CS 564

The Minirel Page Class

Introduction

Pages are used to hold records in the database and are the unit of transfer between disk and the buffer pool that you implemented in Part 3 of the project. A page holds one or more variable-sized records and is organized using a structure termed a "slotted page" which is described below.

The Page Class

The following is the definition of the Page class (given in page.h).

```
const unsigned PAGESIZE = 1024;
const unsigned DPFIXED= sizeof(slot_t)+4*sizeof(short)+2*sizeof(int);

private:
    char  data[PAGESIZE - DPFIXED];
    slot_t  slot[1]; // first element of slot array - grows backwards!
    short slotCnt; // number of slots in use;
    short freePtr; // offset of first free byte in data[]
    short freeSpace; // number of bytes free in data[]
    short dummy; // for alignment purposes
    int  nextPage; // forwards pointer
    int  curPage;  // page number of current pointer

public:
    void init(const int pageNo); // initialize a new page
    void dumpPage() const;       // dump contents of a page

    // returns value of nextPage
    const Status getNextPage(int& pageNo) const;

    // sets value of nextPage to pageNo
    const Status setNextPage(const int pageNo);

    // inserts a new record (rec) into the page, returns RID of record
    const Status insertRecord(const Record & rec, RID& rid);

    // delete the record with the specified rid
    const Status deleteRecord(const RID & rid);

    // returns RID of first record on page
    const Status firstRecord(RID& firstRid) const;

    // returns RID of next record on the page
    const Status nextRecord (const RID& curRid, RID& nextRid) const;

    // returns reference to record with RID rid
    const Status getRecord(const RID& rid, Record& rec);
```

```
};
```

Description of Private Data Members of Page Class

The Page class uses a slotted page approach for organizing variable length records. When a record is in memory, it is convenient for the higher layers to deal with a record in the following form.

```
struct Record
{
  void* data;
  int length;
};
```

On a page, records are stored differently. The data parts of records are stored one after another starting from byte 0 of a page while the slot array (which contains pointers to the start of records) grows from the end of the page towards the front. There are several important things to notice. First, the data[] area in a page is used to hold both records and all but the 0th element of the slot array! Second, since the slot array grows towards the "front", it is indexed with the values 0, -1, -2, -3, ... and not 0, 1, 2, 3 ... This is very important! Each element of the slot array has the structure:

```
struct slot_t {
        short offset;
        short length;  // equals -1 if slot is not in use
};
```

The offset specifies the start of the record from the beginning of data[] and the length equals the length of the record. The slotCnt attribute is used to indicate the current end of the slot array. It is initialized to 0 (by the init() member function) and everytime a record is added to the page, its value is **decremented** by 1. freePtr is the first free byte in data[]. freeSpace keeps track of how much free space is available in data[]. dummy is there for alignment purposes only. The nextPage members are used to form a linked list of the pages in the file. Finally curPage is the page number of the current page. As records are deleted, the resulting hole should be compacted. However, since records are addressed via a (pageNo, slotNo) pair called RID, we cannot compact the slot array (except in one case). RIDs have the following structure:

```
struct RID{
    int  pageNo;
    int  slotNo;
};
```

The pageNo identifies a physical page number (something that the buffer manager and the DB layers understand) in the file. The slotNo specifies an entry in the slot array on the page. The only tricky thing is that the slotNo field of an RID is always positive while the slot array is itself indexed using a negative number, so the sign of the slotNo must be flipped when converting from one form to another. As an aside, in a real DBMS, RIDS usually have a third component that indicates the storage volume or file.

Description of Member Functions of the Page Class

You perhaps have noticed that the Page class does not have either a constructor or destructor. Why is this? Well recall that in the BufMgr class, the bufPool is defined to be an array of Pages. When the bufPool is first created, the Page class constructor would get called on each Page instance in bufPool. This is not when we want the "constructor" to be called. Instead we want it to be called when first allocate an empty page in the file (usually in the HeapFile::insertRecord method). To get around this problem, the Page class has no constructor. Rather an init() method is provided.

void init(const int pageNo)

Initializes the structure of a page. The pageNo parameter is used to set the curPage attribute.

void dumpPage()

Prints the contents of the data structures of the page (but not the contents of the records on the page). This might be useful for debugging.

const Status getNextPage(int& pageNo)

Returns the value of the nextPage value via pageNo. Note that the pageNo of the next page need not be consecutive to the curPage number. This is because the order of pages in the heap file is determined by the linked list of pages and not by the page numbers. The File class in the I/O layer might come up with any pageNo when allocating a new page. The page numbers are used merely as pointers.

Returns OK if no errors occurred.

const Status setNextPage(const int pageNo)

Sets value of nextPage to pageNo. Returns OK if no errors occurred.

const Status insertRecord(const Record& rec, RID& rid)

Inserts the record rec in the page. The data and length portions of the record are copied into appropriate postions in the page. Returns the RID of the record via the rid parameter. Returns OK if no errors occurred, NOSPACE if there is not sufficient room on the page to hold the record.

const Status deleteRecord(const RID& rid)

Deletes record with RID rid, compacting the hole created. Compacting the hole, in turn, requires that all the offsets (in the slot array) of all records after the hole be adjusted by the size of the hole. The slot array can be compacted only if the record corresponding to the last slot is being deleted.

Returns OK if no errors occurred and INVALIDSLOTNO if the rid is invalid (i.e. there is no such record in the page).

const Status firstRecord(RID& firstRid) const

Returns the RID of the first record of the page (via the firstRid parameter). Remember that slot[0] (and possibly slot[-1], ...) might be empty due to deletions. In particular, you should scan the slot array *backward* starting at 0, looking for a non-empty slot (an empty slot is indicated by a length field of -1).

Returns OK if no errors occurred, NORECORDS if the page was empty.

const Status nextRecord (const RID& curRid, RID& nextRid) const

Returns (via the nextRid parameter) the RID of the next record on the page after curRid. Again, keep in mind that the slot array can contain empty entries and that since the slot array grows backwards, you index it using the values 0, -1, -2, -3 and not 0, 1, 2, 3... Also, don't forget to flip the sign of slotNo in curRid. Remember when to stop searching by looking at the value of slotCnt. slotCnt is kept one less than the last valid slot entry. That is, if a page contains 4 records and no records hav been deleted then slot entries 0, -1, -2, -3 will be in use and slotCnt will be -4.

Returns OK if no errors occurred, ENDOFPAGE if curRid was the last record on the page.

const Status getRecord(const RID& rid, Record & rec)

Returns the length and a pointer to the record with RID rid. You can get recLen from the proper entry in the slot array. However recPtr needs to be an address and not just the offset from the beginning of data[]. This means that the upper layers can directly modify part of the page frame that contains the record. Returns OK if no errors occurred, INVALIDSLOTNO if the rid is invalid (i.e. if there is no such record on the page).