1. Arranging data on disks.

Database is mapped to a number of different files on disk.

- Simplest case: a table which is a collection of records.

Records are mapped onto disk blocks.

- Typically all attributes of a record are placed in contiguous byte positions of a disk block.
- Disk blocks are typically 4KB - 16 KB, even 32 kB.
- Certain attributes like images, videos can require significant more space (.jpeg, .mp4 etc). We consider columnar data organization a little later.

Starting with our simplest assumption Consider the table Movies.

```
Create Table Movies (
    title Varchar(200) Primary key,
    year int,
    length int,
    genre Varchar(20) );
```

Fixed length Records:

The simplest organization is to keep a file corresponding to the table "Movies." This file is divided into blocks, and each block keeps only records corresponding to the table "Movies".

(We will look at more complicated file structures where records of two relations can be stored together on a block - called multitable clustering)

In its simplest form, let us assume that the fields of a record are ordered as they have been defined in SQL.

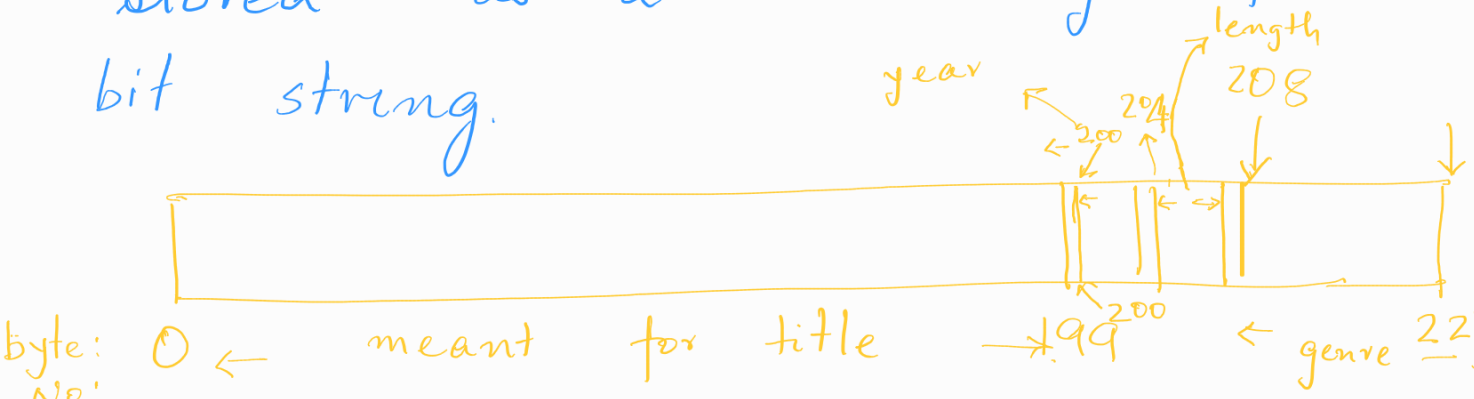
Attribute No.		Fixed number of bytes allocated.
1:	title	200.
2:	year.	4
3:	length	4.
4:	genre.	20.

Typically machines allow efficient reads/writes at multiples of 4 or 8. (word size)

Fixed length record size: each field is a multiple of 4. bytes. 228 bytes.

[Note: In many relational DB systems there is a part of the DB called "catalog". The catalog contains information about each attribute of each table, including its type, size or variable-size, etc.]

Coming back to our record organization, a typical record (fixed length) is stored as a 228 byte fixed length bit string.



We have a small issue - NULL values. It is possible that the attributes year, length and genre could be NULL. How do we represent this? In anticipation of a solution widely used for variable length attributes, we will add a "header" for the record and keep it in the beginning. To handle null values, keep a 1 bit entry: 1 for NULL and 0 for non-NULL value.

For "Movies" table there are 4 attributes, so for now it suffices to store 4-bits; however our assumption of starting at a word boundary gives the structure:

Variable length Records.

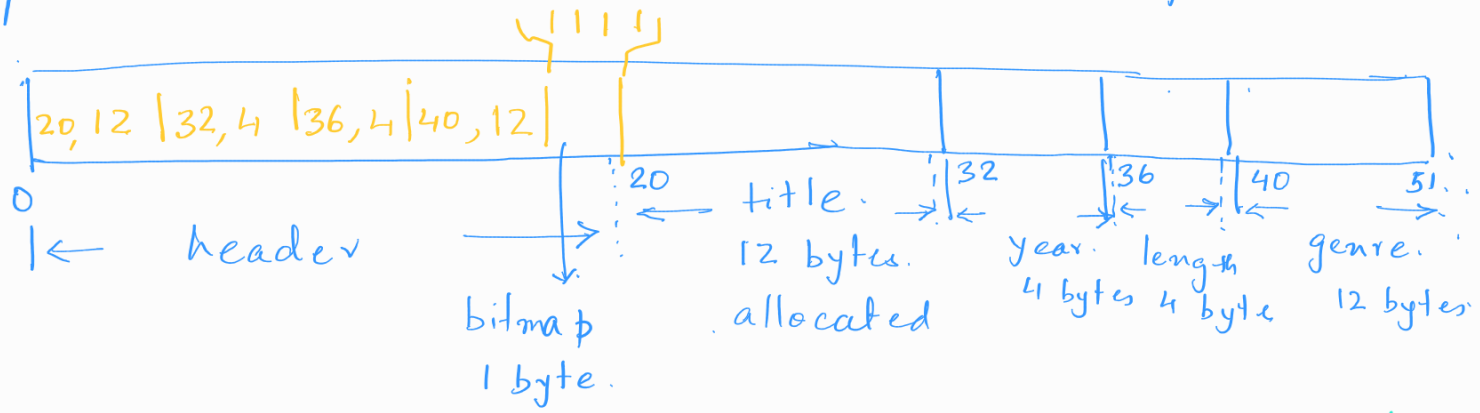
In the schema for "Movies", there are two attributes. title and genre whose lengths are variable and upper bounded upto 200 and 20 chars respectively. The header field is extended to include for each attribute

- starting byte position, and
- length of that field.

for e.g. say we have a record with the following field sizes.

