?
§4 The Heap File (HF) Layer §6 The Front End (FE) Layer

§ 5. The Access Method (AM) Layer

This section describes the interface to the access method layer code that you are to implement for the project. The access method facility is based on B^+ -trees that support the procedural interface described below. In the overall MiniRel architecture that you will be implementing, the AM layer will sit on top of the PF layer, meaning that it sits at the same architectural level as the HF layer. You will use a paged file from the PF layer to store a B^+ -tree index, and you will use pages in the file to store individual nodes of the tree. The name of an index file should be a combination of the name of the file on which the index is built and the index number (1st, 2nd, 3rd, etc.), *e.g.*, *fileName.indexNo*. The code that you build for this layer will help support an indexed file abstraction, where a file is organized as an unordered sequence of records and may have any number of *single-attribute indices* associated with it to speed up selection and join queries. All function names begin with the prefix AM, which identifies the AM layer.

§ 5.1. AM Interface Routines

- **void AM_Init(void)**
  This function is used to initialize the data structures that you will be maintaining for the AM layer. Since the AM layer is typically used together with the HF layer, you may use this as an opportunity to invoke `HF_Init()`. This function takes no parameters and produces no return value.

- **int AM_CreateIndex(…)**

```
char    *fileName;        /* name of indexed file           */
int     indexNo;          /* id of this index for the file  */
char    attrType;         /* 'c', 'i', or 'f'               */
int     attrLength;       /* 4 for 'i' or 'f', 1-255 for 'c' */
bool_t  isUnique;         /* uniqueness of key values       */
```

  This function creates an index numbered `indexNo` on the file `fileName`. The `indexNo` parameter will have a unique value for each index created on a file. Thus, it can be used along with `fileName` to generate a unique name for the PF layer file used to implement the index. The type and length of the attribute being indexed are described by the third and fourth parameters. The fifth parameter `isUnique` must be set to FALSE (or can be ignored) because we do *not enforce primary key integrity*. The job of this function is to create an empty index by creating a PF layer file and initializing it appropriately (*i.e.*, to represent the empty tree). You may assume that all of the indices for a given HF layer file will be created before any records are inserted into the file. So there is *no need to implement a bulk loading operation*. It returns `AME_OK` if it succeeds, and an AM error code otherwise.

- **int AM_DestroyIndex(char *fileName, int indexNo)**

```
char    *fileName;          /* file name of the base table */
```

```
int     indexNo;            /* index number */
```

This function destroys the index numbered indexNo of the file fileName by deleting the file that is used to represent it. It returns AME_OK if it succeeds, and an AM error code otherwise.

- int AM_OpenIndex(char *fileName, int indexNo)

```
char    *fileName;          /* file name of the base table */
int     indexNo;            /* index number */
```

This function opens an index numbered indexNo of the file fileName by calling PF_OpenFile(). If the operation is successful, the return value is a file descriptor of the B^+-tree index, which serves as an index into the index table. An AM error code is returned otherwise.

- int AM_CloseIndex(int AM_fd)

```
int     AM_fd;              /* file descriptor */
```

This function closes the index file with the indicated file descriptor by calling PF_CloseFile(). It deletes the entry for this index from the index table. It returns AME_OK if the file is successfully closed, and an AM error code otherwise.

- int AM_InsertEntry(int AM_fd, char *value, RECID recId)

```
int     AM_fd;              /* file descriptor                */
char    *value;             /* attribute value for the insert */
RECID   recId;              /* id of record to insert         */
```

This function inserts a (value, recId) pair into the index represented by the open file associated with AM_fd. The value parameter points to the value to be inserted into the index, and the recId parameter identifies a record with that value to be added to the index. It returns AME_OK if it succeeds, and an AM error code otherwise.

- int AM_DeleteEntry(int AM_fd, char *value, RECID recId)

```
int     AM_fd;              /* file descriptor                */
char    *value;             /* attribute value for the delete */
RECID   recId;              /* id of record to delete         */
```

This function removes a (value, recId) pair from the index represented by the open file associated with AM_fd. It returns AME_OK if it succeeds, and an AM error code otherwise.

