DBMS Tutorial 21.11.2018

Topics

Number of Bits Required to represent a Number

Addressing, TID, Pointer Swizzling

How many elements can be represented by n bits

1 bit possible values {0 or 1} therefore 2 elements or **2^1**

2 bits possible values {00, 01, 10, 11} therefore **2^2**

3 bits possible values {000, 001, 010, 011, 100, 101, 110, 111} therefore **2^3**

In general:

n bits can represent **2^n** possible values.

How many bits for M elements or number

On the other hand, we want to find out how many bits we need to represent a number:

- M = 2, bits required to represent two values can be simply {0,1} hence 1 bit will suffice!
- M = 5, we can represent five values as {000, 001, 010, 011, 100} therefore 3 bits at least.
- M = 9, we can represent nice values as {0,1,10,11,100,101,110,111,1000}, therefore 4 bits!
- M = 1321, in binary 1320 is 10100101000 ie at least 11 bits.

How many bits for M elements or number

In general: number of bits, n, needed to represent M elements is given by

$$\mathbf{n = \lceil \log_2(M) \rceil = ceil\ (\log_2(M))}$$

In other words, find an **n** such that…

$$2^n \geq M > 2^{n-1}$$

How many bits for M elements or number

- M = 2, bits required to represent two values can be simply {0,1} hence 1 bit will suffice! $\lceil \log_2(2) \rceil = 1$
- M = 5, we can represent five values as {000, 001, 010, 011, 100} therefore 3 bits at least. $\lceil \log_2(5) \rceil = 3$
- M = 9, we can represent nice values as {0,1,10,11,100,101,110,111,1000}, therefore 4 bits! $\lceil \log_2(9) \rceil = 4$
- M = 1321, in binary 1320 is 10100101000 ie at least 11 bits. $\lceil \log_2(1321) \rceil = 11$

Exercise Sheet 4, Question 4

4) Consider Disk 1 discussed in the lecture. We want to represent physical addresses for this disk by allocating one or more bytes to identify each of the cylinder, track within cylinder, and block within a track.

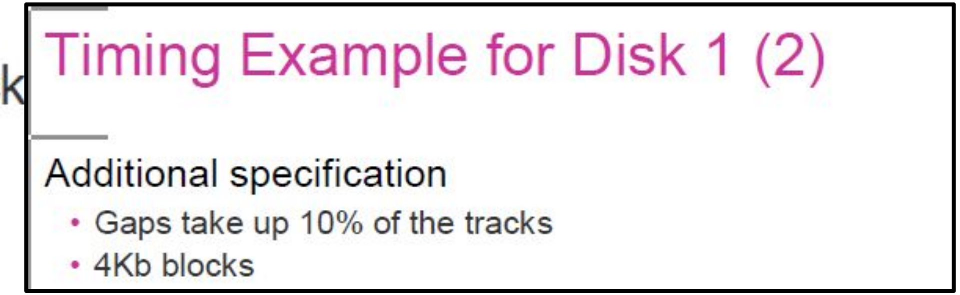
a) How many bytes do we need for a block address? Make a reasonable assumption about the maximum number of blocks on each track. Recall that Disk 1 has a variable number of sectors per track.

b) We want construct a record address by adding the number of the byte within a block to the block address of exercise a). How many bytes would we need for the record address?

Example Disk 1

# Surfaces = 8
# Cylinders = 8192
# Blocks/track = 32
(#Blocks/Track=Track capacity/Block Size = 512 * 256 / (4 * 1024))

- Four platters, eight surfaces.
- Diameter = 3.5 inches; only outer 1 inch is occupied by tracks.
- 2^{13} = 8192 cylinders.
- On average 256 sectors for each track (fewer near center, more near edge, so average bit density doesn't vary too much).
- 512 bytes/sector.
- Capacity = 8 surfaces * 8192 track
  512 bytes/sector = 8 Gbytes.



Timing Example for Disk 1 (2)

Additional specification

- Gaps take up 10% of the tracks
- 4Kb blocks

Exercise Sheet 4, Q4 Solution

4) Consider Disk 1 discussed in the lecture. We want to represent physical addresses for this disk by allocating one or more bytes to identify each of the cylinder, track within cylinder, and block within a track.

We need 1 or more bytes to represent each, ie, we can work with bits in multiples of 8, ie 1 byte.

a) How many bytes do we need for a block address? Make a reasonable assumption about the maximum number of blocks on each track. Recall that Disk 1 has a variable number of sectors per track.

To address a block we need to know the following : The Surface its one, the cylinder (i.e. track its on), and then the block itself. (Think of it as a real address: The Surface is the colony, the track is the street, and the block is the building!)

Bits to address the surface (out of 8 options) = $\lceil \log_2(8) \rceil$ = 3bits or 1byte

Bits to address the cylinder (out of 8192 options) = $\lceil \log_2(8192) \rceil$ = 13 bits or 2 bytes

Bits to address the block (out of 32 options) = $\lceil \log_2(32) \rceil$ = 5 bits or 1 byte.

Exercise Sheet 4, Q4 Solution

We are rounding off to bytes as thats is the granularity we are dealing with ("one or more bytes to identify each of the cylinder, track within cylinder, and block within a track")!

Therefore the total bytes required = 2+1+1 = 4 bytes

b) We want construct a record address by adding the number of the bytes within a block to the block address of exercise a). How many bytes would we need for the record address (house number?)?