1. title : 11 characters
2. year : integer (4 bytes)
3. length : integer (4 bytes)
4. genre : 9 characters

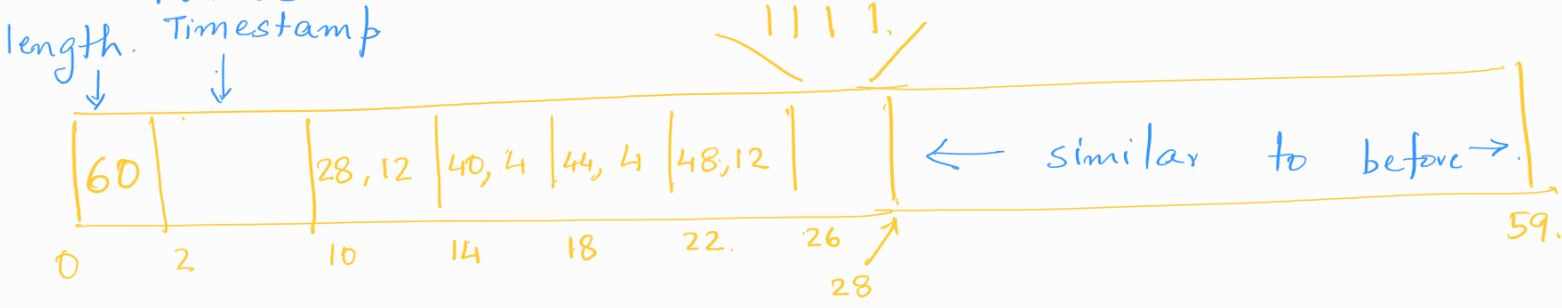
The header keeps track of byte positions allocated to the fields:



Comments: 1. A small excess information may be noted. A NULL field will have its corresponding field length 0; vice-versa a positive length field has a non-NULL value. So really, the NULL bitmap vector is not needed.

2. An additional 2 bytes to store the length of the record is kept in the header for direct access to its length without requiring calculation.

3. Often a timestamp for last update time is also kept.



Storing Records (variable length) in a block.

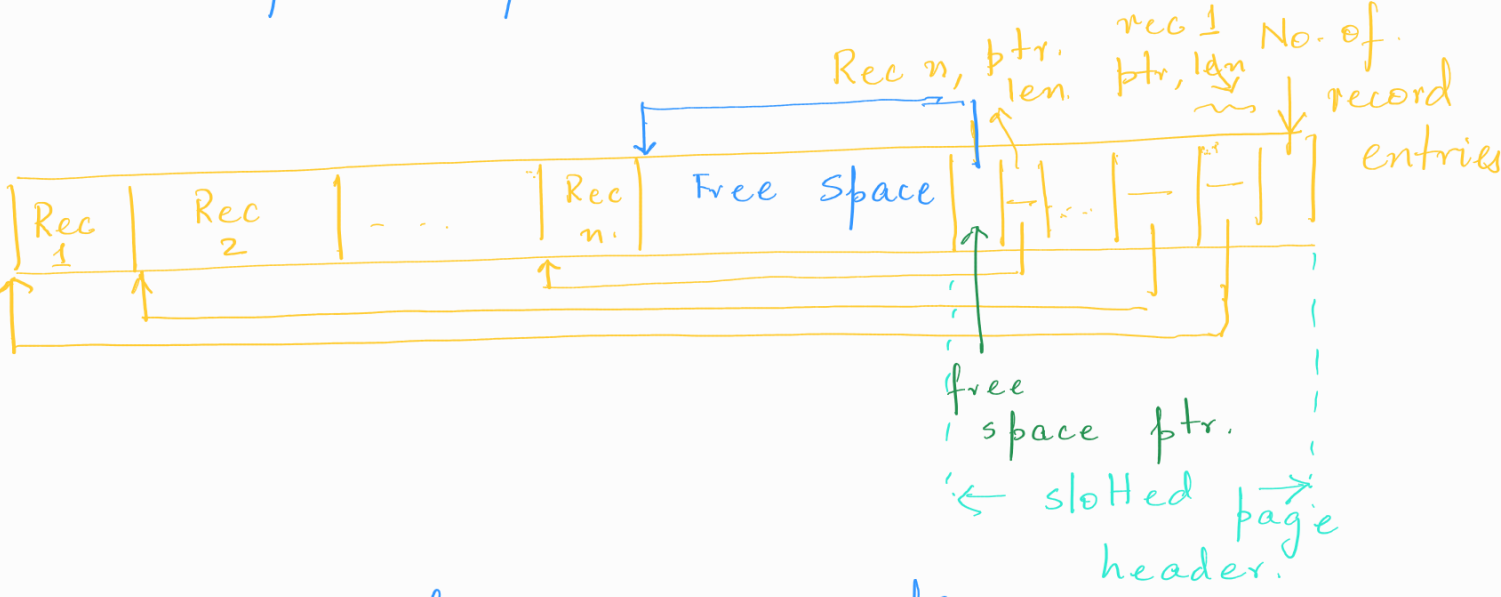
Assume that the block size is fixed, say at 16 KB. (for e.g.). We now consider how records are stored within a block. Closely related to this is the notion of record addresses and block addresses in a DB.