- int AM_OpenIndexScan(int AM_fd, int op, char *value)

```
int     AM_fd,              /* file descriptor                */
int     op,                 /* operator for comparison        */
char    *value              /* value for comparison (or null) */
```

This function opens an index scan over the index represented by the file associated with AM_fd. The value parameter will point to a (binary) value that the indexed attribute values are to be compared with. The scan will return the

record ids of those records whose indexed attribute value matches the value parameter in the desired way. If value is a null pointer, then a scan of the entire index is desired. The scan descriptor returned is an index into the *index scan table* (similar to the one used to implement file scans in the HF layer). If the index scan table is full, an AM error code is returned in place of a scan descriptor.

The op parameter can assume the following values (as defined in the minirel.h file provided).

```
#define EQ_OP           1
#define LT_OP           2
#define GT_OP           3
#define LE_OP           4
#define GE_OP           5
#define NE_OP           6
```

- RECID AM_FindNextEntry(int scanDesc)

```
int     scanDesc;           /* scan descriptor of an index */
```

This function returns the record id of the next record that satisfies the conditions specified for an index scan associated with scanDesc. If there are no more records satisfying the scan predicate, then an invalid RECID is returned and the global variable AMerrno is set to AME_EOF. Other types of errors are returned in the same way.

- int AM_CloseIndexScan(int scanDesc)

```
int     scanDesc;           /* scan descriptor of an index */
```

This function terminates an index scan and disposes of the scan state information. It returns AME_OK if the scan is successfully closed, and an AM error code otherwise.

- void AM_PrintError(char *errString)

```
char    *errString;         /* pointer to an error message */
```

This function writes the string errString onto *stderr*, and then writes the last error message produced by the AM layer onto *stderr* as well. This function will make use of a global integer value, AMerrno, in order to keep track of the most recent error; this error code should be appropriately set in each of the other AM layer functions. This function has no return value of its own.

§ 5.2. Implementation Notes

We elaborate on a few points that we raised in the preceding function descriptions, occasionally suggesting specific implementation ideas. Again, you should view these as suggestions only, the function interfaces being the only things that you may not change.

§ 5.2.1. B^+-trees Node Splitting and Merging

Your indices must be based on B^+-trees stored in a corresponding PF layer file

(where each tree node is stored in a page in the file). You should implement the full search and insertion capabilities provided by B^+-trees, including *node splitting* at any level of the tree when nodes overflow during insertions. You do not have to implement the full deletion algorithm, though. You may rather implement deletion by simply removing an appropriate index entry from the leaf in which it resides, skipping any *node merging* that you would otherwise need to do.

§ 5.2.2. Duplicate keys in B^+-trees

If a B^+-tree index is built on a non-key attributes, there may be more than one entry with the same key value. All such entries with the same key value should be on the same leaf node and not split across two nodes, even if this means that some leaves may be less than 50% full. You do not have to worry about handling the case where there are so many duplicate key values that you can overflow an entire node with record ids related to a single key value. (However, you should be aware that real systems *do* have to handle this case, and you might want to think about how your code would have to be extended if you were going to deal with this problem.)

§ 5.2.3. Index table and Scan table

The AM_OpenIndex() and AM_CloseIndex() functions maintain a table of open indexes called the *AM index table*. There is a maximum number of indexes that can be open at any time, which is determined by the AM_ITAB_SIZE constant in am.h. For each of the open indexes, an entry in the index table can store such information as the name and descriptor of the corresponding PF file, the header information of the index and a pointer to the root node cached in the buffer pool and so forth.

Similarly to the *AM index table*, the AM_OpenIndexScan() and AM_CloseIndexScan() functions maintain a table of scan indexes called the *AM scan table*. Assume that no more than MAXISCANS (defined in am.h) scans will ever need to be performed at one time.

§ 5.2.4. AM Layer Initialization

The AM layer makes uses of PF layer functions, so the PF layer needs to be initialized before the AM layer is. Since the HF and AM layers are in principle independent, the HF layer need not be initialized before the AM layer. However, the AM layer will not be used without the HF layer (because no index will be created for a non-existing relation) in any realistic situation. Thus, it is not a bad idea to have the HF layer initialized as part of the AM layer initialization.

In the tests we will provide, we will assume that the PF layer is initialized by the HF layer (HF_Init()), which is in turn initialized by the AM layer (AM_Init()).

§ 5.2.5. Database Consistency

The AM 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 AM layer will do whatever it is told to. Therefore, you do not have to check whether the index entries are being altered during a scan or not. For example, you should be able to handle entry deletions during an index scan. Of course, more than one scan on an index are allowed at any time.

§4 The Heap File (HF) Layer May 4, 2017 §6 The Front End (FE) Layer