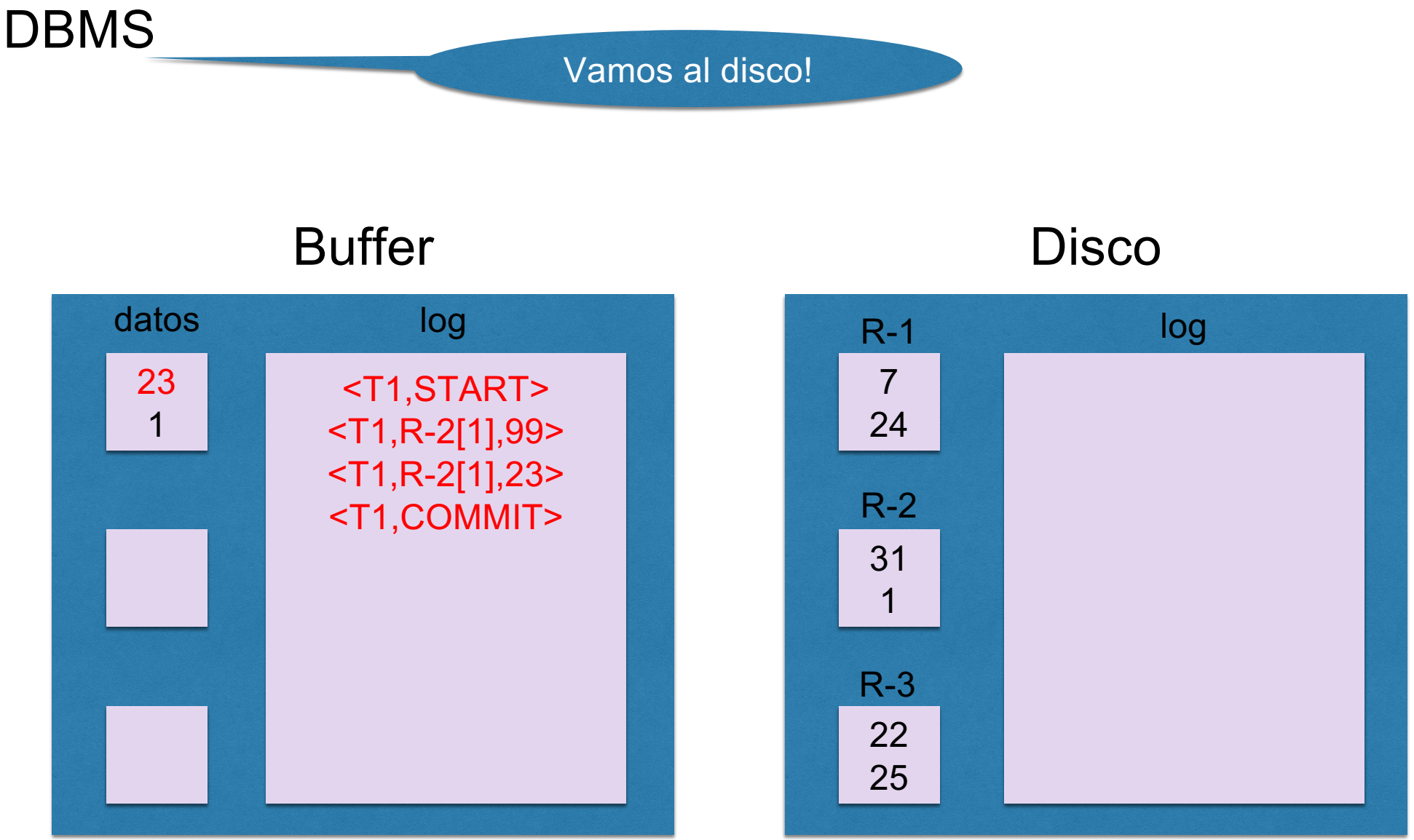
Redo Logging – en la BD



Redo Logging – en la BD



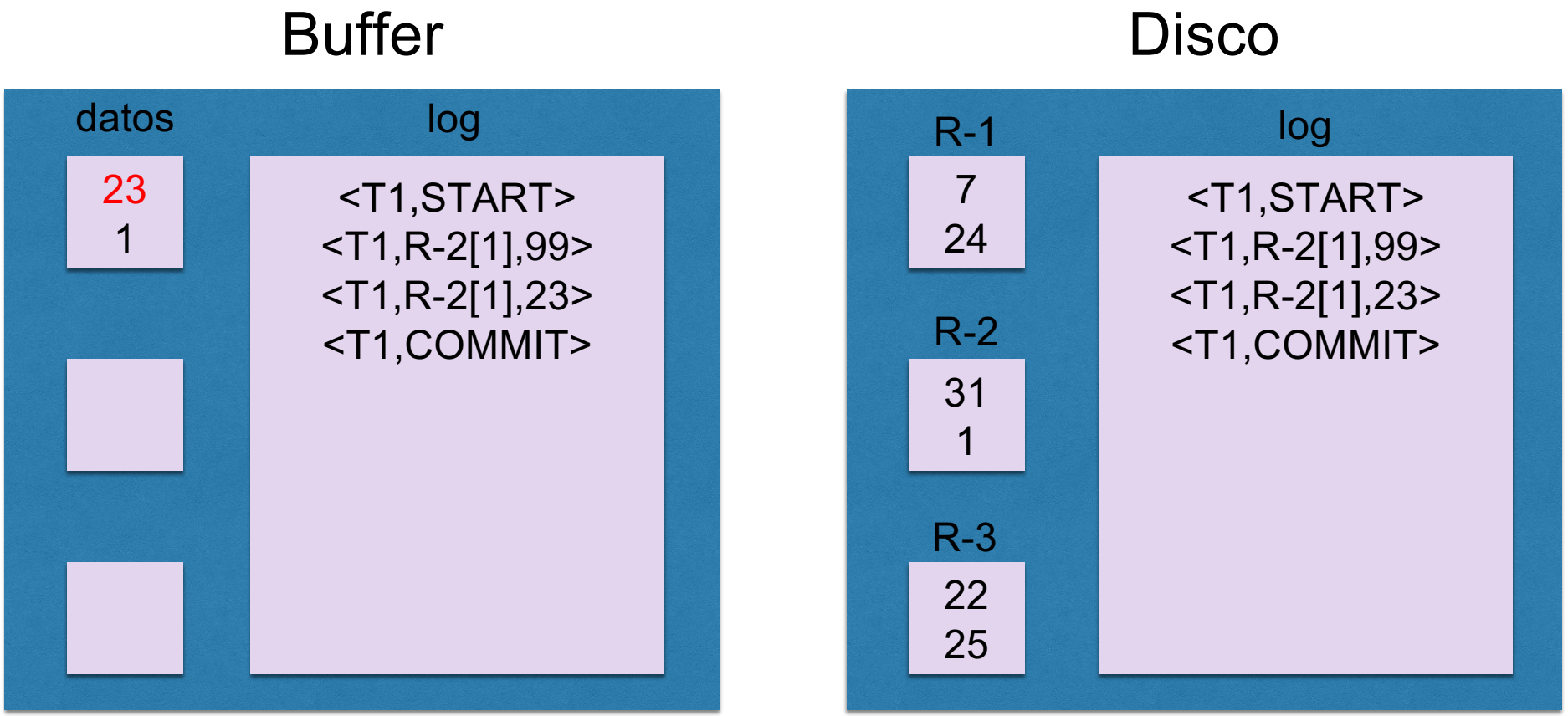