Slotted Page Structure

Each block has a header structure that keeps information:

1. The number of record entries in the header. Each record stored on this block has an entry. (Momentarily, assume that there is one entry in the header corresponding to each record in this block and that's all. It's not all, but we will look at this in a minute)
2. An array of entry, one entry per record (in this block). Each entry contains 1: the starting position of this record. and
   2: its length.

3. The starting position of the free space in the block.



Here, we have n records. Records are stored from the beginning of the block; the slotted structure is stored at the end of the disk. This allows for free space in between.

1. New records can be inserted in the block by (possibly creating) a new entry and reorganizing the records within the block. Both the header and the record(s) positions may be updated.

2. Existing records may be updated. This may require a length change and may require a re-organization of this block.

Deletion of a record:

1. The space it occupies is freed.
2. But, its entry is not fully deleted. Instead, its pointer (starting byte position) is set to 0 (which is obviously invalid) or to -1 or to. blocksize (current valid byte range is 0.. blocksize - 1).
3. Records are reorganized so that free space is recovered and is made contiguous. This requires updating the slot entries.

* Due to deletions, the number of slot entries may be more than the # of actual records stored.

What is the main advantage of this slotted page structure?

Let's understand the addressing mechanism first. To simplify things greatly, let's assume that the database resides one one disk. We will generalize immediately after this. At a conceptual level, say in "Movies" table, a tuple (record) is simply identified by its "title." value. At the hardware level, the block containing

the record is unambiguously addressed, for e.g.

(cylinder No., track No, sector No)

of the first byte of the block.

Database Internal structures, e.g. Indexes often work with pointers from one record to another. So we have to consider how to address a record. The slotted page technique gives an elegant answer. Within a block, a record occupies a certain slot. It could be the 1st, or 5th or 3rd record in terms of the slot number it occupies in that block. Note that once a record is inserted into the block, its slot number does not change, despite its position in the block changing due to updates within this block. The address of a record is

(block address, slot number)

This maintains its address despite changes to its position within the block.

Comment: An issue we didn't consider is that a record may be updated and its size increases so much that there is not enough free space to fulfil this requirement within this block.

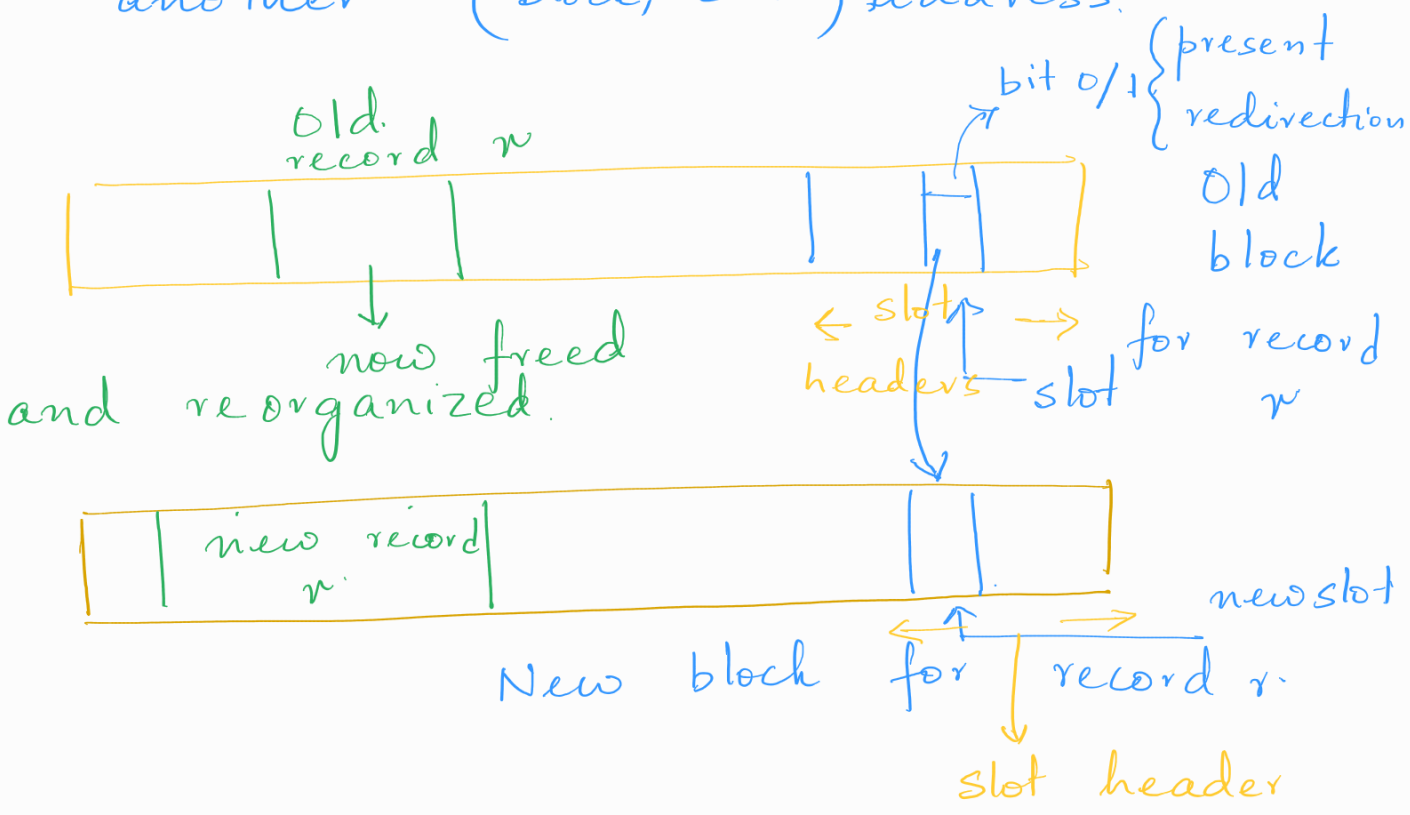
This is typically solved as follows and generalizes the notion of pointer to the beginning of the record in the block. (Again, keeping things simple).