No. of bytes in a block = 4kb = 4096 bytes.

No. of bits needed to address each of these bytes = $\lceil \log_2(4096) \rceil$ = 12 bits or 2 bytes

Therefore the complete record address would need 4+2 = 6 bytes.

Addressing

How does one refer to records?

Record

Options : Physical and Logical

☆ Purely Physical

E.g.,

$$\begin{array}{l}\text{Record}\\ \text{Address}\\ \text{or ID}\end{array} = \left\{ \begin{array}{l} \left.\begin{array}{l}\text{Device ID}\\ \text{Cylinder \#}\\ \text{Track \#}\\ \text{Block \#}\end{array}\right\} \text{Block ID} \\ \text{Offset in block}\end{array}\right.$$

☆ Purely Physical

$$\text{E.g., Record Address or ID} = \left\{ \begin{array}{l} \left.\begin{array}{l}\text{Device ID}\\ \text{Cylinder \#}\\ \text{Track \#}\\ \text{Block \#}\end{array}\right\} \text{Block ID} \\ \text{Offset in block} \end{array}\right.$$

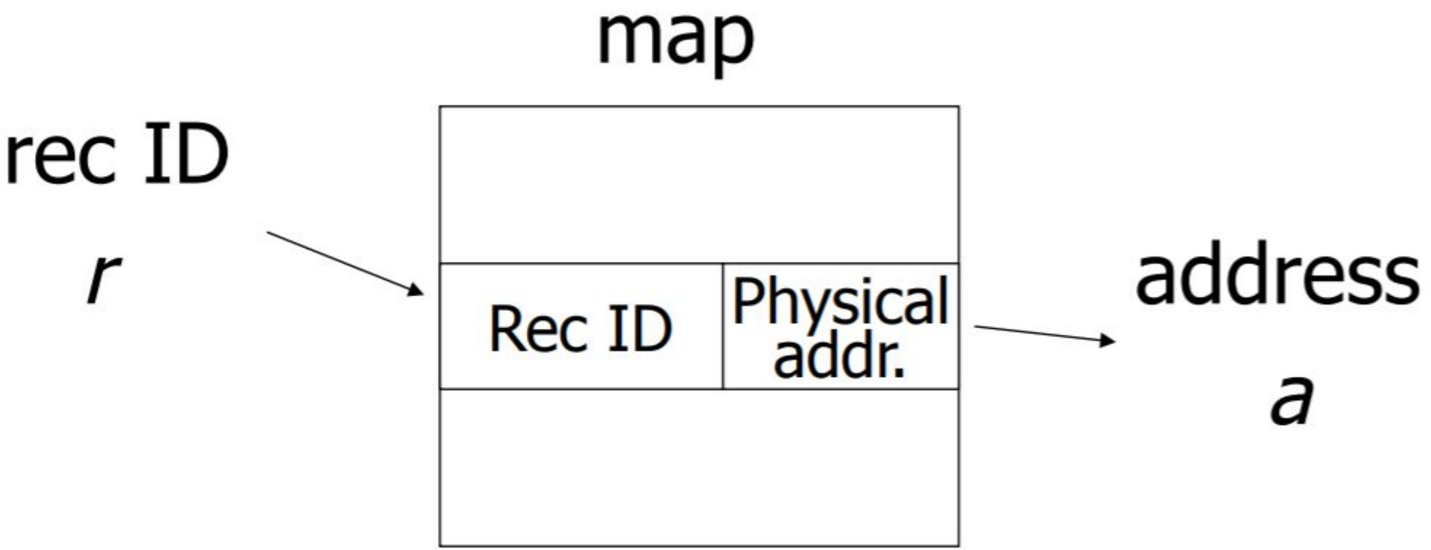
(+) Physical Addresses are faster (quicker to access a record with its physical address) but but address itself is longer. If we have physical address, we can fetch the record in one IO.

(-) If the physical address of a record changes, need to update all pointers pointing to that record. If the record is deleted or moved, might have a dangling pointer.
(-) to avoid dangling pointers for deleted records we need to mark the place with a tombstone, so that anyone following that pointer will know the record doesn't exist ⇒ we can't reclaim all the space freed.

☆ Fully Indirect

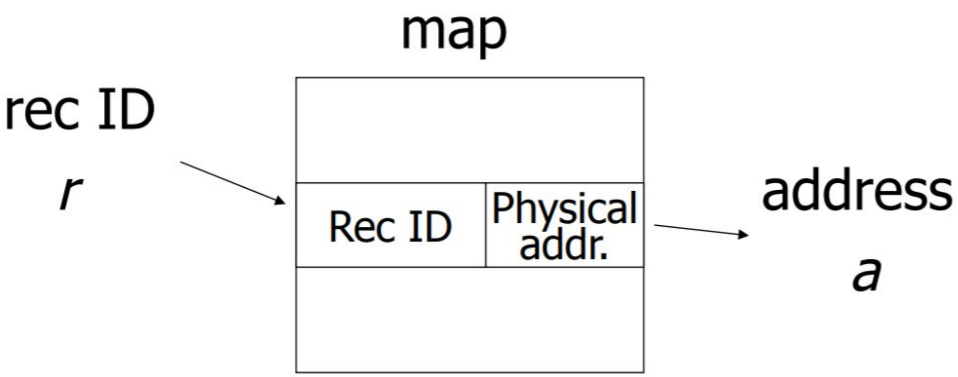
Or Logical

E.g., Record ID is arbitrary bit string



☆ Fully Indirect

E.g., Record ID is arbitrary bit string



Each record has a "logical address," which is an arbitrary string of bytes of some fixed length. A **map table**, stored on disk in a known location, relates logical to physical addresses. Allows for more flexibility