Redo Logging – en la BD



Redo Logging – en la BD

DBMS

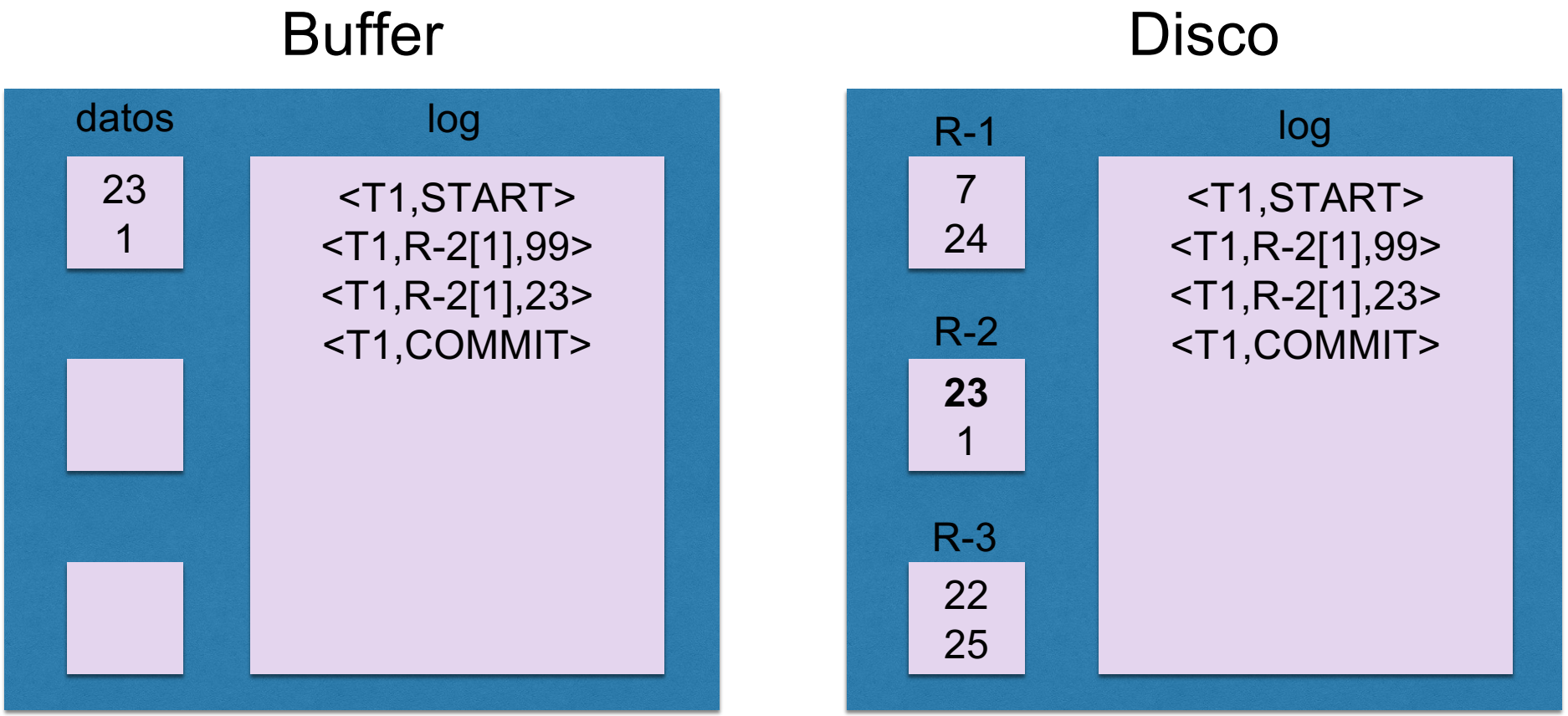
Vamos al disco!