1. The new record can be stored in a new block or the database system identifies another block with sufficient free space to store the new updated record.

2. This new record is stored in this new block and gets a (Block id, Slot No) address.

3. In the original block, the corresponding slot entry points to the (new block, newslot) address. The slot entry field for pointer is large enough to store this,

A bit in each slot entry tells if the record is in this block or is redirected to another (block, slot) address.



Addresses in Client-Server Systems

Commonly, a database system consists of a server process that provides data from secondary storage to one or more client processes. The client processes run applications using the data.

Client applications may be running on the server machine,

or there can be many clients distributed over many machines.

Client apps conventional "virtual address" space, typically 32 bits or 4G different addresses.

The DBMS (or OS) decides which parts of the addr. space are currently located in main memory. Hardware maps virtual addr to. main memory physical addr.

> We will not make a distinction between virtual and physical addresses and think of the client addr. space as if it were virtual address.

Server data lives in a database address space. The database address may be sourced as follows.

1. The database may be using many hosts, for. e.g. it may be using 1 million hosts. Each host is identified by an identifier

– in its simplest form it is an IP address. [It could be more general and hierarchical]

This would require 32 bits (say)

2. To each host, many disks may be attached. An identifier for the disk on this host has to be specified

3. Within the (host, disk) specification, the

(Cylinder No, Track No, Sector No.)

gives the address of the block.

4. Add to this the slot number of the record stored in this block (or, has redirection pointer); this specifies the physical address of the record in the database address space.

This can require 64 – 128 bits or more.

A comment on the slotted page structure.

1. Deleted tuples retain their entry in the slot header. The pointer is set to 0, etc. to signify deletion. Why is this needed? Because there may be many records in the DB that may be pointing to this record. The record which is deleted — if its slot is now used by some other record, then all pointers to the deleted record will point to a new record, which would be generally incorrect. To avoid this issue, the "tombstone" of the deleted tuple is maintained.

Pointer Swizzling:

Motivation: A disk block is brought in from disk and stored in client memory. Records may have pointers in them as fields: these pointers are pointing to different records or blocks.

Typically, in relational DBs, records may not have pointers. But there are.

indices, and records inside an Index File have lots of pointers.

Also, object-relational DBMS allow pointer / reference attributes.

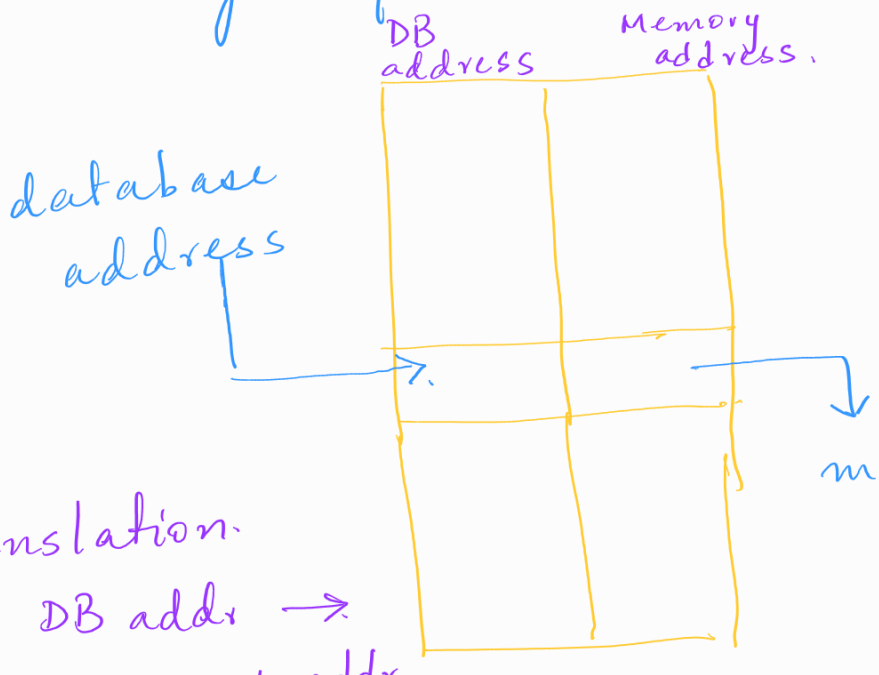
We now study the management of pointers as blocks are moved between secondary storage and main memory

While in secondary storage, every block, record, object etc. has a database address. If it is in memory, then it has a (temporary) (virtual) address.

Suppose a block has been currently copied into memory. (a) This block, and records within this block can be referred both by their DB address as well as by its memory address.

(b) Record(s) in this block may be updated by the client application. Then, all accesses to the block must be to its main memory copy, rather than to its secondary storage copy, which is currently outdated.

This brings us to the question: how do we resolve a database address when the block (or record) is in main memory. One solution is to have a (database address, memory address) translation table in the virtual memory of the client.



Translation Table: DB addr → memory addr.

All addressable items currently in this client's memory (virtual) has an entry in the translation table

Suppose a block B is in memory and contains a record r. This record r references a record say s which is in block C. Say block C is also

in memory. Whenever the application follows the pointer reference from r to s, the database system has to consult the translation table to ultimately resolve the memory address of s.

This step is quite expensive. Instead, several techniques have been developed to replace references to database addresses within a block, i.e., database pointers, by virtual addresses (i.e, memory pointers) This is called Pointer Swizzling.

A pointer in a block in main memory consists of.

1. A bit: Swizzled or Not Swizzled.
   Is it a memory addr. or a DB address?
2. The database or memory pointer, as appropriate.