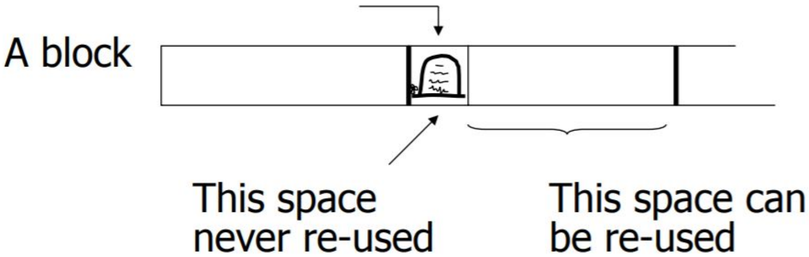
(+) If address of the record changes we need to update only one place.
(+) If the record is deleted, we can put a tombstone in the map table, and reclaim the space originally used by the record.
(-)Access takes longer due to **indirection** (need to fetch the physical address from the map table first, therefore 2IOs, first for map table entry, second for the record itself.)

Deletions

- On delete: can we remove record, possibly consolidate blocks (we would like to!), delete overflow blocks?
- Danger: pointers to record become dangling.
- Solution:
  - Physical : tombstone in record header = bit saying deleted/not. Rest of record space can be reused. But Tombstone cannot be moved (at least not till we are sure no one is pointing to this record!).
  - Logical : Map entry now has a tombstone instead of the address, the actual space where the record was can be completely used.

Deletions : Leaving a Mark

- Physical IDs



A block

This space never re-used

This space can be re-used

- Logical IDs

map

ID	LOC
7788	🪦

Never reuse ID 7788 nor space in map...

Insertions

Easy case: records not in sequence

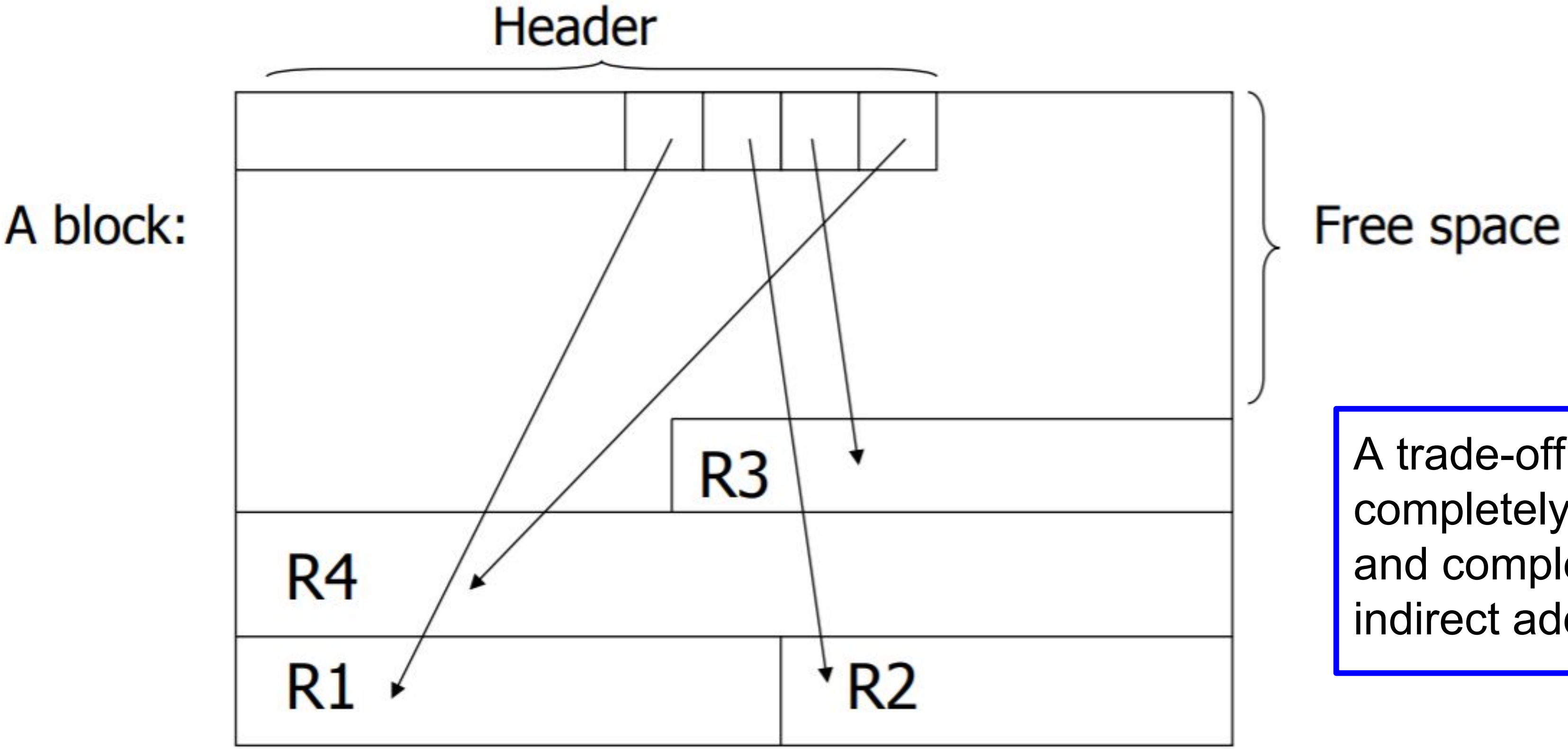
→ Insert new record at end of file or in deleted slot

Hard case: records in sequence

→ If free space “close by”, not too bad...

