MINIREL PART II: The Access Methods Layer
Due Date: Thursday, 2/17/05, 1PM

1 THE ACCESS METHOD (AM) LAYER

For Part II, you will implement an access method facility based on exogenous $B^+$ trees that supports the procedural interface described below. In the overall DBMS architecture that you will be implementing, the AM layer will sit on top of the PF layer, on the same level as the HF layer. You are to choose how the index files will be named. File names 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 to support an indexed file abstraction, where files are organized as heaps and may have any number of single-attribute secondary indices associated with them to speed up selection and equality-join queries. Note that all indices are secondary since we are using a heap file as the base file rather than a sorted file. All routine names begin with the prefix AM, which identifies the associated layer.

1.1 Access Method Layer Interface Routines

(1) $\text{errVal} = \text{AM_Init}()$

This routine does any initialization you want to do for your AM layer.

(2) $\text{errVal} = \text{AM_CreateIndex(fileName, indexNo, attrType, attrLength)}$

\begin{verbatim}
char *fileName; /* name of indexed file */
int indexNo;   /* number of this index for file */
int attrType;  /* INT_TYPE, REAL_TYPE, or STRING_TYPE in minirel.h */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
\end{verbatim}

This routine creates a secondary 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. This routine will only be called when the corresponding heap file is empty. This routine creates an empty index by creating PF layer files and initializing them appropriately.

(3) $\text{errVal} = \text{AM_BuildIndex(fileName, indexNo, attrType, attrLength, offset)}$
char *fileName;  /* name of indexed file */
int indexNo;    /* number of this index for file */
int attrType;  /* INT_TYPE, REAL_TYPE, or STRING_TYPE in minirel.h */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
int offset;    /* offset within base record of indexed attribute */

This routine creates a secondary index numbered indexNo on the file fileName. The type and length of the attribute being indexed are described by the third and fourth parameters. The parameter offset is the offset from the beginning of the record of the indexed attribute. This routine will only be called when the corresponding heap file is not empty. First an empty index is created and then the entire heap file is scanned while each record in the file is entered into the index. Note, I suggest you use AMInsertEntry (below) to add the records. The function returns AME_OK if it succeeds, and an AM error code otherwise.

(4) errVal = AM_DestroyIndex(fileName, indexNo)

char *fileName;  /* name of indexed file */
int indexNo;    /* number of this index for file */

This routine 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.

(5) errVal = AM_OpenIndex(fd,fileName, indexNo)

int *fd ;        /* used to return file descriptor of the opened index */
char *fileName;  /* name of indexed file */
int indexNo;    /* number of this index for file */

This routine opens the index identified by the pair (fileName, indexNo). The file descriptor is returned via the fd parameter. It returns AME_OK if it succeeds, and an AM error code otherwise.

(6) errVal = AM_CloseIndex(fd)

int fd ;        /* file descriptor of the index to be closed */

This routine closes the index identified by fd. It returns AME_OK if it succeeds, and an AM error code otherwise.

(7) errVal = AM_InsertEntry(fd, attrType, attrLength, value, recId)

int fd ;        /* file descriptor */
int attrType; /* INT_TYPE, REAL_TYPE, or STRING_TYPE in minirel.h */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
char *value; /* attribute value for the insert */
RECID recId; /* id of record to insert */

This routine inserts a (value, recId) pair into the index specified by fd. Note, fd was obtained
from AM_OpenIndex. 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. The recId
is obtained from the insertion of the record into the heap file. It returns AME_OK if it succeeds,
and an AM error code otherwise. An example of use of this function would be:

errVal = HF_InsertRec(fileDesc, &recId, record, recordSize);
errVal = AM_OpenIndex(&fd, fileName, indexNo);
errVal = AM_InsertEntry(fd, attrType, attrLength, value, recId);
errVal = AM_CloseIndex(fd);

where all the parameters of AM_InsertEntry are obtained from the structure and contents of
the record being inserted.

