?
§3 The Paged File (PF) Layer §5 The Access Method (AM) Layer

§ 4. The Heap File (HF) Layer

This section describes the interface to the heap file management layer that you are to implement for this part of the project. The PF layer code will be used to implement this layer. The task of the heap file layer is to provide functions for managing unordered files of records.

In writing the code for this layer, it is up to you to decide how exactly to store records on pages of PF layer files. The decision, however, can be simplified somewhat by the fact each heap file will contain a set of records that are all of the *same size*. Thus, an *unpacked with bitmap* design will be an excellent candidate as the page format. You also get to combine page numbers and record positions in order to produce the record identifiers mentioned in the function interfaces below.

§ 4.1. HF Interface Routines

- **void HF_Init(void)**
  This function is to be used to initialize the data structures that you will be maintaining for the HF layer. You may use this as an opportunity to call `PF_Init( )`, and to do any initializations needed by the HF layer (for instance, initializing all file table entries to *not in use*). This function takes no parameters and produces no return value.

- **int HF_CreateFile(char *fileName, int recSize)**

```
char    *fileName;          /* name of the file to be created */
int     recSize;            /* record size in bytes */
```

  This function calls `PF_CreateFile()` and `PF_OpenFile()` to create and open a paged file called `fileName` whose records are of size `recSize`. In addition, it initializes the file, by allocating a page for the header and storing appropriate information in the header page. This paged file must be closed by `PF_CloseFile()`. It returns `HFE_OK` if the new file is successfully created, and an HF error code otherwise.

- **int HF_DestroyFile(char *fileName)**

```
char    *fileName;          /* name of the file to be destroyed */
```

  This function destroys the file whose name is `fileName` by calling `PF_DestroyFile()`. It is included here only for completeness of the HF layer interface. This function returns `HFE_OK` if the file is successfully destroyed, and `HFE_PF` otherwise.

- **int HF_OpenFile(char *fileName)**

```
char    *fileName;          /* name of the file to be opened */
```

  This function opens a file called `fileName` by calling `PF_OpenFile()`. Its return value is the HF file descriptor of the file if the new file has been successfully opened; an HF error code is returned otherwise. In addition to opening the file, this function will make an entry in the HF file table.

- **int HF_CloseFile(int HFfd)**

```
int     HFfd;           /* HF descriptor of the file to be closed */
```

This function closes the file with the file descriptor HFfd by calling PF_CloseFile() with its PF file descriptor. If successful, it deletes the entry of this file from the table of open HF files. If there is an active scan, however, this function fails and no further action is done. It returns HFE_OK if the file has been successfully closed, HFE_SCANOPEN if there is an active scan, an HF error code otherwise.

- RECID HF_InsertRec(int HFfd, char *record)

```
int     HFfd;           /* HF descriptor of the file to be closed */
char    *record;        /* record to be inserted */
```

This function inserts the record pointed to by record into the open file associated with HFfd. This function returns the record id (of type RECID) that is assigned to the newly inserted record if the insertion is successful, and an HF error code otherwise.

- int HF_DeleteRec(int HFfd, RECID recId)

```
int     HFfd;           /* HF file descriptor of an open file */
RECID   recId;          /* id of the record to be deleted */
```

This function deletes the record indicated by recId from the file associated with HFfd. It returns HFE_OK if the deletion is successful, and an HF error code otherwise.

- RECID HF_GetFirstRec(int HFfd, char *record)

```
int     HFfd;           /* HF file descriptor of an open file */
char    *record;        /* pointer to the record buffer */
```

This function retrieves a copy of the first record in the file associated with HFfd. If it succeeds, the record id of the first record in the file is returned, and a copy of the record itself is placed in the location pointed by record. If the file is empty, it returns HFE_EOF, otherwise it returns an HF error code.

- RECID HF_GetNextRec(int HFfd, RECID recId, char *record)

```
int     HFfd;           /* HF file descriptor of an open file */
RECID   recId;          /* record id whose next one will be retrieved */
char    *record;        /* pointer to the record buffer */
```

This function retrieves a copy of the record following the one with id recId in the file associated with HFfd. If it succeeds, the record id of this record is returned, and a copy of the record itself is placed in the location pointed to by record. If there are no more records in the file, it returns HFE_EOF. If the incoming record id is invalid, it returns HFE_INVALIDRECORD, otherwise it returns another HF error code.

- int HF_GetThisRec(int HFfd, RECID recId, char *record)

```
int     HFfd;           /* HF file descriptor of an open file */
RECID   recId;          /* id of the record that will be retrieved */
char    *record;        /* pointer to the record buffer */
```

This function retrieves a copy of the record with recId from the file associated with HFfd. The data is placed in the buffer pointed to by record. It returns HFE_OK if it succeeds, HFE_INVALIDRECORD if the argument recId is invalid, HFE_EOF if the record with recId does not exist, or another HF error code otherwise.

- int HF_OpenFileScan(…)

```
int     HFfd;           /* HF file descriptor */
char    attrType;       /* 'c', 'i', or 'f' */
int     attrLength;     /* 4 for 'i' or 'f', 1-255 for 'c' */
int     attrOffset;     /* offset of attribute for comparison */
int     op;             /* operator for comparison*/
char    *value;         /* value for comparison (or null) */
```

This function opens a *scan* over the file associated with HFfd in order to return its records whose value of the indicated attribute satisfies the specified condition. The attrType field represents an attribute type, which can be a character string of length 255 or less (c), an integer (i), or a float (f). If value is a null pointer, then a scan of the entire file is desired. Otherwise, value will point to the (binary) value that records are to be compared with. The scan descriptor returned is an index into the *scan table* (implemented and maintained by your HF layer code) used for keeping track of information about the state of in-progress file scans. Information such as the record id of the record that was just scanned, what files are open due to the scan, etc., are to be kept in this table. If the scan table is full, an HF error code is returned in place of a scan descriptor.