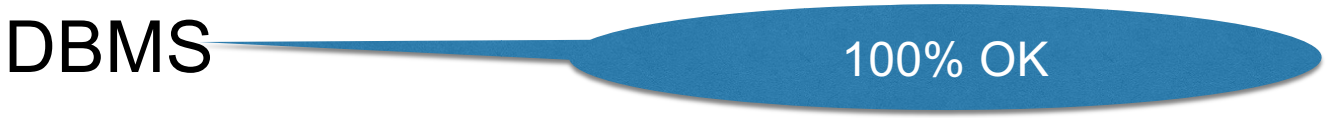
Redo Logging – en la BD

DBMS

Actualizo los datos



Redo Logging – en la BD



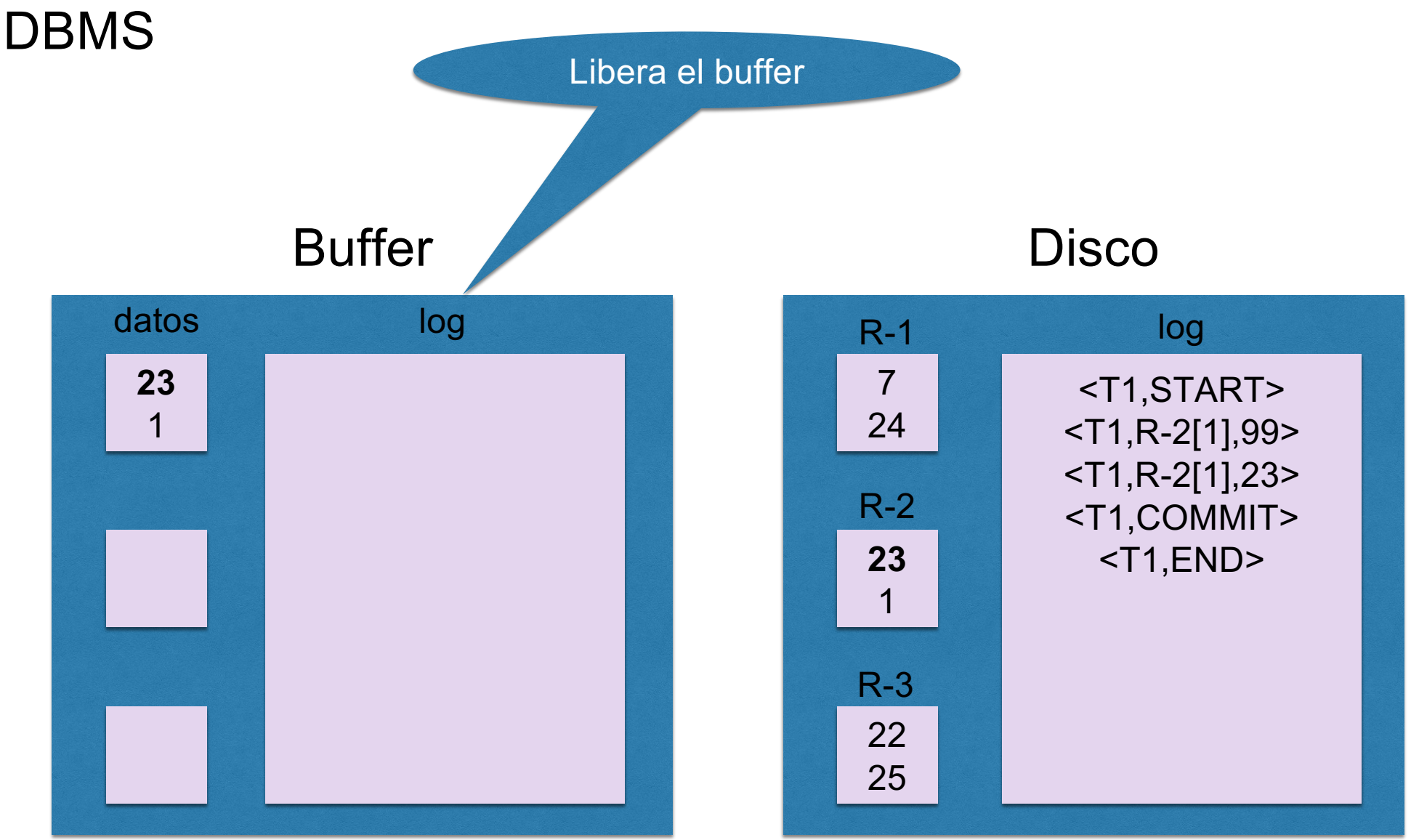
Buffer

datos	log
23 1	<T1,START> <T1,R-2[1],99> <T1,R-2[1],23> <T1,COMMIT> <T1,END>

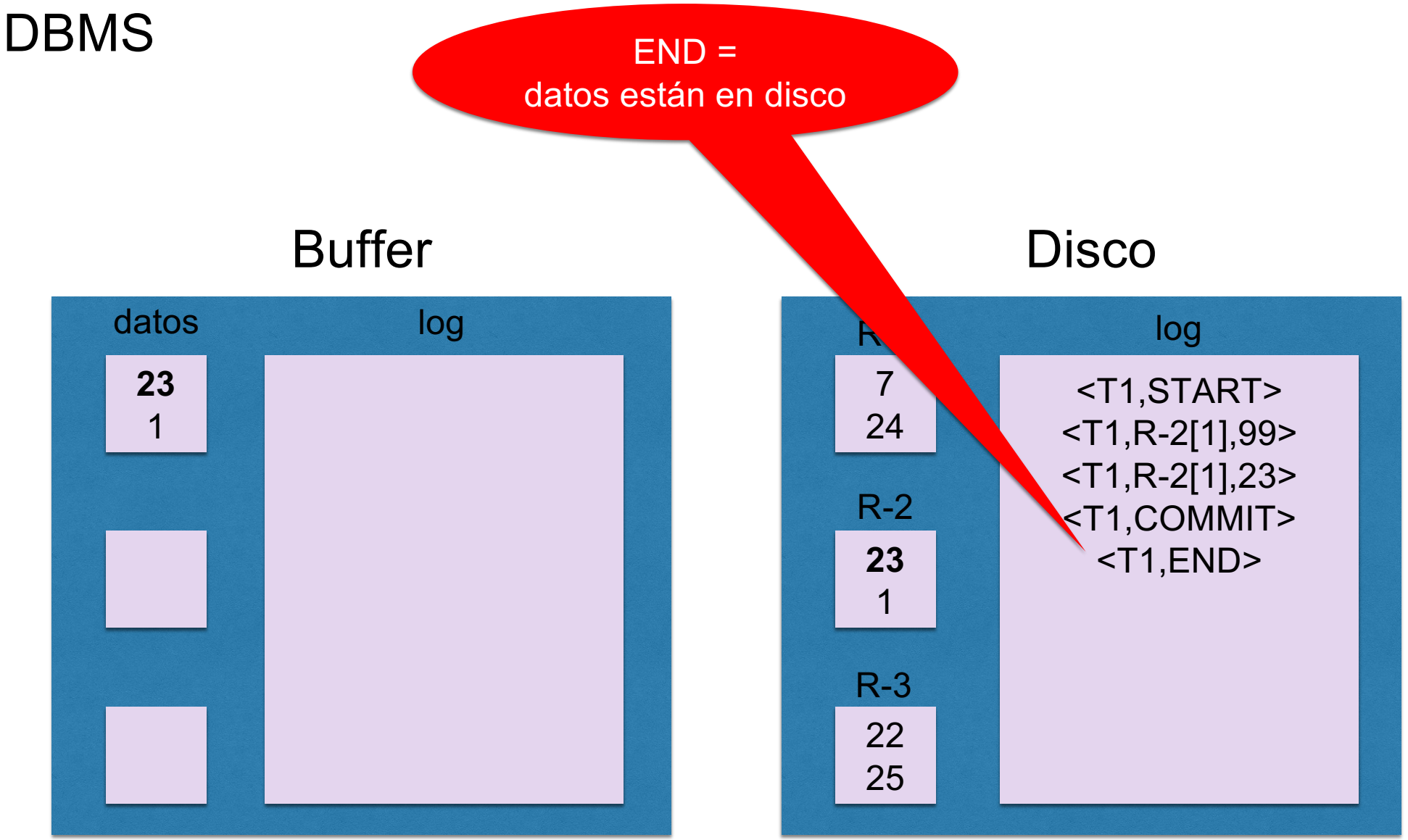
Disco

R-1	log
7 24	<T1,START> <T1,R-2[1],99> <T1,R-2[1],23> <T1,COMMIT> <T1,END>
R-2	
23 1	
R-3	
22 25	

Redo Logging – en la BD



Redo Logging – en la BD



Recuperación con Redo Logging

Detectando fallas en el *log*:

... <START T> ... <END T> ...