Example: Block B has a tuple r with pointers to record s in Block C and t in Block D. The figure shows the

pointers as on disk. Suppose Block B is copied and then Block C is copied.

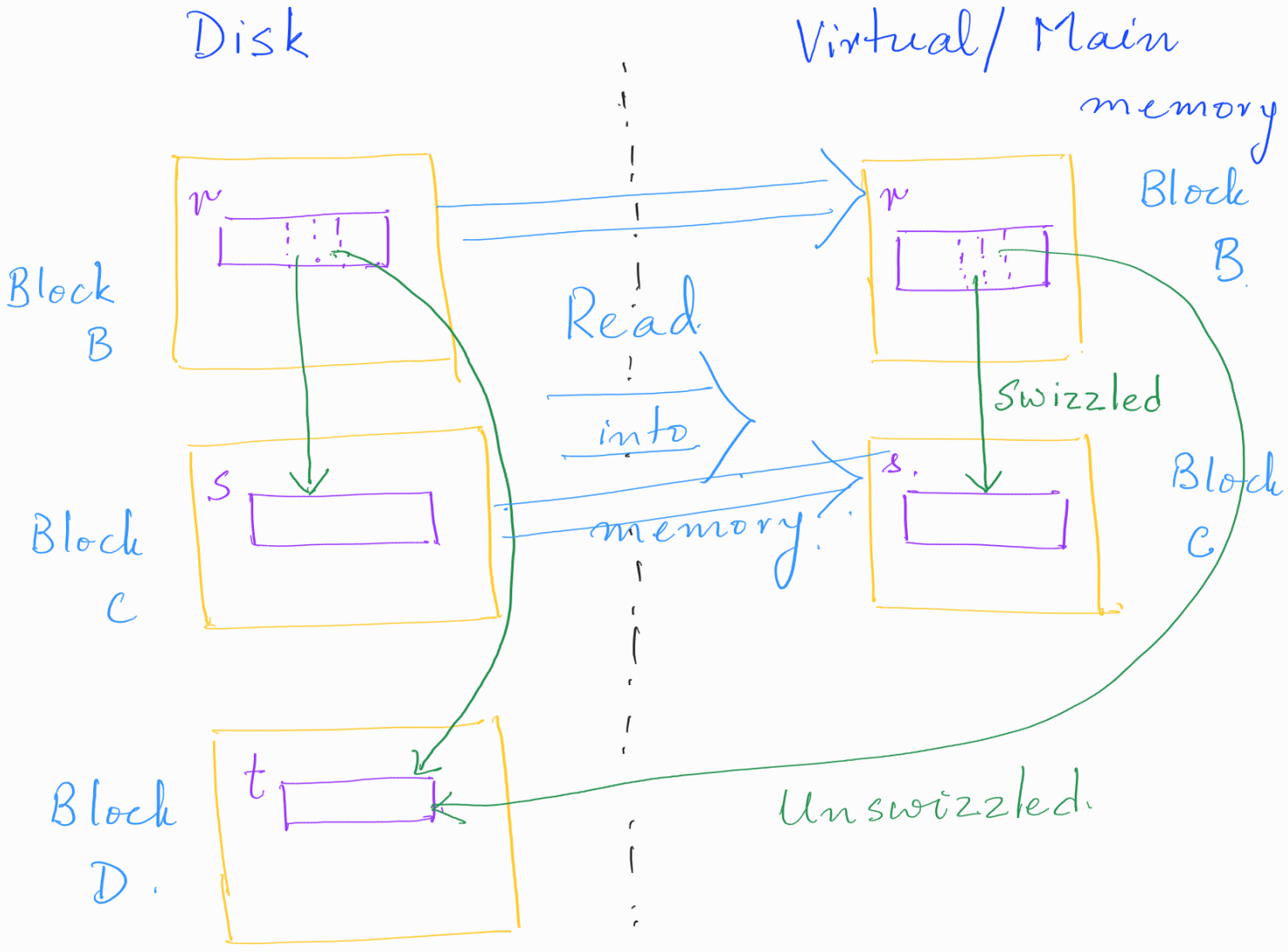


Fig: An illustration of pointer swizzling.

The figure shows that the pointer in Block B from tuple r to tuple s in Block C is swizzled, i.e., the pointer is replaced by a virtual memory reference. However Block D is not in memory, So the

reference from tuple r to tuple t in Block D remains unswizzled, i.e, the pointer is a database address. Should Block D be brought to memory later, this reference can be swizzled.

Swizzling strategies:

Several swizzling strategies can be used:

1. Automatic Swizzling.
2. On demand swizzling.
3. No swizzling.
4. Programmer control over Swizzling.

As soon as a Block B is brought into memory, we locate the following:

Automatic Swizzling

1. The addressable items within this block, e.g. all records and block itself.

2. All pointer references from within this block B.

Momentarily, assume that 1 and 2 can be done efficiently, if necessary, add a little structure to the block header. (We will return to this).

The action would be the following:

1. All DB addresses corresponding to addressable items within block B are now entered into the

DB → Memory translation table.

Corresponding memory addresses of these records / addressable items are available directly from the memory copy. This creates the translation table entry for these DB addresses.

2. Now, we also look up all the DB pointer references within block B.

We look up the translation table for each database addr. e A. If A has an entry in the translation table, say Amem, then in the block B in memory, the reference to A is replaced by Amem and its swizzling bit is set to 1.

Otherwise, if A does not have an entry in the translation table, the corresponding pointer reference in the block B remains unswizzled; i.e., remains a

DB pointer.

Following pointers :

During a database application execution, say that a block B is accessed and we follow a pointer A.

a) if A is swizzled, then we directly access the (virtual) memory addr. as stored in the pointer structure.

b) However, if A is not swizzled, we first check the translation table to see if the disk block referenced by A (and the referenced record of A) is in memory. If so, the pointer A is swizzled and we now have the memory pointer corresponding to the record/object referenced by A.

c) If A is not swizzled and the referenced DB addr of A is not found in the translation table, then the database system follows the database pointer A and brings its containing block C

into memory. Once block C is in memory, pointer A is swizzled.

