Main-Memory Databases - Course Notes

Topic 4 - "Data organisation"

Maurice van Keulen

November 21, 2002

1 Data organisation in disk-based RDBMS

Study material: From [LBK02], the sections 11.1, 11.2, 11.3.

These sections are well-readable, so the text below only contains some additional comments.

11.1 Disk organisation I included this section in the study material, because section 11.3 refers to some concepts like "seek time" and "rotational latency". I suggest you only read those bits necessary for understanding 11.3.

11.2 Heap files Figure 11.2 shows how rows of a table are stored in pages in some data file. In a heap file organisation, a page starts with all data belonging to one row, followed by all data belonging to the next row, and so on. Note that the figure is a simplification: when there are attributes of variable length (such as `VARCHAR(255)`), it becomes a more complicated, because then the rows vary in length. Nevertheless, all data belonging to one row is kept together.

Buffer managment in an RDBMS acts as a cache on the data file. Whenever you access a certain row of a table, the RDBMS determines on which page the row is located. It then checks if it is already in the buffer. If not, it retrieves the entire page from disk replacing some other page in the disk buffer.

Disk buffering is actually an operating system functionality. But because the RDBMS better knows in what pattern it will access the data, it is able to do more intelligent buffering. Typically, it will deviate from the standard LRU-mechanism if it is doing, for example, a sequential scan of a table. Some DBMS-products even try to circumvent the OS completely by claiming a part of disk and doing raw disk I/O themselves. In that case, data is not in files, but pages are written directly to certain locations on disk.

Performance in RDBMSs is often measured in terms of I/O accesses, which is equal to the number of page retrievals and writes. The argument for this simplification is that disk access dominates the performance of a query or transaction. Therefore, the number of I/O accesses is a reasonable measure. Because of deletions, it may happen that 'holes' exist on a page, empty space where once a row was stored. The negative effect of holes, is that when an entire table needs to be read from disk, more pages have to be read compared to pages without 'holes'. So, many deletions and insertions may degrade the performance. 'Cleaning up' the table, however, is a very expensive operation, so that is done only periodically during idle hours.

11.3 Sorted files By keeping the rows in a file sorted, one can lookup a row quicker using binary search techniques. In terms of I/O accesses, a lookup will cost approximately $\log_2 F$ page accesses, where F is the number of pages used by the table. As noted in section 11.3, this may nevertheless mean worse performance, because binary search does not access pages sequentially, which would be the optimal access pattern for a disk.

2 Data organisation in Monet

Study material: From [Bon02], the sections 5.0, 5.1, 5.2, and 5.3.2.

These sections are also well-readable, so the text below again only contains some additional comments and explanations.

5.2 Data storage in Monet Figure 5.2 illustrates the data structure of a BAT. Some concepts may not be known to you:

- *map a file into virtual memory.* As you know, a part of a disk can be treated as additional memory through the virtual memory services of the OS. Programs work exclusively with virtual memory addresses. The OS will take care of retrieving pages from disk when they are not in main-memory, and translating virtual addresses to physical addresses. If the operating system allows you to specify which part of the disk should be used for virtual memory, you can do a special trick. By specifying that part of the disk that is actually occupied by the BUN head of a BAT, you can 'move' the heap into memory without actually having to read it. At a subsequent access at the corresponding virtual addresses, the OS will read in the appropriate pages.

- *Pointer swizzling*. This is a term that comes from the area of object-oriented databases. An object can refer to another by means of an object-identifier. Accessing an object means that the object that corresponds to the object-identifier be loaded into memory which among others means the object-identifier has be looked up in a table to find the page on which the object is located on disk. If the object was already loaded, accessing an object means just this lookup to get a pointer to the corresponding object. An object-identifier, hence, acts as a kind of logical pointer which upon dereferencing it is converted into a physical memory pointer (also called hard pointer). It would obviously be faster to use physical pointers instead of object-identifiers. Unfortunately, it is unknown beforehand in which memory location an object is loaded, so storing physical pointers inside objects doesn't work. But as long as an object stays in memory, the memory location stays the same. Pointer swizzling is the technique whereby object-identifiers are converted into pointers for objects that are in memory. This can be done upon first derefencing an object-identifier (to make subsequent accesses to the same object faster) or upon discovery, that is if an object is loaded or subject of some operation the pointers (=object identifiers) inside it are swizzled.

5.3.2 Data structure optimizations The BAT data structure incorporates a number of features that optimize certain operations. By creating or maintaining different descriptors for the same BUN heap, several often-used operations can be performed without having to access the data at all.

3 Example exam-questions

- Suppose we have inserted into a table stored as sorted file many times and now each page has an extra overflow page. How many I/O accesses does a binary search take?

- Suppose I have a BAT `b` with strings in the head and integers in the tail and I'd like to perform the formula $|2(x+1)|$ to all values x in the tail. I can do this in MIL with

  ```
  result:=b.[+](1).[*](2).[abs].
  ```

  The program below is more than twice as fast. Can you explain why based on your knowledge of data organization in Monet?

  ```
  o:=b.mark(0@0);
  p:=b.reverse.mark(0@0).reverse;
  p:=p.[+](1).[*](2).[abs];
  result:=o.join(p);
  ```

- Suppose I have a huge BAT with characters in the tail. Since, only the characters 'a' through 'z' occur, I'm considering if I should use an enumerated type. I'm performing two operations on the BAT: (1) do a $\{$`count`$\}$ (i.e., for each character, count how often it occurs), and (2) replace all a's by b's. What do I gain/loose in (a) storage space, (b) performance on operation 1, and (c) performance on operation 2?