The parameter op determines the way that the value parameter is compared to the record's indicated attribute value. The different comparison options are encoded in op as follows.

```
1 for EQUAL (i.e., attribute = value)
2 for LESS THAN (i.e., attribute < value)
3 for GREATER THAN (i.e., attribute > value)
4 for LESS THAN or EQUAL (i.e., attribute <= value)
5 for GREATER THAN or EQUAL (i.e., attribute >= value)
6 for NOT EQUAL (i.e., attribute != value)
```

- RECID HF_FindNextRec(int scanDesc, char *record)

```
int     scanDesc;       /* descriptor of the file being scanned */
char    *record;        /* pointer to the record buffer */
```

This function retrieves a copy of the next record in the file being scanned through scanDesc that satisfies the scan predicate. If it succeeds, it returns the record id of this record and places a copy of the record in the location pointed to by record. It returns HFE_EOF if there are no records left to scan in the file, and an HF error code otherwise.

- int HF_CloseFileScan(int scanDesc)

```
int     scanDesc;       /* descriptor of the file being scanned */
```

This function terminates the file scan indicated by scanDesc. It returns HFE_OK if it succeeds, and an HF error code otherwise.

- bool_t HF_ValidRecId(int HFfd, RECID recid)

```
int     HFfd;           /* HF file descriptor of an open file */
RECID   recId;          /* id of the record to be validated */
```

This function returns TRUE if both HFfd and recid are valid, and FALSE otherwise. The notion of recid validity is based on the value ranges of its pageNum and recNum.

The recid is considered valid as long as both the values are in proper ranges for a given HF file. In other words, a valid recid can point to a record slot that may be occupied by an existing record or may be empty.

§ 4.2. Implementation Notes

In this section, we elaborate on a few points that we raised in the preceding function descriptions, occasionally suggesting specific implementation ideas. Let us emphasize, however, that all of our implementation suggestions are just that: *suggestions*. You are free to use alternative ideas if you wish, as long as they do not trivialize the problem but, in fact, improve on some aspect of the system. The function interfaces themselves are the only things that you may not change. Each heap file should be represented as a PF layer file. You should place no limit on the number of records to be stored in a file (other than the limit that is imposed by your RECID structure). Each file should be able to grow arbitrarily long.

§ 4.2.1. File header and Data pages

In addition to the data pages, you will probably want to have a *HF header page*, on which you store information about the file. The file header may include information such as the size of the records in the file, the number of records per page, the number of data pages, and so forth.

It is *strongly discouraged* to do a sequential scan of the file to find a slot to insert a record or to get a record. Your solution should be more sophisticated, so that some *acceptable performance* is achieved.

§ 4.2.2. Page format and Record identifiers

Each data page of a heap file will contain some metadata in the page header as well as a collection of records. Exactly what goes into a page header is for you to decide. You will want to keep track of where the records are in the page, and where there is free space in the page. One of the best ways to do this is with an *unpacked with bitmap* page format. If a page can hold n records, then you will have an n-bit bitmap somewhere on the page which indicates which slots contain valid data and which do not. We assume that no records can be larger than a data page.

A record id uniquely identifies a record within a heap file, and will also serve as the id for the corresponding tuple in upper layers. You can use the following RECID type definition, as declared in minirel.h file.

```
typedef struct {
     int   pagenum;
     int   recnum;
} RECID;
```

Whatever you do, it will have to permit the HF layer code to identify the page where the record is stored and the slot that it occupies on that page.

There is one constraint, though. Record ids *must* be permanent, in the sense that the id of a record cannot change as a result of the insertion or deletion of a different record.

§ 4.2.3. File table and Scan table

The HF_OpenFile() and HF_CloseFile() functions maintain a table of open files called the

HF file table. There is a maximum number of files that can be open at once, which is determined by the HF_FTAB_SIZE constant in hf.h. When HF_OpenFile() is called, the file header data are copied into the file table. By doing so, you can keep such information as the record size and the number of pages in the file more readily accessible in the table instead of having to read in the header page again (or keeping that information on all pages). One tricky thing about the file table is that any changes made to its content while the file is open (for instance, the number of pages in the file may change while the file is open) have to be written back when the file is closed.

Similarly to the *HF file table*, the HF_OpenFileScan() and HF_CloseFileScan() functions maintain a table of scan files called the *HF scan table*. Assume that no more than MAXSCANS (defined in hf.h) scans will ever need to be performed at one time.

§ 4.2.4. Error handling

For the HF layer functions whose return value type is RECID, the RECID returned in case of error can be the one that reflects an error condition (*e.g.*, pageNum set to -1 and recNum set to a specific error code). Alternatively, an error code can be returned by use of a global variable (*e.g.*, HFerrno), which needs to be declared in the HF layer.

§ 4.2.5. Database Consistency

The HF layer takes no responsibility of the database consistency. It should be taken care of by an upper layer, which is not part of the single-user MiniRel project. The HF layer will do whatever it is told to. Therefore, you do not have to check whether the data are being altered during a scan or not. Of course, more than one scan on a file are allowed at any time.