Swizzling on Demand:

Basically, leave all pointers unswizzled when the block B is first brought into memory. That is,

a) the translation table is updated for every addressable entry in the block B, along with their memory equivalents.

b) But, all pointers referenced within block B are not swizzled.

c) If a pointer address A is referenced within B and accessed by the database program, then we follow A onto disk, bring its containing block C into memory and swizzle pointer A (as described in automatic swizzling).

	Automatic Swizzling	On Demand.
Pros:	1. Gets all pointers swizzled as quickly as possible.	1. Delays swizzling as per necessity.
Cons:	2. Perhaps many swizzled pointers will not be followed.	

On demand Swizzling : an interesting option.

1. Arrange the DB pointer field to look like an invalid memory address.
2. This allows the computer to follow any pointer as if it were a memory address.
3. Unswizzled pointer causes a memory reference to an unresolved memory: causes a hardware trap.
4. If DBMS provides a function invoked

by the trap, then this function carries out the "swizzling", as described above.

No Swizzling:

1. Possible never to swizzle pointers.
2. We do need the translation table, and pointers are followed in their database addres. form.
3. Advantage: No Pinning of blocks is needed. [Pinning and unpinning is described later]

Programmer Control of Swizzling:

1. Should we swizzle a block's pointers when it is first brought into memory, or should we not?
2. This depends on whether the pointers in the block are likely to be followed. If so, then swizzling is a good possibility, otherwise on-demand may be better.

3. Programmers are more aware of the possibility, also.

a) the top-level index blocks of B^+-trees are swizzled,

b) blocks loaded into memory and then likely dropped from memory, may not need to be swizzled.

▷ Returning Blocks to Disk:

When a block is moved back from

memory to disk, two things must happen.

1. If the block contains pointers (address references) that have been swizzled, then they have to be unswizzled.

2. If there is any other block in memory that refers via a pointer to this block, and the pointer has been swizzled, then it has to be unswizzled. All such pointer references to within this block must be unswizzled before it can be written back to disk.

The unswizzling operation has to be done efficiently. The translation table.

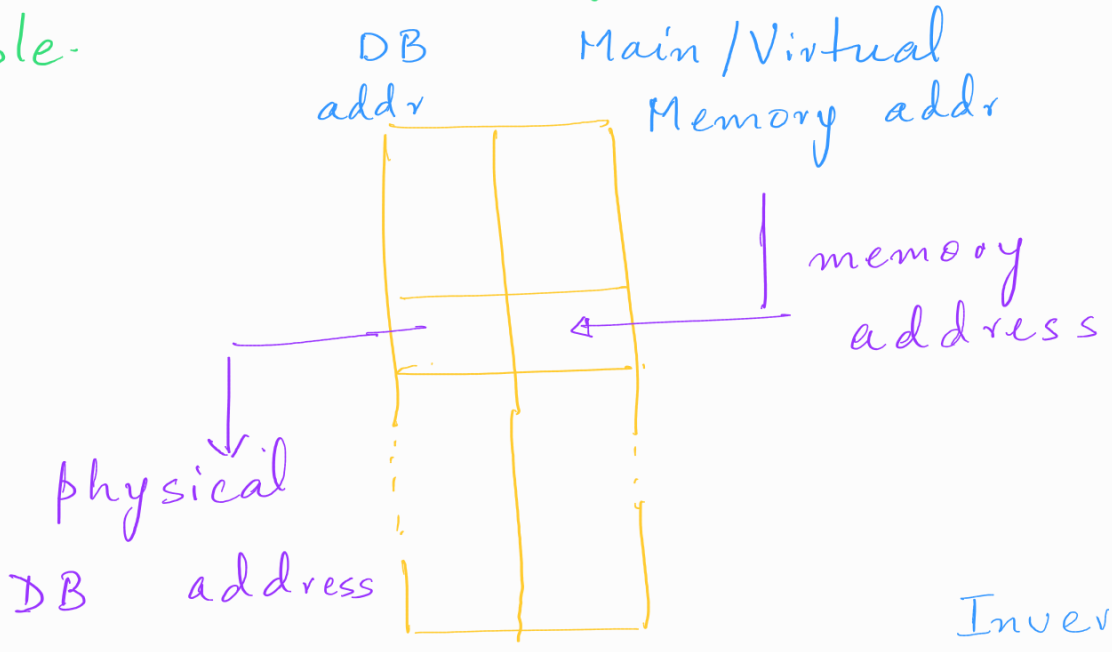


Fig: Unswizzling operation: Inverse mapping.

The unswizzling operation requires replacement of the memory addr by the database addr. This requires us to traverse or search the translation table in the reverse direction. Given mem addr → find DB addr.

In the normal app computation scenario, on-demand swizzling, or referencing a database address requires search in the forward direction. Given DB addr → map it to mem addr.

This shows that we need to support two indices on the translation table.

1. DB addr → Mem Addr.
2. Mem Addr → DB addr (reverse lookup)

Pinning Blocks

The notion of pinning a block in memory is akin to placing a pin on a note onto a board. It means that as long as the block remains pinned, it cannot be written back to disk. The notions of pinning and un-pinning

arise due to a number of reasons, e.g.