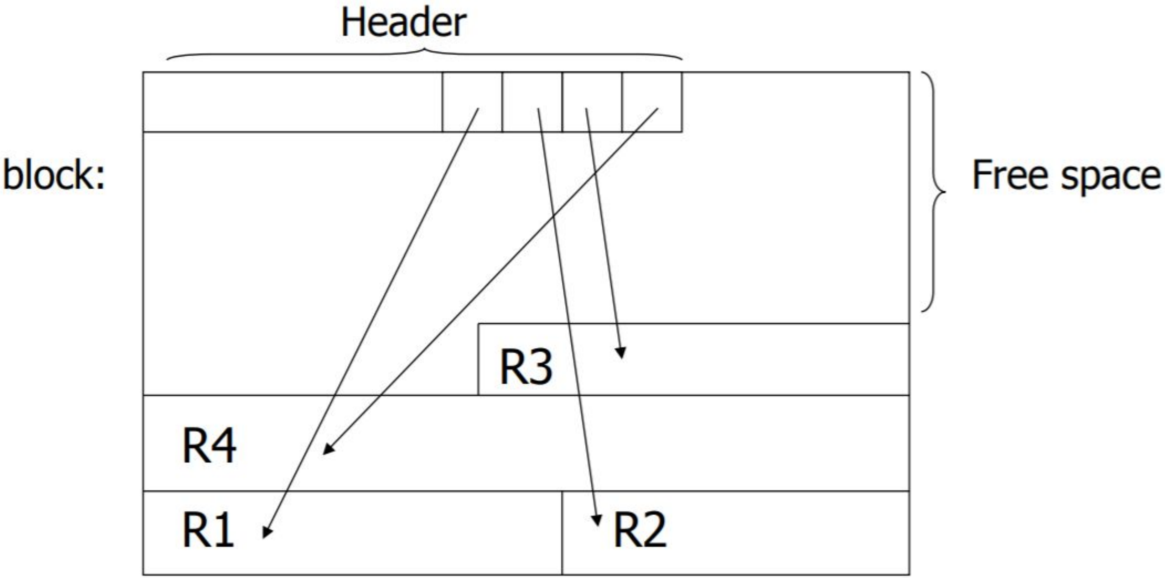
→ Or use overflow idea...

Indirection in Block : TID



A trade-off between completely physical and completely indirect addressing.

Indirection in Block : TID



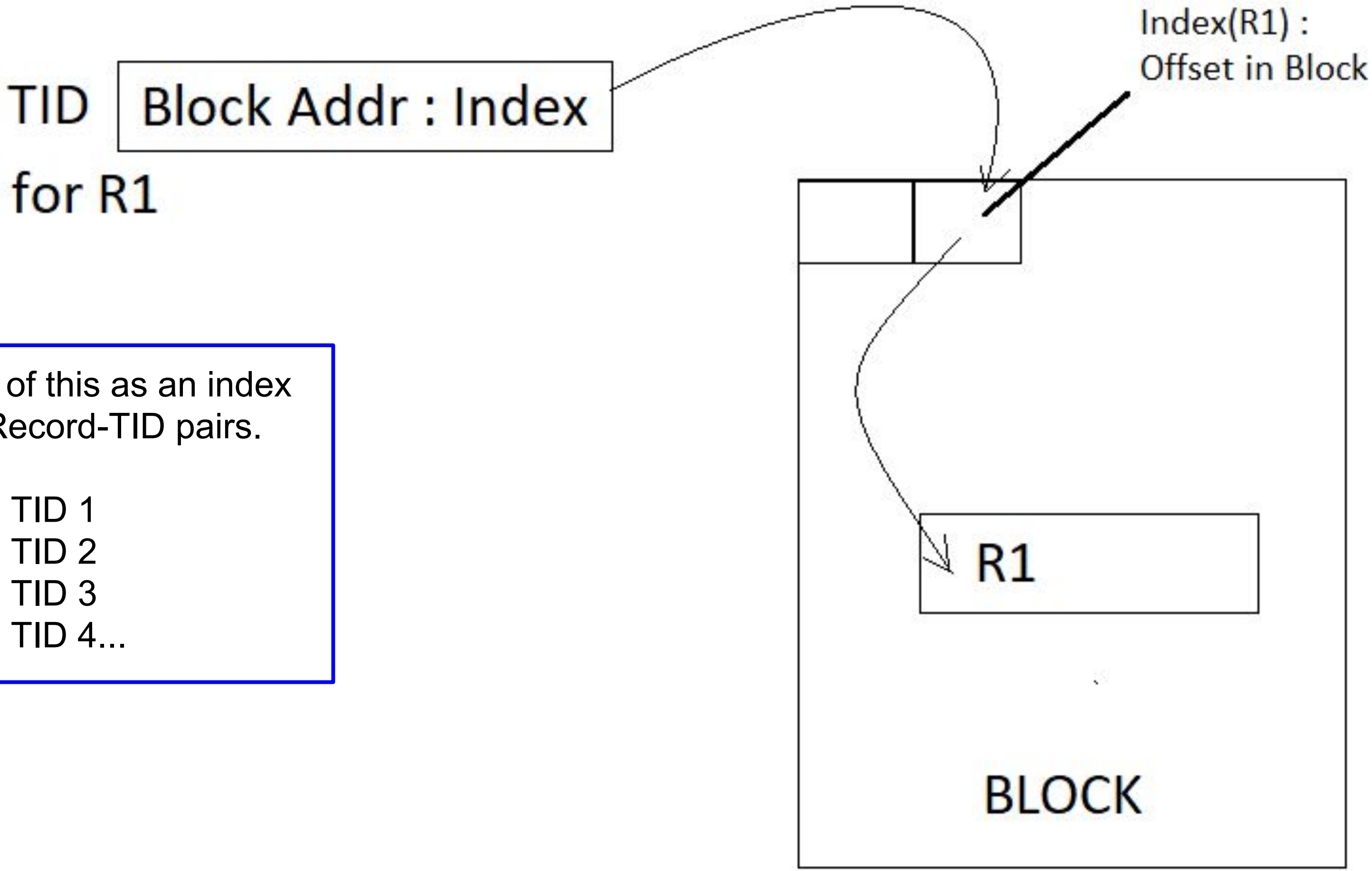
A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the tuple within its table. Address of a record = address of the block that contains the record + some index (offset information).

