Block and Record Address / Variable Length Records

CS157B

Chris Pollett

Feb 9, 2005.

Outline

- Client-Server Systems
- Logical and Structured Addresses
- Pointer Swizzling
- Returning Blocks to Disk
- Pinned Records and Blocks
- Records with Variable Length Fields
- Records with Repeating Fields
- Formats
- Records that do not fit in Blocks
- Blobs

Issues with Addresses

It is useful to know how record and block addresses work for several reasons:

- We will work with addresses a lot when talk about access structures like addresses.
- When a block is in the buffer the address can be taken to be the location of its first byte in virtual memory.
- However, when the block is in secondary storage, its address is according to the DBMS filesystem.
- There is also a trend to use intermediary ``object brokers'' to facilitate independent creation of objects

Client-Server Systems

- A DBMS typically consists of a *server* that provides data from secondary storage to one or more *client* processes that manipulate the data.
- The server's data lives in a *database address space*.

Client Server Systems (cont'd)

There are several ways to represent addresses in the database address space:

- **Physical Addresses** - byte string that let one figure out where in the secondary storage the data lives. Might consist of:
  - The host to which storage is attached
  - An identifier for the disk or device
  - The cylinder number
  - The track within the cylinder
  - The block within the track
  - The offset to the beginning of the record
- **Logical Addresses** - in this scheme each block or record has a ``logical address'' which is a string of bytes of some fixed length. Then a *map table* is used to look up where on the disk this corresponds to.

Logical Addresses

Having logical addresses means one also needs a map table. Why would one want to use logical addresses?

- Easier to move records/blocks around.
- This is especially useful when have access structures and so might have several pointers to the same block. It allows one to update just one place.

Structured Addresses

- Often one uses a logical address for the block and then have a physical address (a byte offset) for the record. This is called a **structured address**.
- The start of a block then usually has an offset table that holds the offsets for the record within the block:

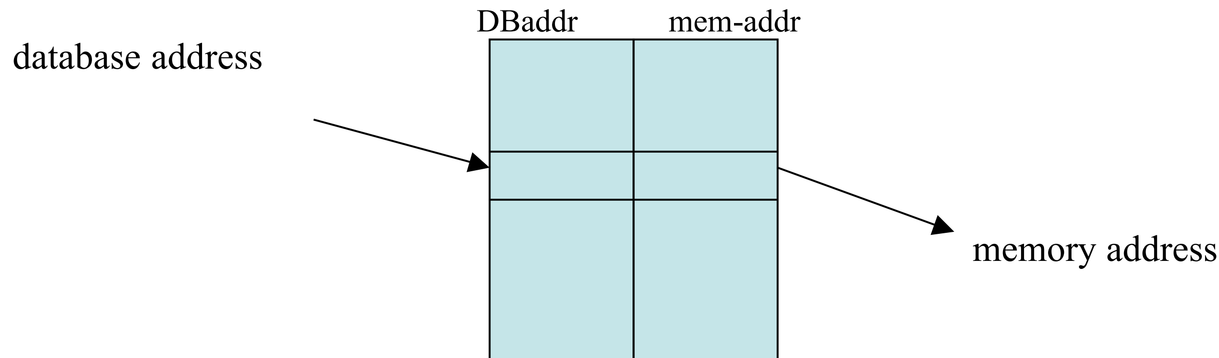
Advantages of this Scheme

- We can move the record around within the block by adjusting the offset.
- Good if have variable length records
- If have room can allow for records to move between blocks by usinf a forwarding address for the record
- Can have a *tombstone* indicator in the offset to indicate if a record had been deleted

Pointer Swizzling

- Recall the address for a block can be either in the database address space (slow) or in the memory address space (faster).
- Want to use the latter address if possible.
- The techniques to do this generally go under the name of *pointer swizzling*. A pointer being what the DBMS uses to look up an address
- The rough idea is to have a table that contains the addresses of all blocks in memory together with their database address.

More on Pointer Swizzling



- When we move a block from secondary storage to memory, pointers within the block are allowed to be swizzled. that is a DB address is translated to a memory address.
- A pointer consists of a bit to indicate whether the address is a DB or memory one, followed by the address itself.

Types of Swizzling

- **Automatic Swizzling** -- when read in a block, update the translation table, and swizzle all pointers to this block.
- **Swizzling on Demand** -- when we read in a block, update the translation table, and only swizzle a pointer when try to dereference it.
- **No swizzling** -- when read in a block, update the translation table. If try to dereference a pointer check if in table, if yes, go to where table says, else read in block.
- **Programmer Control of Swizzling** -- sometimes app programmer allowed to force a swizzle.

Returning Blocks to Disk

- When a block is moved back to disk all the pointers that reference into it must be unswizzled.
- The translation table can now be used in reverse to put back into the pointer the database address given the memory address.
- To keep things fast translation table is often a hash table.

Pinned Records and Blocks

- A block is said to be *pinned* if it cannot at the moment be safely returned to disk.
- A bit in the header of the block is often used to say whether the block is currently pinned or not.
- With swizzling might need to pin blocks because a block might have within it a reference to another block. We need to pin blocks which are referred to by either swizzled blocks so these references don't get messed up. What needs to be pinned can be determined using ref counts or a linked list.
- Another reason to pin blocks that will be discussed later is related to the DBMS recover system.

Records with Variable Length Fields

If a record has one or more fields of variable length, then the record must contain enough info to find each of its fields. To do this we can place in its header:

- The length of the record
- Pointers to each of the fields.

Records with Repeating Fields

- Suppose have a record with a lot of DATE's in it.
- Can group multiple occurences together then in the header store a pointer to the first.

Variable Format Records

- Sometimes have records whose format changes from record to record.
- For example, XML data.
- Can store these as a sequence of tagged fields.
- Each field consists of:
  - attribute name
  - the type of the field
  - the length of the field
  - the value of the field

Records that do not fit in Blocks

- Sometimes have records with many fields or which contain things such as video or audio clips which are large. So the record is to big to fit into one block.
- In which case, we can use *spanned records*.
- These kinds of records consist of a sequence of record fragments which do fit into a block.
- Each record or fragment then has a header which says:
  - If its is spanned or not.
  - If it is the first or last fragment in a record
  - If it is a middle fragment where the next and previous fragments are.

Blobs

- A record which has a field that has a value that forces spanning (such as a JPG, MPEG, etc) is called a **blob** (binary large object).
- Blobs often need to be retrieved quickly so want to store in sequential blocks or stripe across different disks.
- Blobs also often need special index structures to be able to do things like retrieve say the 40 minute of a movie.