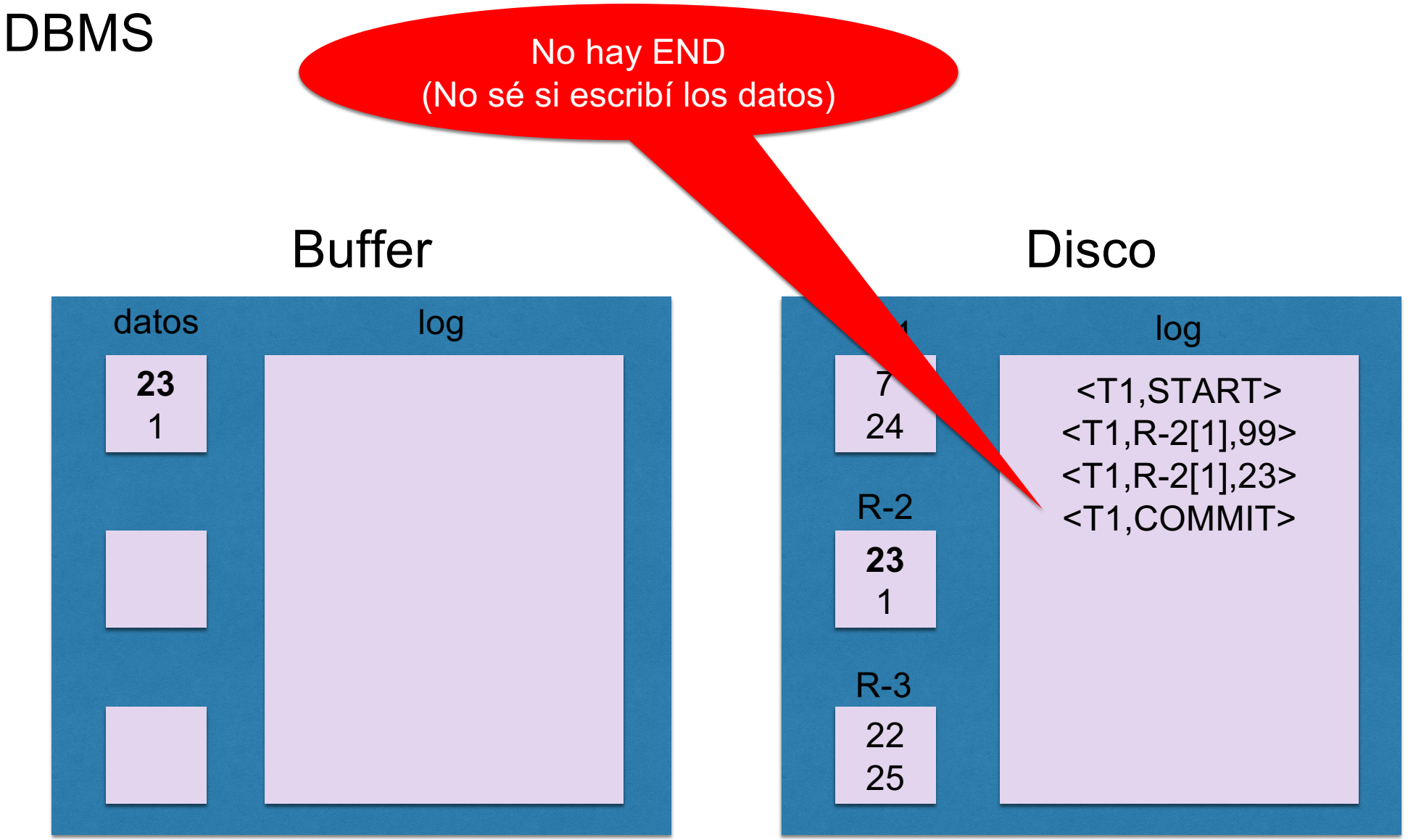
Recuperación con Redo Logging

Detectando fallas en el *log*:

... <START T> ... <COMMIT T> ...



Redo Logging – en la BD



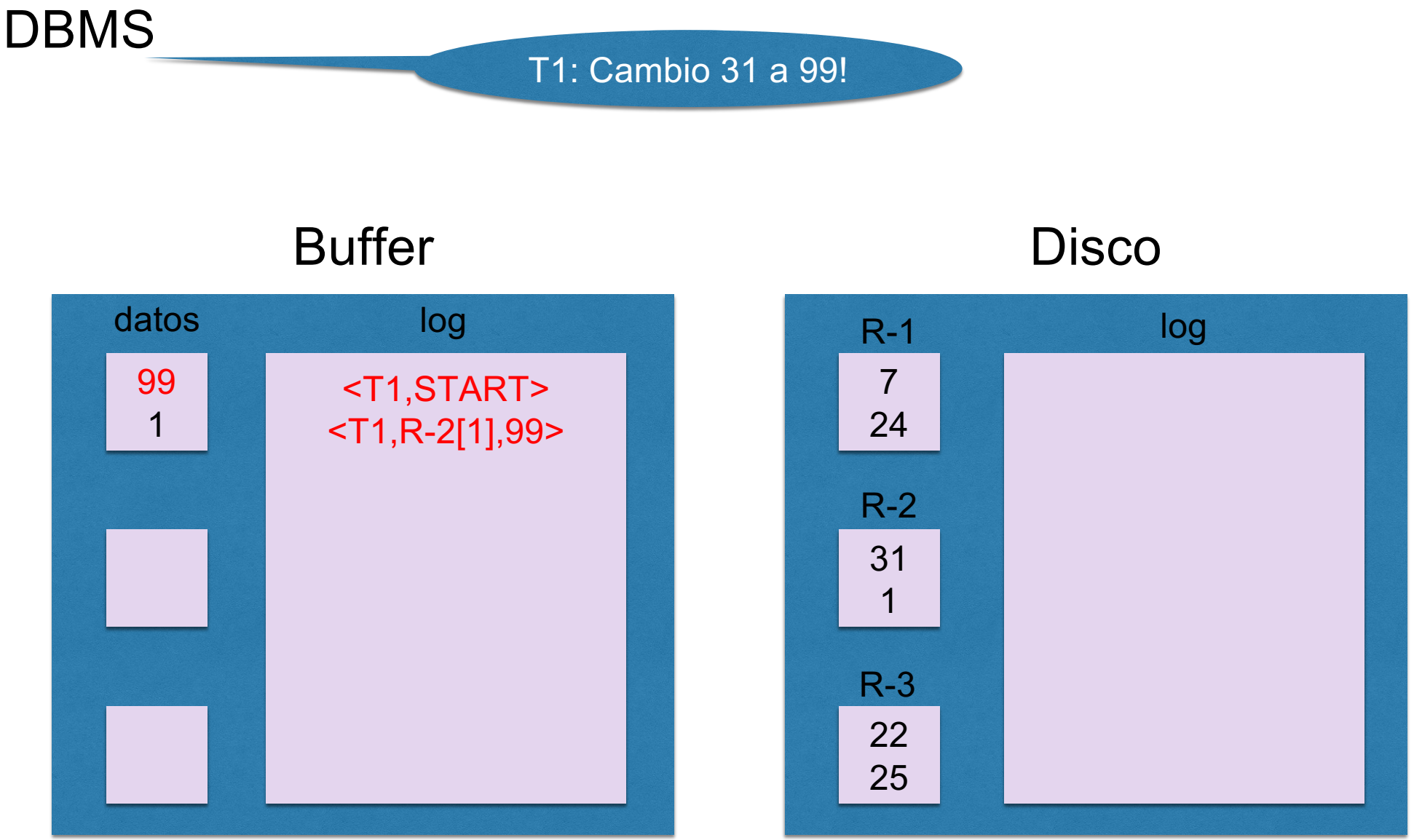
Recuperación con Redo Logging

Detectando fallas en el *log*:

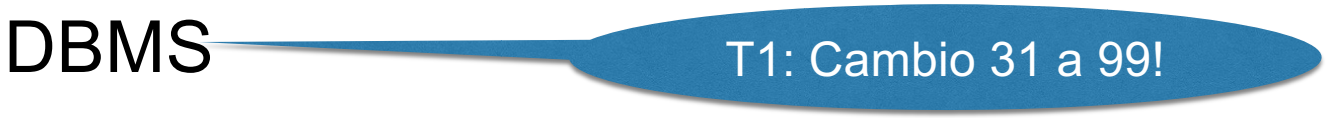
... <START T> ... <ABORT T> ...