The block header will have the index mapped to the correct offset for the record.

TID

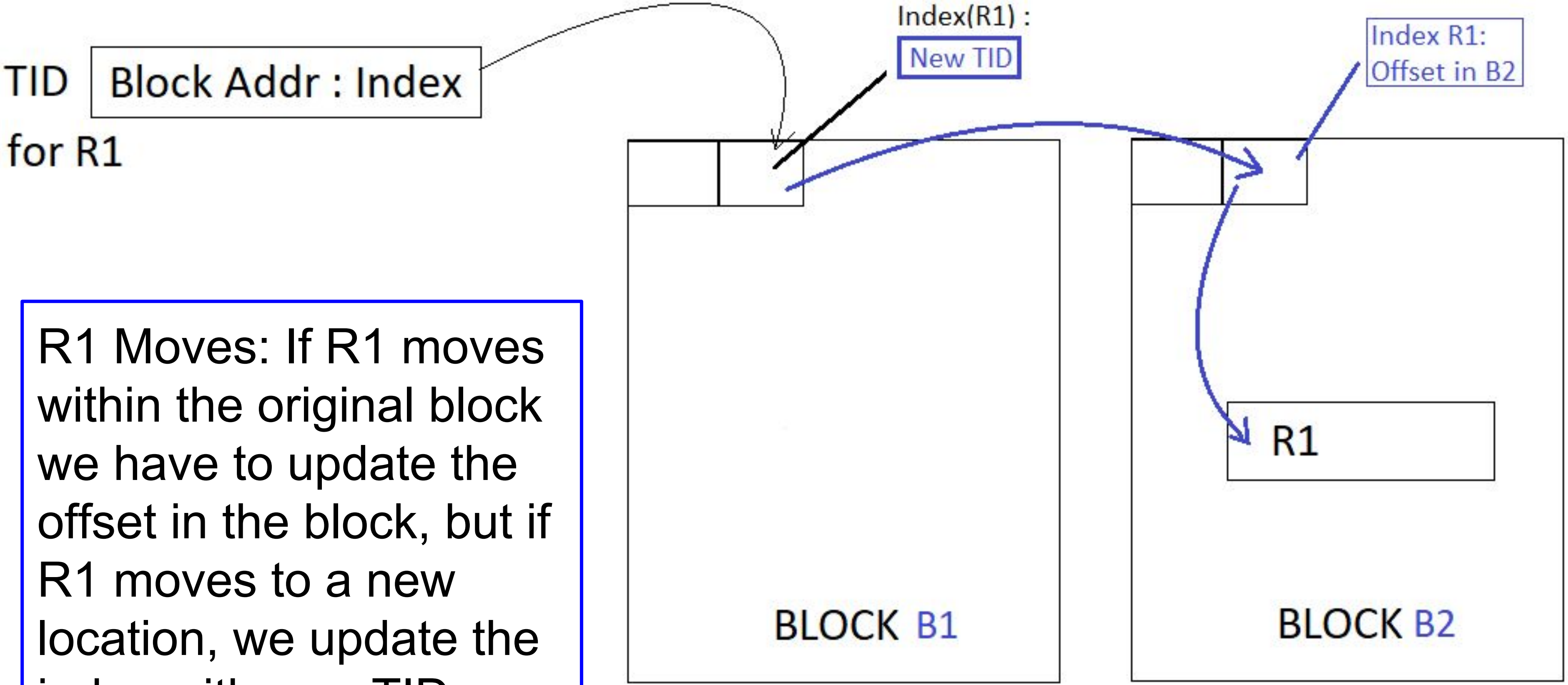
- Guarantees stability w.r.t. relocation within a block. Even if the offset of the block changes, only need to update the index.

- Guarantees stability w.r.t. relocation outside a block. Need to leaving a forwarding address at original location (now the index has the new TID instead of the offset). Update this forwarding address every time the block of the record changes.

