UNIVERSITY OF WISCONSIN–MADISON
COMPUTER SCIENCES DEPARTMENT

CS 564: DATABASE MANAGEMENT SYSTEMS

Project 3: Access Methods and Joins
Due: Thursday, December 1, 1994, 6 p.m.

Instructor: Raghu Ramakrishnan

1 Introduction

The *relation* is a central abstraction in a Relational DBMS. In Minibase, a relation is represented as a collection of *access methods* (AM's). A relation has exactly one *clustered* AM, which corresponds to how the tuples are stored physically. This can be a SequentialAM (i.e., a HeapFile) or a BTreeAM (implemented as a BTreeFile). In addition, there can be zero or more *un-clustered* AMs (also called *secondary indexes*). To retrieve tuples from the relation, the database system will use one of the associated AMs. Each AM provides a scan for this purpose.

In this assignment, you will implement a simple class called BTreeAM, to support access using a BTreeFile associated with a relation. We will provide you with SequentialAM, which supports sequential access to records using a HeapFile, to serve as an example. (You should be able to implement BTreeAM by making relatively small modifications to SequentialAM.)

An interesting point is that a DBMS regards the result of joining two relation as just another relation. This is reflected in the fact that a join is viewed as just another *access method* (for a relation that is the result of the join). You will also implement a class called SimpleJoinAM, which is derived from SequentialAM.

You will carry out this assignment in teams of two, continuing with the same partner as for the previous assignment, unless you wish to make a change. It will account for 8% of your final grade.

1.1 Available Documentation

You should get familiar with the classes Relation, AccessMethod, RecordScan, SequentialAM and SequentialScan. Read the corresponding header files: relation.h and sequential_am.h in /u/course/cs564-1/minibase/include. The source code for SequentialAM is available in /u/course/cs564-1/minibase/src/rel/sequential_am.C. A template for the classes you need to write is in /u/course/cs564-1/assigns/project3/btree_am.h The following man-pages (in /u/course/cs564-1/minibase/cat) are also helpful: Relation.3, AccessMethod.3, RecordScan.3, SequentialAM.3, SequentialScan.3, BTreeAM.3, BTreeScan.3, SimpleJoinAM.3, SimpleJoinScan.3.

The code that you write and the tests that you will run require familiarity with this material.

2 Getting Started

Copy all the files from */u/course/cs564-1/assigns/project3* to your own local project3 directory and study them carefully. The files are:

- btree_am.h (header file for BTreeAM and BTreeScan)
- btree_am.C (empty file. Put your codes for BTreeAM and BTreeScan here)

- joinam.h (header file for SimpleJoinAM and SimpleJoinScan)
- joinam.C (empty file. Put your codes for the above here)
- Drivers and makefile to test your code.

2.1 BTreeAM

You will implement a class called *BTreeAM*. An instance of this class is associated with a relation, and serves as either a *clustered* or an *un-clustered* access method. BTreeAM is derived from a virtual base class, AccessMethod, and AccessMethod::file() returns the underlying RecordFile of the current AM, a BTreeFile in this case.

A *clustered* AM is quite similar to a RecordFile. Consider the call BTreeAM::insert(*rec*), for example. It simply calls file()→insert(*key, data*), where *key* is a record that stores the (search) key field of *rec* and *data* is the record *rec*. The current implementation of the AccessMethod class assumes a single key field. This means key will always be a record with a single field, even though your BTree implementation is more general. The key field can be found by using the AccessMethod::keyfield() method, while the RecordSpec of the key can be found by using AccessMethod::keyspec().

Now consider an *un-clustered* AM. The call BTreeAM::insert(*rec*) is handled differently since an un-clustered AM stores only the key and the record ID into the underlying RecordFile. In this case, the pair $\langle key, recid \rangle$ of *rec* is inserted into the underlying BTreeFile by using file()→insert(*key, recid*).

You will also implement class called BTreeScan. The BTreeScan class behaves almost exactly like the BTScan class that you implemented in the previous assignment. The two most important methods are BTreeScan::first_record(*rec*) and BTreeScan::next_record(*rec*). If the AM is clustered, they are essentially calls to BTScan::first_record and BTScan::next_record, which retrieve records of the form $\langle key, data \rangle$ from the underlying BTreeFile. The only additional detail is that the BTreeScan methods discard the *key* component, and return the *data* component as *rec*. As with all other scans you have implemented, the returned record should simply point to the buffer manager's buffer.

If the AM is un-clustered, BTScan::first_record and BTScan::next_record retrieve records of the form $\langle key, data \rangle$ from the underlying BTreeFile, where *data* is the recid of the record we want. (It could also be a list of recid's, but we will only consider the case where it is a single recid.) To retrieve this record, we must go to the RecordFile for the clustered AM associated with the relation. Given a un-clustered AM, you can find the clustered RecordFile by using relation()→clustered_AM()→file(). (The call relation() will return a pointer to the relation that owns this un-clustered AM, then clusterd_AM() will get you the clustered AM, then file() will get you the RecordFile.) You may then do a scan of this RecordFile to return the required record (the one whose id is *data*).

3 SimpleJoinAM

As we noted earlier, a join is viewed as just another *access method* for a relation that is the result of the join. You will also implement a class called SimpleJoinAM, which is derived from SequentialAM, and an associated class SimpleJoinScan.

The arguments to the SimpleJoinAM constructor include two AMs, say $am1$ and $am2$, two field numbers $field1$ and $field2$, a Relation object result_relation, and a boolean variable 'realize'. When the user calls SimpleJoinScan, the result of the join is computed by performing a nested loop join between (the sets of tuples retrieved using) $am1$ and $am2$. Since SimpleJoinAM is an access method, it must be associated with some relation, in this case the result of the join; this is the Relation object. The 'realize' flag indicates whether the result relation is to be materialized by writing the tuples into a HeapFile. If this flag is *true*, the SimpleJoinScan::first_record and SimpleJoinScan::next_record methods call result_relation→insert_record to insert the output tuples. If the scan is not materialized, the returned record should point to some local buffer in the SimpleJoinScan object. Only one buffer should be kept. This means the next call to first_record or next_record will overwrite this buffer. If the scan is materialized, the returned record points into the buffer manager as usual.