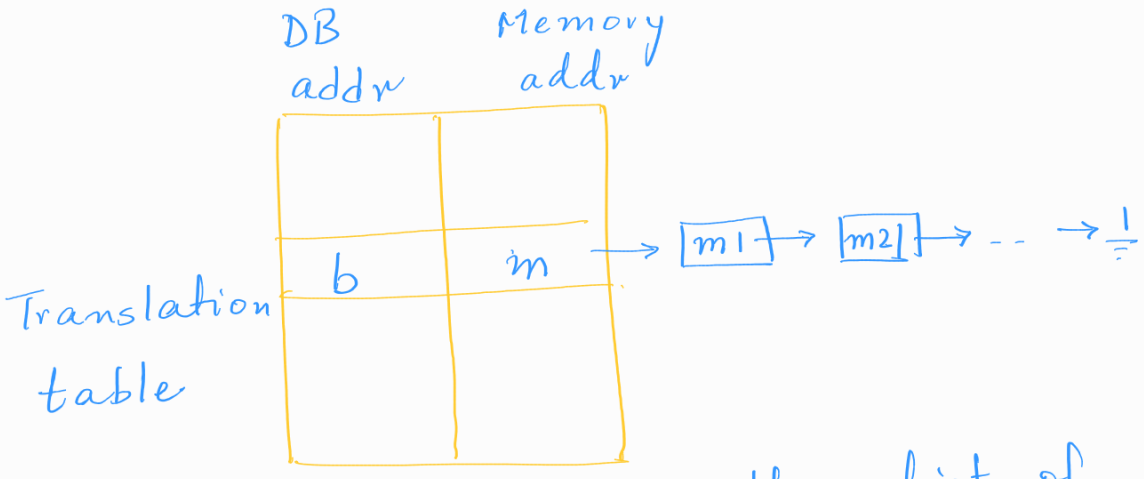
1. Pointer swizzling
2. Concurrent access by multiple threads on the same client to the shared database DB memory buffer pool.
3. Aspects of the Recovery Manager submodule of the DB System.

We look at aspects 1 and 2; 3 is studied later.

1. Suppose a block B is in main memory (DB buffer.) and there is some reference to a record/object contained in B that is swizzled. So the reference to this record is a memory address. This is done by pinning block B.
Unless all references to records/objects residing in B are unswizzled, block B remains pinned and cannot be written back to disk.

There must be a way to record, for each memory addr, all memory addresses list that refer to this addr. This can be done using the

translation table



Here m1, m2, ... is the list of memory addresses that refer to memory addr. m.

A second technique uses the fact that often, the database address is significantly longer than the main memory address. Consider a (b, m) address pair from the translation table. Now consider an address reference to record addr b that has been swizzled. This reference occurs in some block B' in some record r' within it. The corresponding field for this addr. reference to b inside r' has enough storage space (bits) to store database address. By swizzling, instead of the database addr, it is replaced by m. However, we have enough bits left: So we add a pointer to the

next occurrence of this pointer m. That is, we create a linked list of pointers to m, but it is stored in the space used for the pointers themselves.

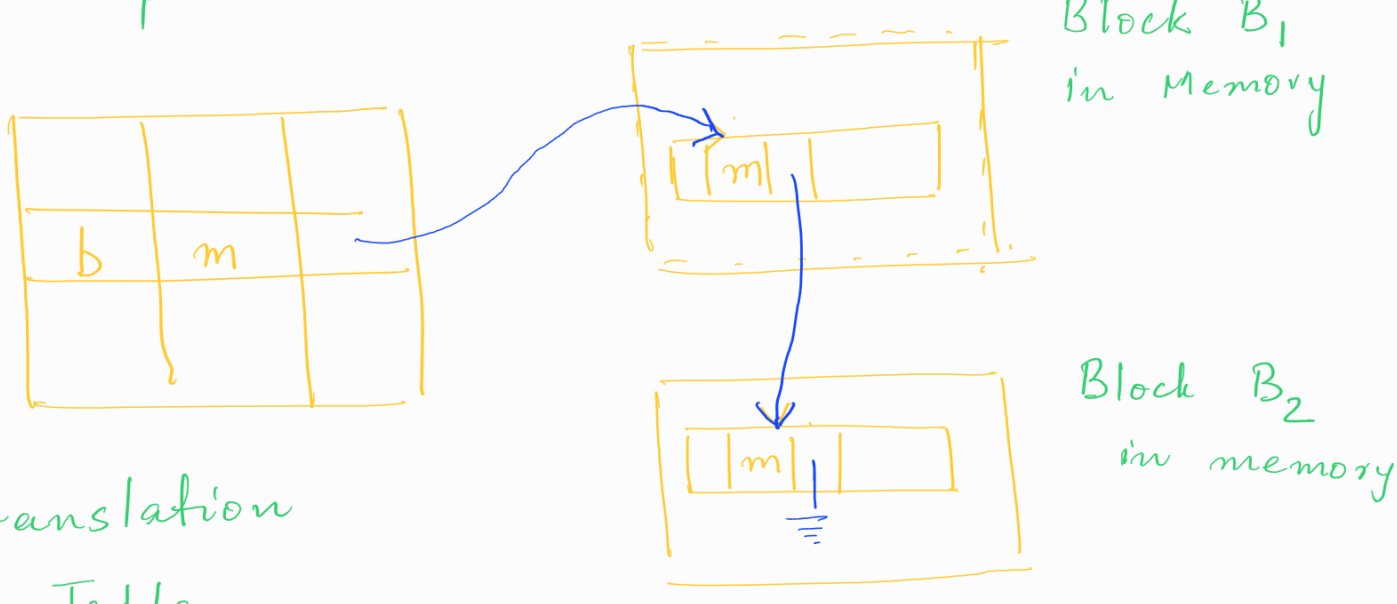
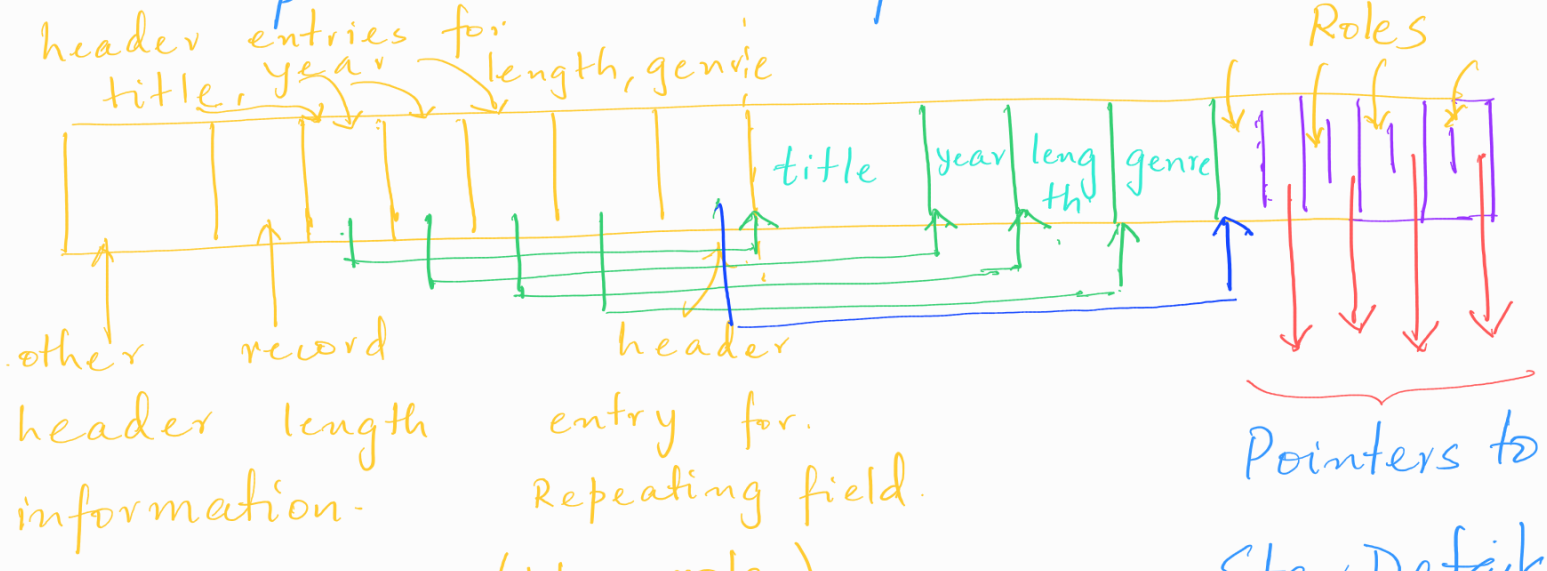