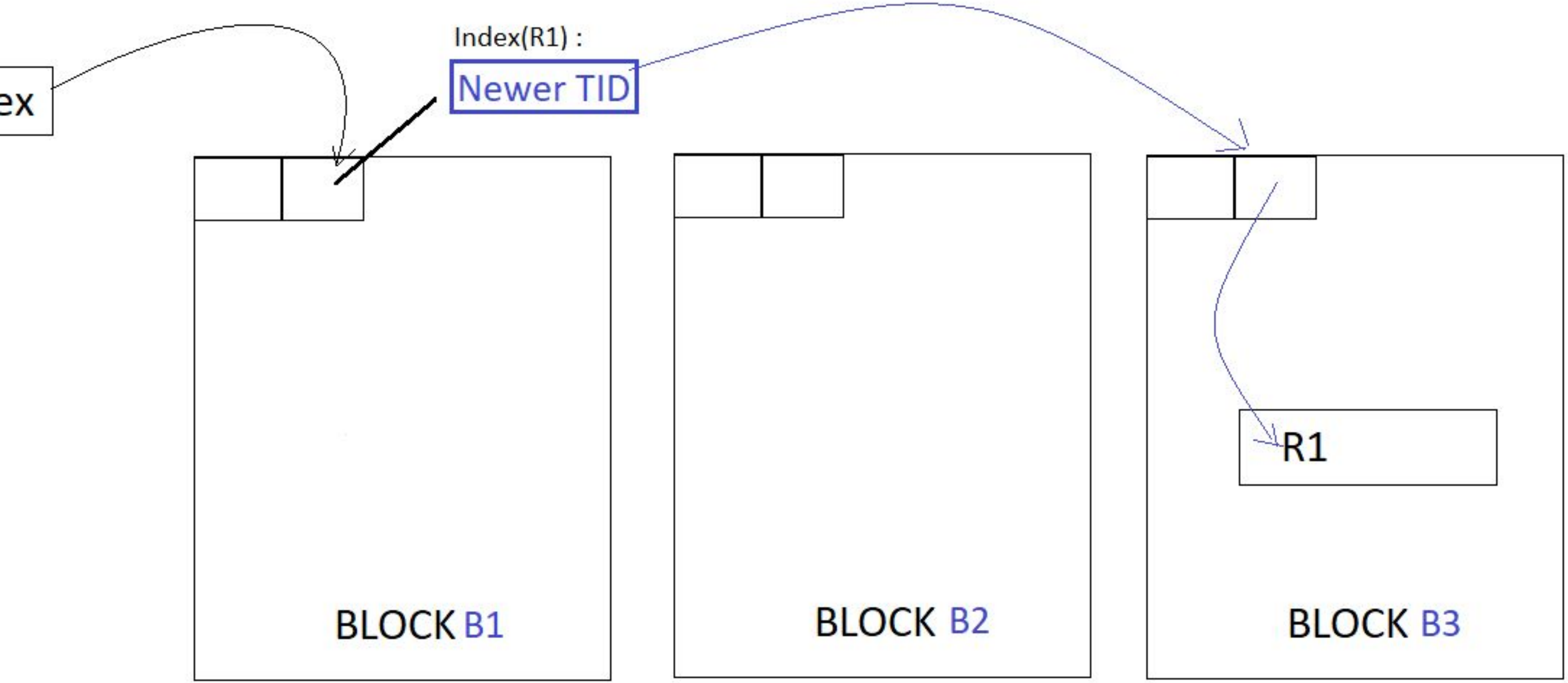


Think of this as an index with Record-TID pairs.

R1 ⇒ TID 1
R2 ⇒ TID 2
R3 ⇒ TID 3
R4 ⇒ TID 4...



R1 Moves: If R1 moves within the original block we have to update the offset in the block, but if R1 moves to a new location, we update the index with new TID



R1 moves yet again: update pointer on original page. Max IOs to fetch record 2 (when record has moved). Min 1 (when the record is in the original place).