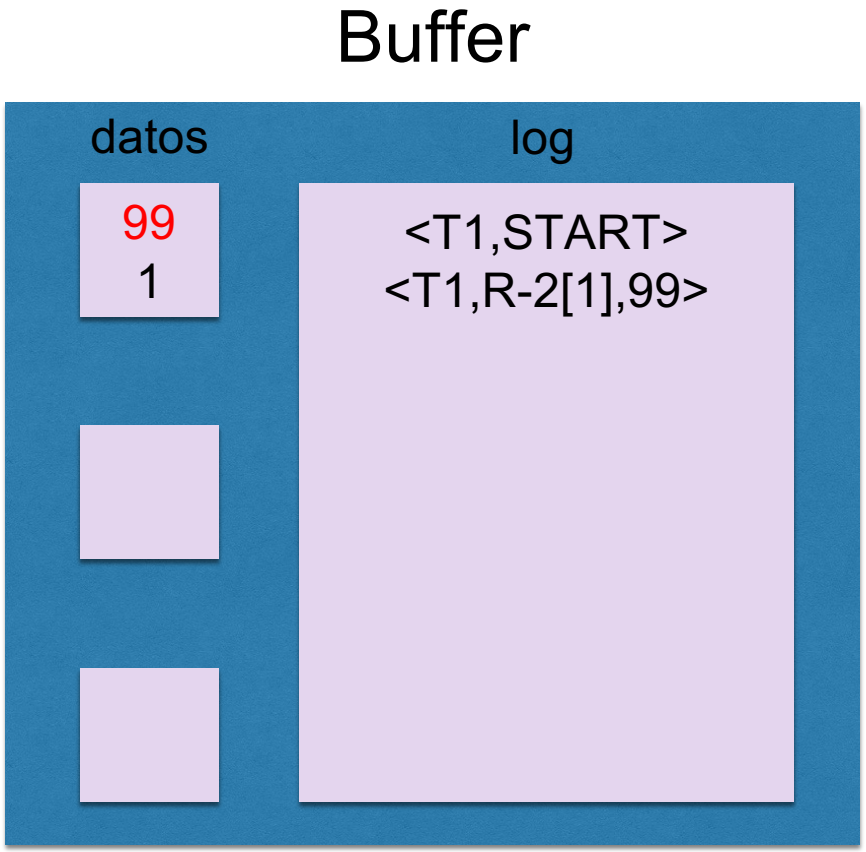


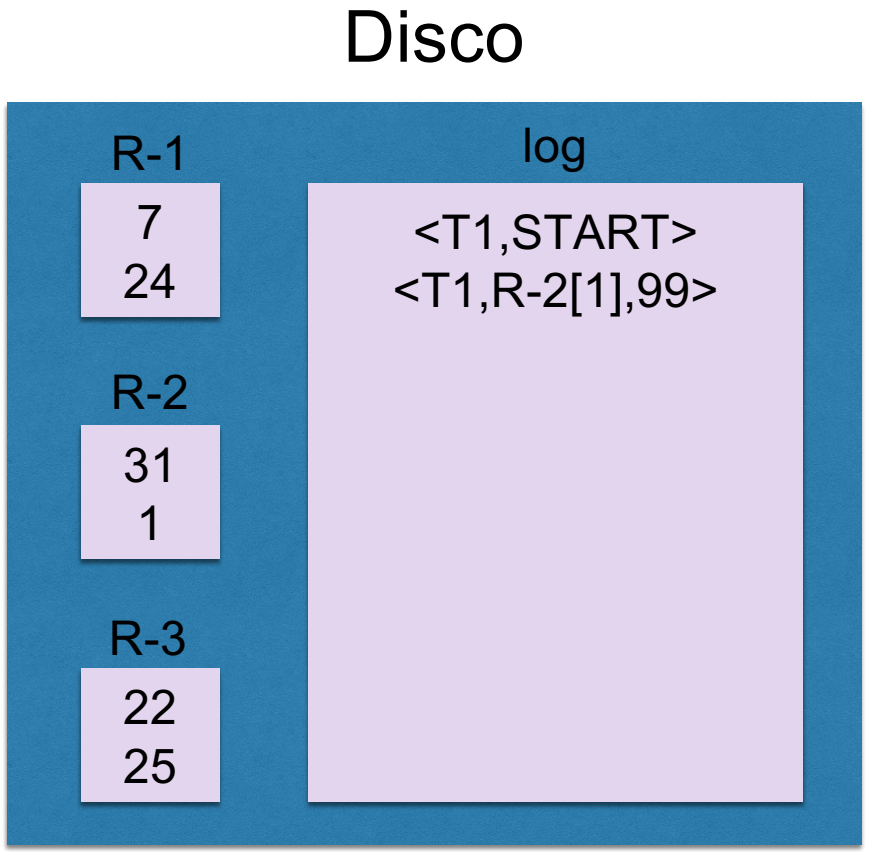
Redo Logging – en la BD



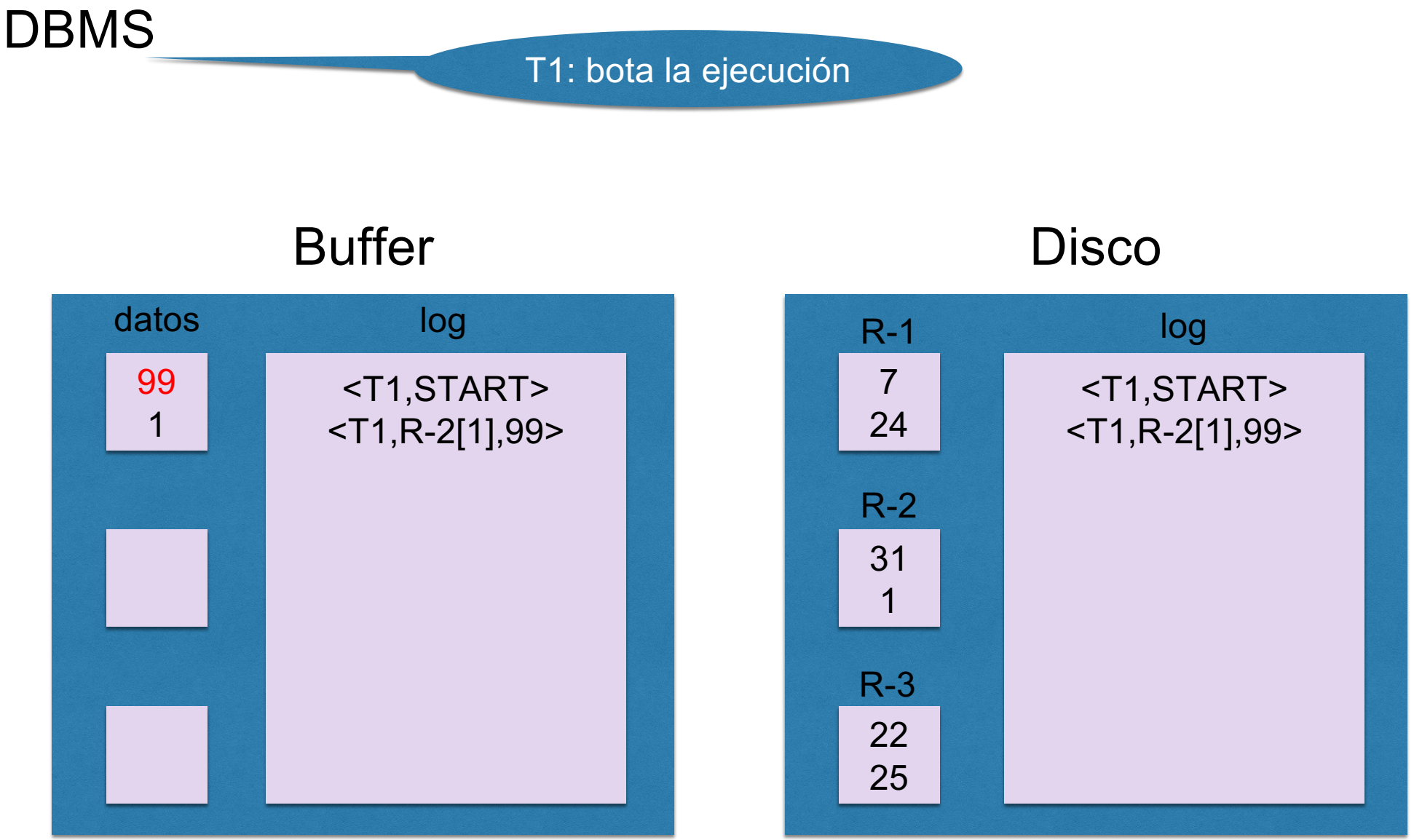
Redo Logging – en la BD







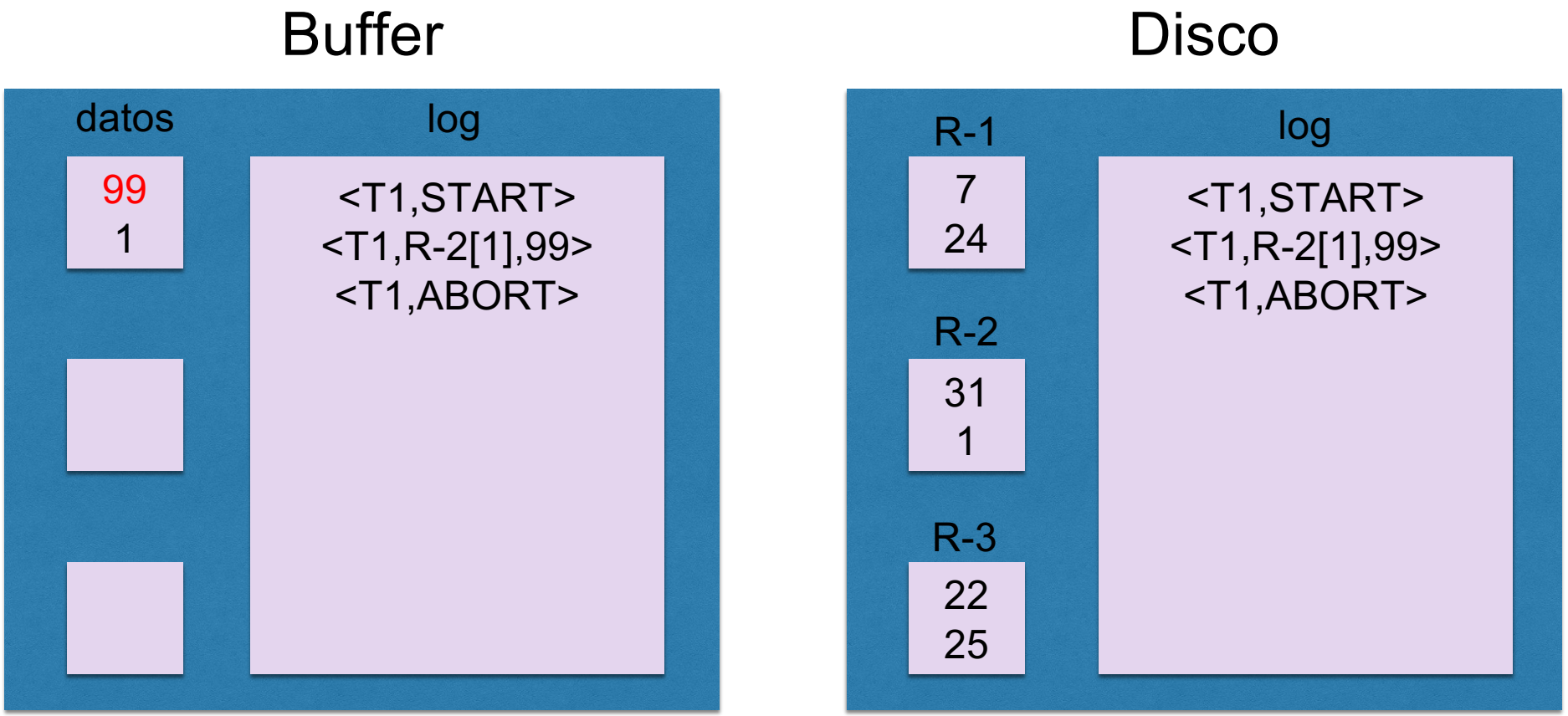
Redo Logging – en la BD



Redo Logging – en la BD

DBMS

T1: bota la ejecución



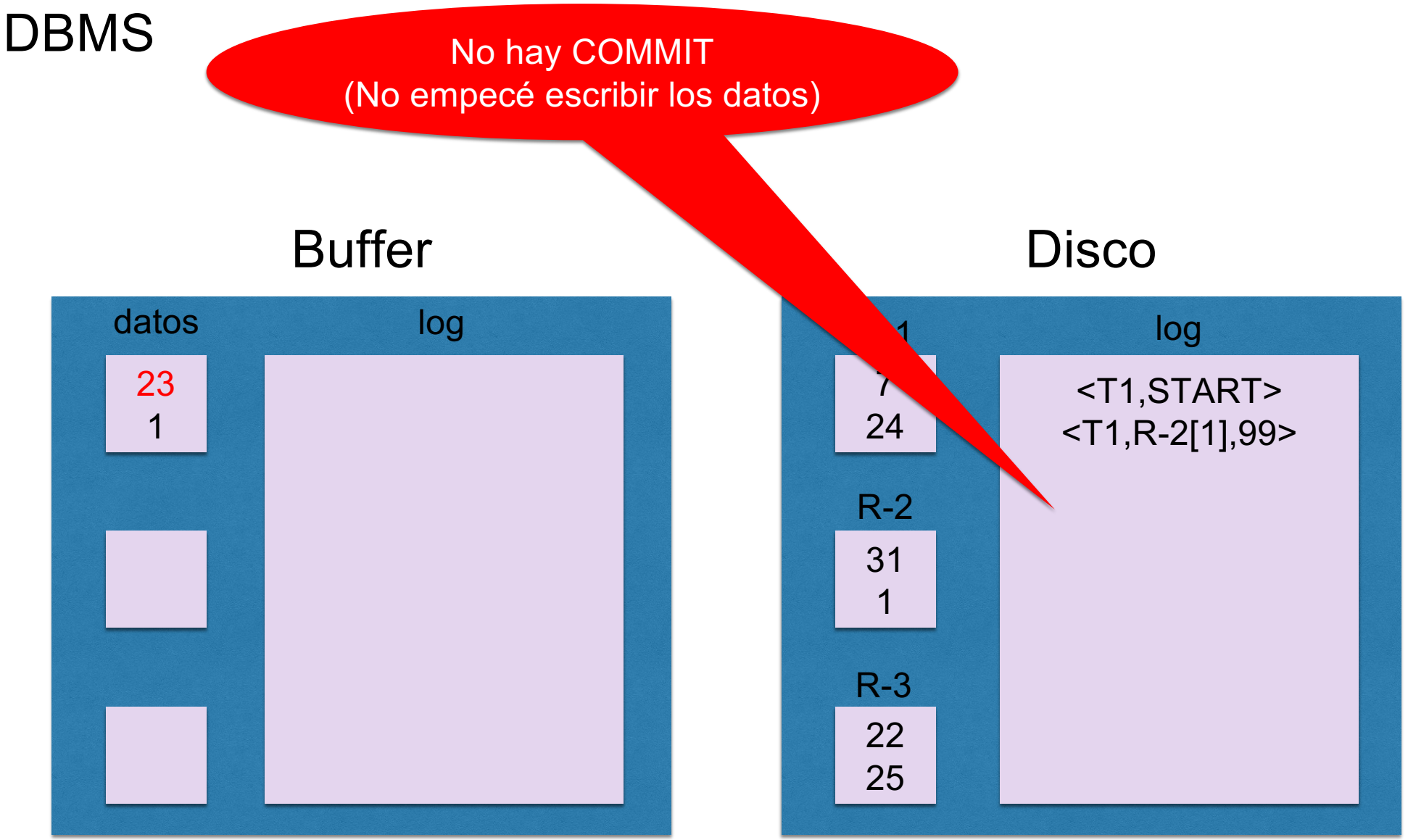
Recuperación con Redo Logging

Detectando fallas en el *log*:

... <START T> ...



Redo Logging – en la BD



Recovery

Algoritmo para un *Redo Logging*

Procesamos el *log* desde el principo hasta el final:

- Identificamos las transacciones que hicieron COMMIT *sin hacer END* (si hicieron END todo OK)
- Hacemos un *scan* desde el principio
- Si leo <**T**, X, v>:
  - Si **T** no hizo COMMIT, no hacer nada
  - Si **T** hizo COMMIT, reescribir con el valor v
- Para cada transacción sin COMMIT, escribir <ABORT **T**>

Recovery

Uso de *Checkpoints* en *Redo Logging*

¿Cómo utilizamos los checkpoints en el Redo Logging?

Recovery