(8) errVal = AM_DeleteEntry(fd, attrType, attrLength, value, recId)

int fd; /* file desriptor of index */
int attrType; /* INT_TYPE, REAL_TYPE, or STRING_TYPE in minirel.h */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
char *value; /* attribute value for the delete */
RECID *recId; /* id of the record to delete */

This routine removes a (value, recId) pair from the index identified by fd. It returns AME_OK if
it succeeds, and an AM error code otherwise. Once again this function will be used in conjunc-
tion with the HF layer counterpart, HF_DeleteRec(), AM_OpenIndex(), and AM_CloseIndex()..

(9) errVal = AM_OpenIndexScan(scanDesc, fileName, indexNo, attrType, attr-
Length, op, value)

int *scanDesc; /* scan descriptor */
char *fileName; /* name of indexed file */
int indexNo; /* number of this index for file */
int attrType; /* INT_TYPE, REAL_TYPE, or STRING_TYPE in minirel.h */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
int op; /* operator for comparison*/
char *value; /* value for comparison */

3
This routine opens an index scan over the index represented by the file associated with fileName and indexNo. The scanDesc is returned by passing a pointer to an integer. The (non-negative) scan descriptor returned is an index into an in-progress index scan table (similar to the one used to implement file scans in the HF layer). The scan will return the record ids of those records whose indexed attribute value compares in the desired way with the value parameter. The desired comparison will be specified as it is for the HF_OpenFileScan( ) routine. You may also assume that no more than MAXSCANS = 20 scans will ever need to be performed at one time. If the index scan table is full, an AM error code is returned in place of a scan descriptor.

(10) errVal = AM_FindNextEntry(scanDesc, recId)

```c
    int     scanDesc;    /* index scan descriptor */
    RECID  *recId;       /* ptr to a recId */
```

This routine returns the record id of the next record that satisfies the conditions specified for index scan associated with scanDesc. It returns AM_EEOF, if there are no more records that satisfy the scan predicate, and an AM error code otherwise.

(11) errVal = AM_CloseIndexScan(scanDesc)

```c
    int     scanDesc;    /* index scan descriptor */
```

This routine terminates an index scan, closing the associated index file, and disposing of the scan state information.

(12) errVal = AM_PrintIndex(id)

```c
    int     id;          /* index descriptor */
    % int    keyType;    /* INT_TYPE, REAL_TYPE, or STRING_TYPE in minirel.h */
```

This routine prints out the key of all index entries.

1.2 Some Implementation Comments

Your secondary indices must be based on $B^+$ trees, with a given index being represented by a corresponding PF layer file (where each node is represented by a page in the file).

You should implement the full search, insertion, and deletion capabilities provided by $B^+$ trees. Do not start on the deletions until you are sure insertions work correctly. You will get the majority of the points without doing deletions. Furthermore, for deletions implement delete-on-empty do not bother merging nodes (this will make deletions during a scan much easier).
You should also be able to handle entry deletions during an index scan. That is, the following code segment should work:

```c
int scanDesc; /* scan descriptor */
RECID recId;     /* record id */

errVal = AM_OpenIndexScan(&scanDesc, ...);
while ((errVal = AM_FindNextEntry(scanDesc, &recId)) >= 0)
{
    ...

    /* delete the record */
    errVal = AM_DeleteEntry(...,recId,...);

}
```

Making this code work is not too difficult. Moreover, it will make it possible for you to implement deletions based on indexed attributes in Part III.

Each entry in the leaf level node of your index files will be of the form `<attrValue, recId>` or `<attrValue, recIdList>`. This signifies that in the indexed file there exists a record with an id of `recId`, or a set of records with the ids in `recIdList`, whose value for the indexed attribute is `attrValue`. (It is up to you to choose the representation that you prefer.) You do not have to worry about handling the case where there are so many duplicate key values that you can overflow an entire page 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.)

Finally, unlike the preceding phase, I want you to come up with additional tests of your code. In addition to running `testPhase2.c`, you should create a set of experiments that compare the performance with just a Heap File versus an exogenous $B^+$ tree. Provide a description of your tests and experimental results timing information that show the speedup using your $B^+$ tree code.