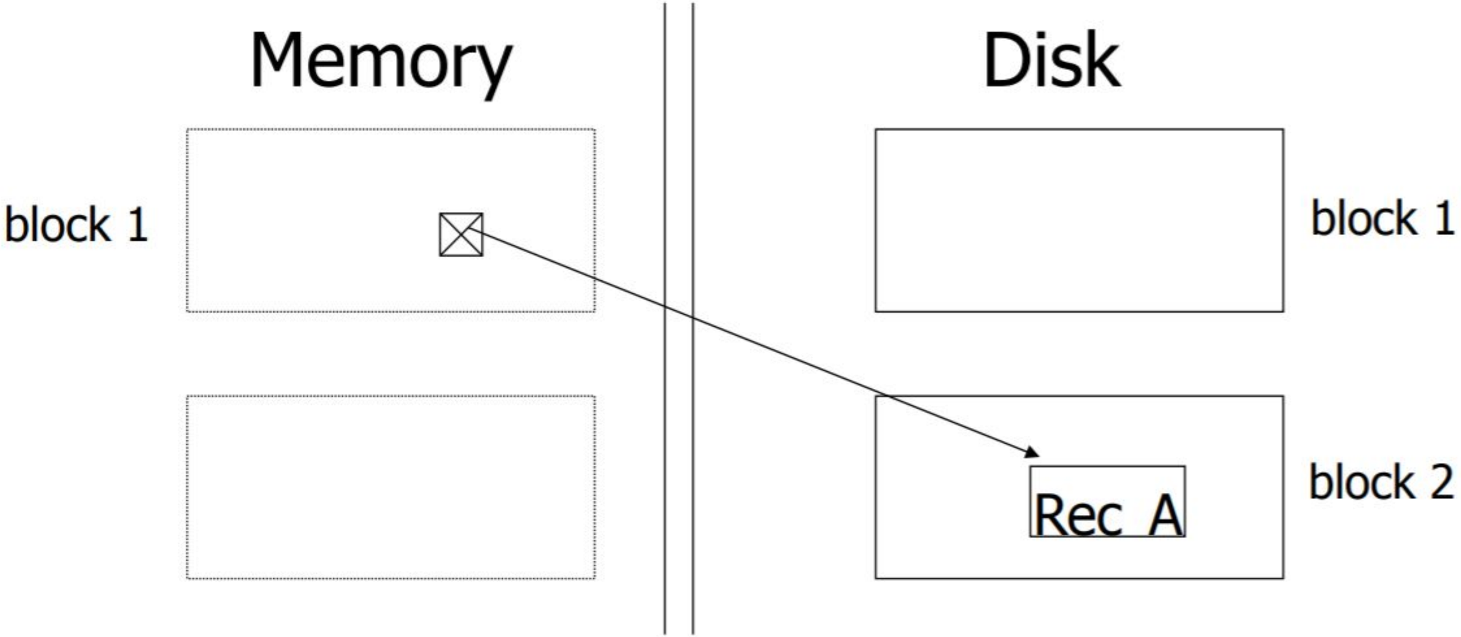
Swizzling

- Pointers are often parts of blocks/records
- Records moving in and out of memory can have two addresses, the actual physical database address and the memory address(the latter only when the block containing the record is buffered)
- When a record is in database or secondary storage, we can only use the database address, but while the record is in main memory it is cost effective to refer to it using the memory address.

Swizzling

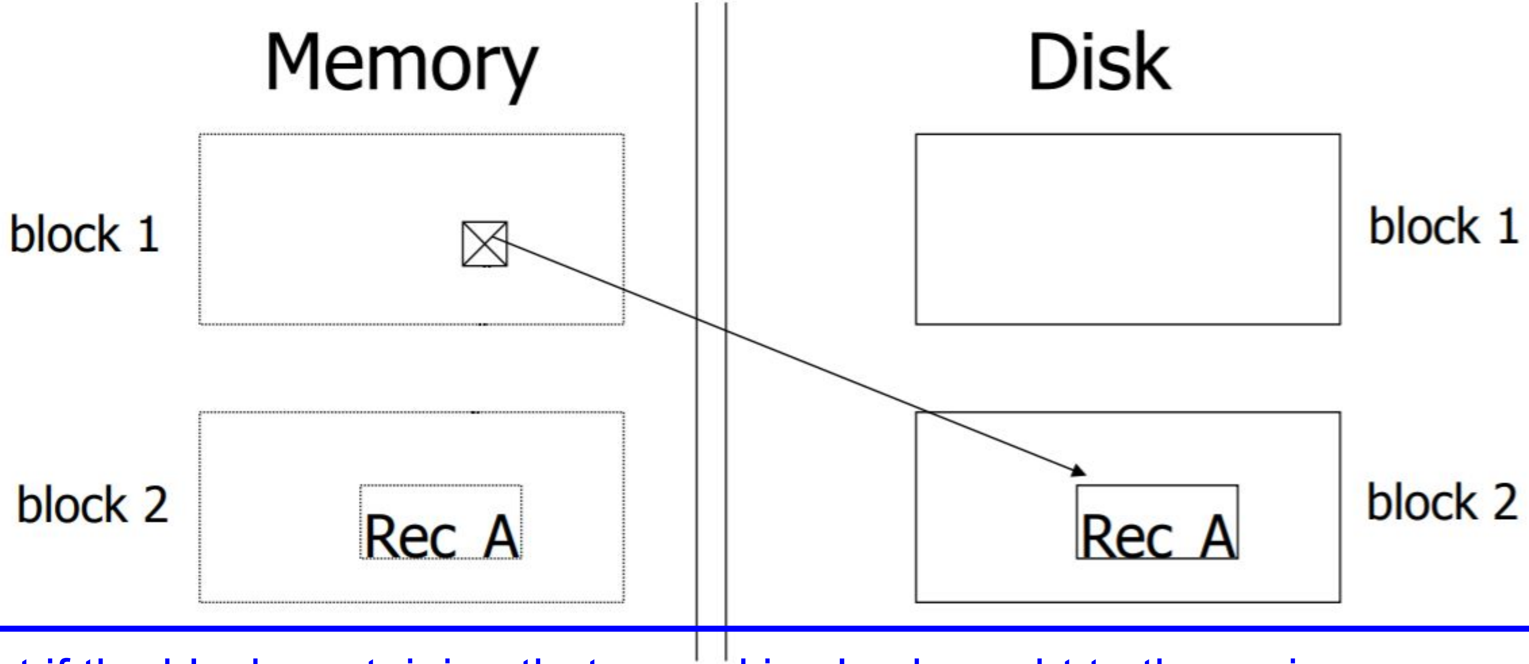
- When objects are brought in main memory, they are entered into a translation table (which maps the database address to memory address so that we know where the object is stored in the buffer) Note: Logical and physical addresses both refer to the database address, while the memory address in the translation table are only for the copies of the object in memory. Only those items currently in the memory are mentioned in the translation table.
- To avoid having to look up the translation table repeatedly for objects already in memory it might be more useful to ‚swizzle‘ all the pointers to this object while it’s in memory.