Uso de *Checkpoints* en *Redo Logging*

- Escribimos un *log* <START CKPT (T_1, ..., T_n)>, donde T_1, ..., T_n son transacciones activas y sin COMMIT
- Guardar todo el log en el disco
- Guardar en disco todo lo que haya hecho COMMIT hasta ese punto; escribir en log END al finalizar
- Una vez hecho, escribir <END CKPT>

Ejemplo

Cuand finalizamos un checkpoint

Log
<START T1>
<T1, a, 5>
<START T2>
<COMMIT T1>
<T2, b, 10>
<START CKPT (T2)>
<T2, c, 15>
<START T3>
<T3, e, 25>
<END T1>
<END CKPT>

T1 no hizo END

Cambios de T1 en disco

Todo antes de T2 listo

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

- Revisar el *log* desde el final al inicio
- Si encontramos un <END CKPT>, debemos retroceder hasta su su respectivo <START CKPT (T_1, ..., T_n)>, y comenzar a hacer *redo* desde la transacción más antigua entre T_1, ..., T_n – las *sin* END
- No se hace *redo* de las transacciones con COMMIT antes del <START CKPT (T_1, ..., T_n)>

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

- Revisar el *log* desde el fina
- Si encontramos un <E , debemos retroceder hasta su su respe <START CKPT (T_1, ..., T_n)>, y comenzar a cer *redo* desde la transacción más antigua entr T_1, ..., T_n
- No se hace *redo* de las transacciones con COMMIT antes del <START CKPT (T_1, ..., T_n)>

Su END aparece antes de <END CKPT>

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

- Revisar el *log* desde el fina
- Si encontramos un <E , debemos retroceder hasta su su respe <START CKPT (T_1, ..., T_n)>, y comenzar a cer *redo* desde la transacción más antigua entr T_1, ..., T_n
- No se hace *redo* de las transacciones con COMMIT antes del <START CKPT (T_1, ..., T_n)>

En realidad END de estas transacciones es redundante

Redo Recovery

Uso de *Checkpoints* en *Redo Logging*

Si encontramos un <START CKPT (T_1, ..., T_n)> sin su <END CKPT>, debemos retroceder hasta encontrar un <END CKPT> más antiguo

Ejemplo

Uso de *Checkpoints* en *Redo Logging*

Considere este *log* después de una falla:

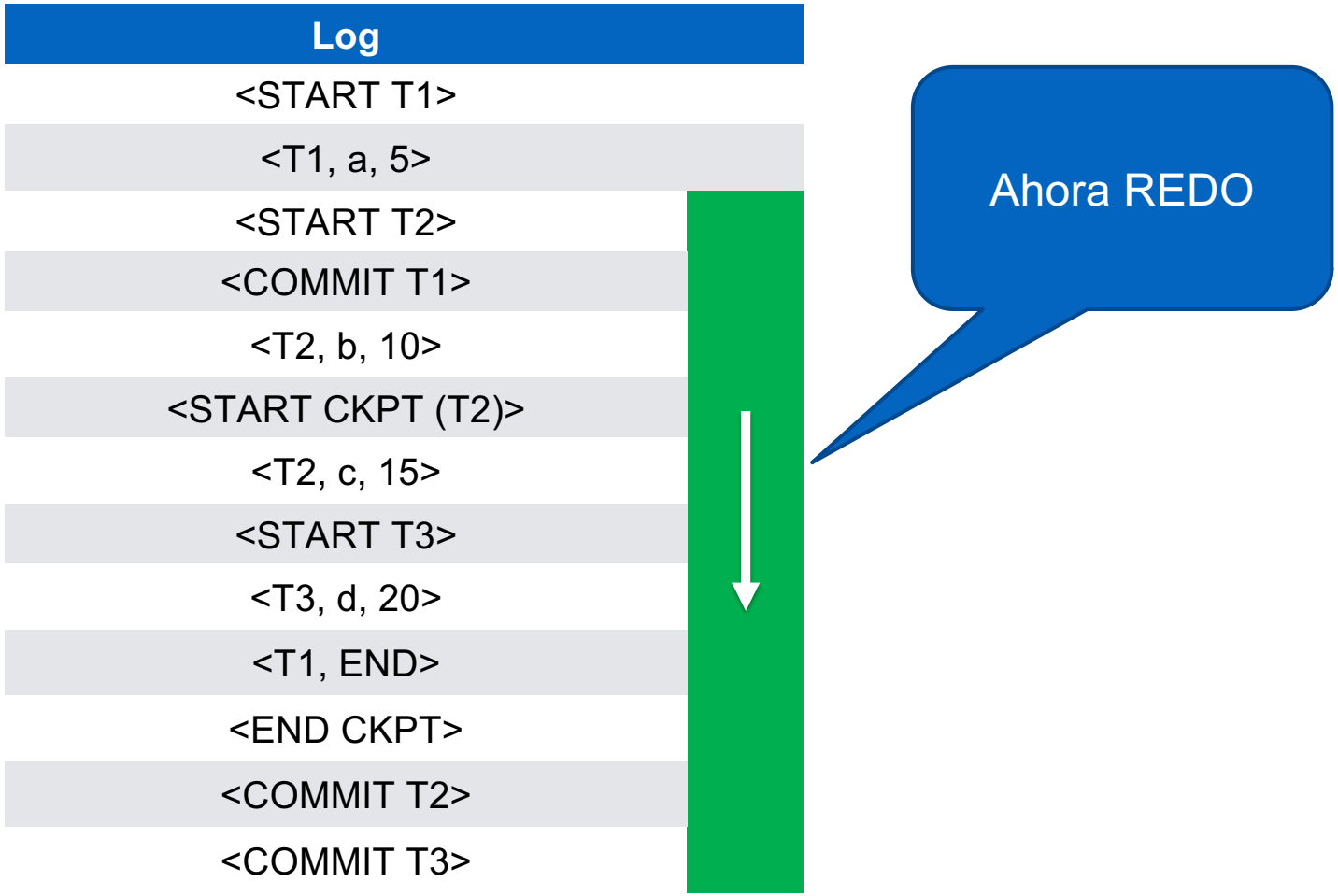
Log
<START T1>
<T1, a, 5>
<START T2>
<COMMIT T1>
<T2, b, 10>
<START CKPT (T2)>
<T2, c, 15>
<START T3>
<T3, d, 20>
<T1, END>
<END CKPT>
<COMMIT T2>
<COMMIT T3>



Ejemplo

Uso de *Checkpoints* en *Redo Logging*

Considere este *log* después de una falla:



Redo Logging

Pero hay un problema: no es posible ir grabando los valores de X en disco antes que termine la transacción

¡Por lo tanto se congestiona la escritura en disco!

Undo/Redo Logging

Es la solución para obtener mayor performance que mezcla las estrategias anteriormente planteadas

Undo/Redo Logging

Los *logs* son:

- <START **T**>
- <COMMIT **T**>
- <ABORT **T**>
- <**T**, X, $v_{antiguo}$, v_{nuevo}>
- <**T**,END> (todo en disco – COMIT y valor)

Undo/Redo Logging

El flujo:

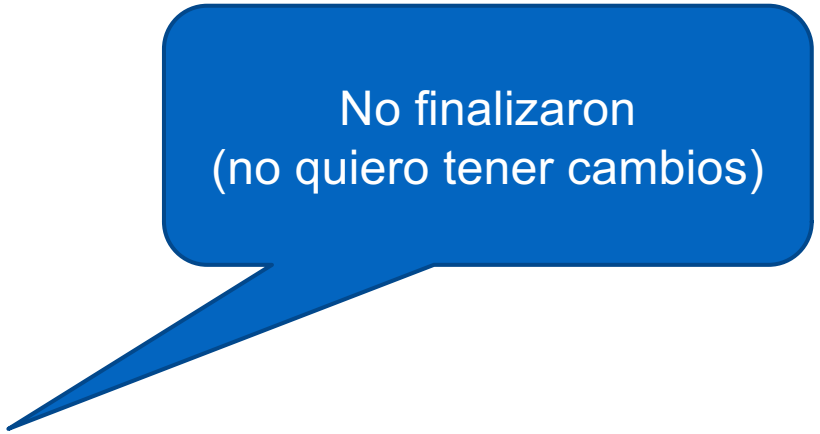
- Log <**T**, X, $v_{antiguo}$, v_{nuevo}> va al disco antes de X
- (write-ahead logging)
- <**T**,COMMIT> va de manera arbitraria al disco
- (antes o después de X)
- <**T**,END> va cuando escribimos y datos y COMMIT

Undo/Redo Logging

Recuperación:

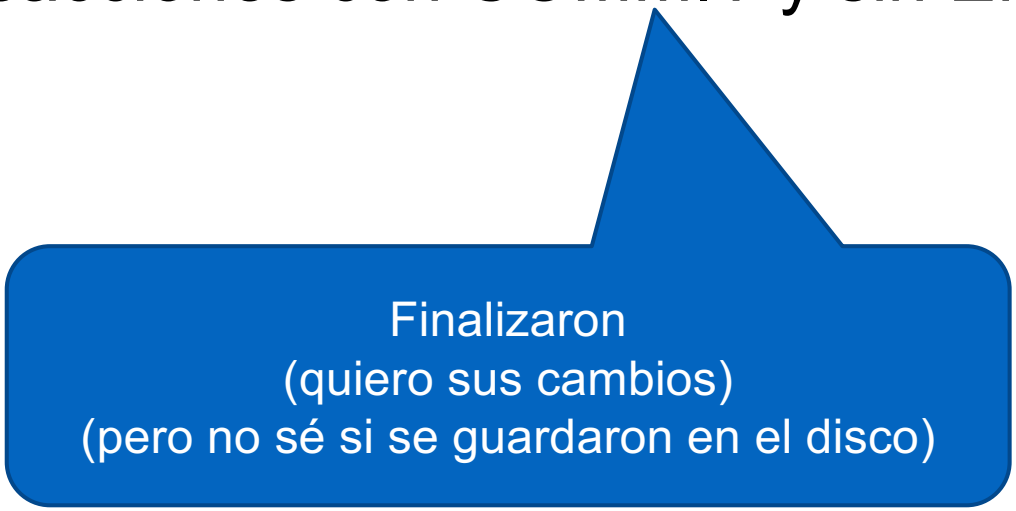
- Undo de transacciones sin COMMIT
- Redo transacciones con COMMIT y sin END

Undo/Redo Logging



Recuperación:

- Undo de transacciones sin COMMIT
- Redo transacciones con COMMIT y sin END



Técnicas de Logging

Resumen

	Undo	Redo
Trans. Incompletas	Cancelarlas	Ignorarlas
Trans. Comiteadas	Ignorarlas	Repetirlas
Escribir COMMIT	Después de almacenar en disco	Antes de almacenar en disco
UPDATE *Log Record*	Valores antiguos	Valores nuevos

Comentario importante

Para que todo esto funcione:

- Asumimos ACID
- En particular, que no hay conflictos RW, WR, o WW

Recursos para estudiar

http://mlwiki.org/index.php/Database_Transaction_Log

Database Management Systems, 3rd edition, de Raghu Ramakrishnan y Johannes Gehrke. Cap 18