Fig: Linked list of occurrences of a swizzled pointer.

Variable length Records with Repeating fields.

E.g. Consider our "Movies" table example. We also have the table "StarsIn" (name, title, role). It is possible to organize the file for the "Movies" table so that for each movie record, in addition to

its fields, title, year, length, genre, we keep pointers to all stars in the movie, by pointing appropriately to the record in "StarDetails" table file. Also, for each star in the movie, along with this pointer, we keep the value of "role" attribute.



Header structure is slightly altered to keep track of repeating field information: its header bits keep track of its attributes: in this case, the attributes are role: and pointer. For role, its length is stored in the header. (Varchar(50)).

Records spanning blocks:

A record may span two blocks, or sometimes more, because (a) there wasn't enough space to store it, or, b) initially it fit into a block but then the updated version couldn't fit. In such cases, (1) a record may span multiple blocks and the fragments of the records on different blocks are organized into a linked list (doubly linked).

Also, (2) the header structure for the fragment record needs to keep a bit on, saying it is fragmented, and (3) its fragment no, and (4) whether it is the last fragment of this record.

Typically database implementations keep "room" within a block to allow tuples to grow. For. e.g. typical occupancy may be $\leq 80\%$

BLOBs (Binary Large Objects).

Fields may be large, such as images (.jpeg, gif), video (mp4, etc.), signals (audio, radar etc.). These are called

binary, large objects or blobs.

We need to revisit some storage and retrieval issues.

1. Storage; Blobs must be stored as a sequence of blocks, preferably consecutively on a cylinder, or on multiple cylinders on disk.

For speed of retrieval, it may be necessary to stripe the blob across multiple disks. This would allow several blocks of blob to be retrieved simultaneously, increasing the retrieval rate by a factor approximately equal to the striping factor.

Retrieval of Blobs: E.g. for a 2 hr movie blob, the client requests the movie to be played, the blob is shipped consecutive blocks at a

time at the rate necessary to play the movie.

Indexing may be needed: E.g. if client requests to see the 45th minute of this movie, the DBMS needs a suitable index structure (e.g. index by seconds) on a movie blob.

COLUMN STORES.

An alternative to storing tuples as records is to store each column as a record.

E.g. Consider the relation

X	Y	Z	W
a	b	g	h
c	d	i	j
e	f	k	l

The column store model keeps a table for each col: There are 4. tables: Col X, Col Y, Col Z, Col W.

Tuple id	X
1	a
2	c
3	e

Table ColX

Tuple Id	Y
1	b
2	d
3	f

Table ColY

Tuple Id	Z
1	g
2	i
3	k

Table ColZ

Tuple Id	W
1	h
2	j
3	l

Table Col W.

Advantages of column representation.

1. All values in a column table are of the same type, so it is much more amenable to data compression.

   E.g. Consider gender attribute in StarDetails. Normally it would take 1 word = 4 bytes in a record. Column Table ColGender would store 1 bit for every star name, compressing by ×32.

2. Column-based storage is relevant if most queries need to access all, or a significant part of the column values in that column. These "analytic" or "OLAP" queries may benefit from it.