Swizzling



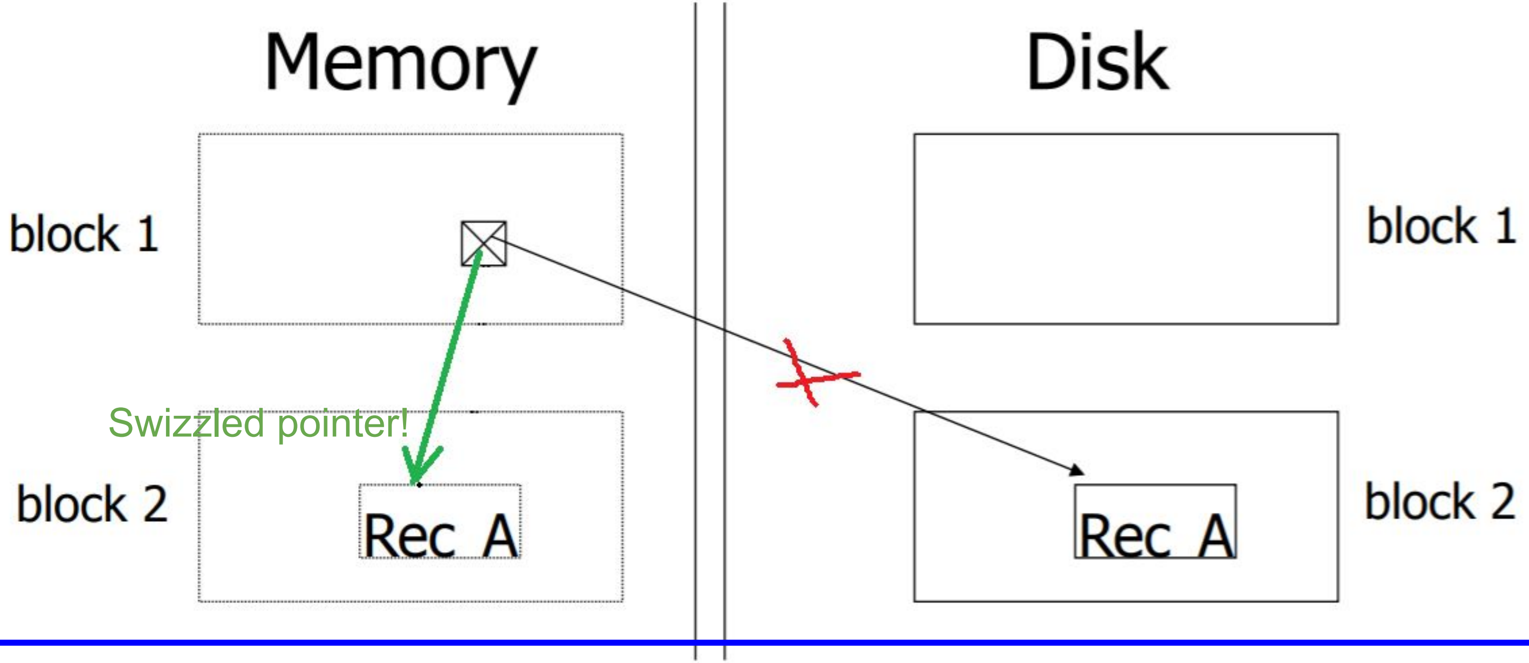
A block in Main memory is pointing to a Record on Disk.

Swizzling



But what if the block containing that record is also brought to the main memory. Should we still point to the actual physical address or to the new Main memory address (remember it will be much faster to follow the main memory address).

Swizzling



Pointer Swizzling: when we move a block from secondary to main memory, pointers within the block may be "swizzled," that is, translated from the database address space to the virtual address space.

Swizzling

1.**Never swizzle**. Keep a translation table of DB pointers à local (memory) pointers; consult map to follow any DB pointer.

•Problem: time to follow pointers (specially if a pointer is used multiple times).

2.**Automatic swizzling**. When a block is copied to memory, replace all its DB pointers by local pointers.

•Problem: requires knowing where every pointer is (use block and record headers for schema info).

•Problem: large investment if not too many pointer- followings occur.

3.**Swizzle on demand**. When a block is copied to memory, enter its own address and those of member records into translation table, but do not translate pointers within the block. If we follow a pointer, translate it the first time.

•Problem: requires a bit in pointer fields for DB/local,

•Problem: extra decision at each pointer following.

Swizzling

- Useful only if we are going to follow the pointer
- Pointers need to unswizzled before blocks can be returned to disk (therefore need to keep record of swizzled pointers)

Sheet 5, Question 2

Suppose that if we swizzle all pointers automatically, we can perform the swizzling in half the time it would take to swizzle each one separately. If the probability that a pointer in main memory will be followed at least once is p, for what values of p is it more efficient to swizzle automatically than on demand?

Assume cost of swizzling one pointer is = t

Since the probability of following a pointer is p, the cost for following a pointer for on demand swizzling = p*t

Automatic swizzling costs (in general per pointer) = t/2

We need to find out at what value of p, will automatic swizzling be cheaper, ie.

$p*t > t/2 \Rightarrow p > 1/ 2$