Relational Database Management
Implementation Project

Princeton University
Computer Science Department

William O’Connell

In this project, a rudimentary relational database management system will be implemented. The first part includes routines to manage files and records. In the second part, these routines will be used by others that manage relations and system catalogs. In the third part, general database utilities are developed. In the fourth part, relational algebra routines will process queries. Finally, buffer management routines will be developed, along with appropriate changes to record and file routines.

For the time being, it is sufficient to think of a relation as a table with rows and columns. The correct name in the relational model for a row is a "tuple", and for a column the name is "attribute". Each table is stored in its own file, with one record per row. Records contain only data values, no column names, etc. Information about a user’s tables (also called *meta-data*: data about data) is stored in two catalogs, the relation catalog and the attribute catalog. The catalogs are also stored as relations, i.e., as tables, and can therefore be accessed using the normal file and record routines. Since catalog relations contain information not only about user relations but also about themselves, it is easy to get confused. In fact, this is probably the most difficult concept in the entire project to understand.

Many routines will be provided in order to speed up development of the system. However, students are strongly encouraged to read the provided code to understand its functionality.

1. Files and Records

In this project, data files reside on disk. Each file corresponds directly to a UNIX file. Properties of files, e.g., the number of records, are stored in a *relation descriptor*. In order to access a file, it must be *opened*. While a file is open, the file descriptor is kept in the *open file table*, which is basically an array of relation descriptors. The open file table is indexed by a file number, i.e., the *file_no* parameter, which is the *file descriptor* returned by the UNIX *open* call. Each slot in the open file table contains information about one opened file.

Files with uniform record length will be used in this project. Thus, record slots can be addressed by record number within each file, and it is easy to calculate a record slot’s location in a file using the record number and the record’s length. The first record slot in a file has the number 0.

A record slot consists of a header and a field for data. A record header contains a flag that indicates whether the slot contains a valid record, and a record number used for free space management. The valid flag is updated only by the routines that append a record or delete a record. All records have record numbers, whether they are valid or invalid.

Unused record slots in a file are chained into a (singly) linked list using this field. The anchor of this list, i.e., the slot number of the first free record, is kept in the *open file descriptor*. *NIL* (-1) is used to mark the end of the list. Notice that this linked list uses record numbers, not main memory addresses or byte offsets in the file.

In the first part of the project, only the fields *rel_name* (both relation name and UNIX file name), *cardinality* (number of valid records), *width* (valid bytes in data field), *file_size* (record number right behind end-of-file, i.e., number of valid plus invalid record slots), *free_slot* (first free record slot, i.e., the anchor of the linked list of invalid record slots) in *REL_DESC* and the fields *open_count* and *rel_desc* in *OPEN_FILE_DESC* are used.

For the *record* module, the following routines are necessary. Fragments for these routines can be found in the file *record.c*. The first two routines are the only ones to use the *lseek*, *read*, *write* UNIX system calls. (Later on even these record routines will call the buffer routines, which in turn use the system calls.)

(1) *get_record (file_no, rec_no, record)* reads a record (including its header) from an open file into the space provided in memory. The record header is not modified.

(2) *put_record (file_no, rec_no, record)* writes a record (including its header) to an open file. The record header is not modified.

(3) *append_record (file_no, record, rec_no)* inserts a new record (including its header) into an open file. If an empty record alot is available, it is used. Otherwise, the record is appended to the end of the file. This routine maintains the cardinality, *file_size*, and *free_slot* appropriately.

(4) *delete_record (file_no, rec_no)* deletes a valid record by marking it invalid and inserting it into the free list.

(5) *get_next_record (file_no, last_rec_no, record, this_rec_no)* provides a scanning mechanism by finding and reading the first valid (not deleted) record behind *last_rec_no*. *Get_next_record* passes back a record and its record number when it returns *OKAY*. At the end of the file, it returns *ERR_EOF*.

(6) *find_record (file_no, last_rec_no, record, this_rec_no, cond_fct, cond_arg)* provides a selective scanning mechanism by finding and reading the first valid (not deleted) record behind *last_rec_no* for which the condition function returns *TRUE*. The selection predicate is tested by the condition function *cond_fct* which returns *TRUE* (record qualifies) or *FALSE*. *Find_record* calls the condition function with the address of a record's data field as first and the condition argument *cond_arg* as second parameter. The condition function pointer may be *NULL*; in this case *find_record* should pretend that the condition is always satisfied. *Find_record* passes back a record and its record number when it returns *OKAY*; otherwise, the value of **this_rec_no* and of **record* are undefined and may be any value. At the end of the file, it returns *ERR_EOF*.

For the *file* module, the following routines are necessary. They use the *open*, *close*, and *unlink* UNIX system calls. Fragments for these routines can be found in the file *file.c*. Filling the fields in the open file table is delegated to the routines that call the file routines, e.g., in the *relation* module described below.

Checking whether a file is already open requires checking the *rel_name* entry of all open files. Open files can be determined by checking the *open_count* entry of all open file table entries between *RES_FILE_NO* and *MAX_FILE_NO* (see *defs.h*); an open count of 0 indicates that no file is open with the respective file number.

(1) *find_open_file (name, file_no)* determines whether or not a file is currently open, and returns its open file number if it is. If the implementations of *get_record()* and *put_record()* use the field *last_seek_pos* (they do not have to) in the open file table to avoid redundant invocations of *lseek()*, this field must be set to zero when a UNIX file is opened.

(2) *create_file (name, file_no)* creates and opens a new file, passing back an open file number. The open file count is set to 1.

(3) *open_file (name, file_no)* opens an existing closed file, passing back an open file number. If the file is already open, it passes back the old file number, keeping a count how often a file has been opened. When a file is opened, its open file descriptor must be initialized.

(4) *close_file (file_no)* closes an open file if the decreased open count is zero.

(5) *destroy_file (name)* destroys a closed file.

2. Relations and Catalogs

On top of the file and record routines, a relational storage system including catalog management is developed in the second part of the project. There are two catalogs, the *relation catalog* and the *attribute catalog*. Each of them is a datafile that contains relation or attribute descriptors, respectively, instead of user relations. The catalogs include descriptors for themselves, i.e., the catalogs of an empty database contains entries about the catalog relations. Since the catalogs are used rather frequently, they are always kept open while the database is open. Their open file numbers are kept in the global variables *rel_cat_file_no* and *attr_cat_file_no*.

For each relation, one relation descriptor is kept. Its *arity* attribute indicates the number of attribute descriptors. An attribute descriptor includes the relation name and the attribute's name, type (*T_INT*, *T_REAL*, or *T_CHAR*, defined in *defs.h*), position (attribute number, 0 to *arity-1*), offset (byte position in the relation's data records), and length (field length in bytes). While a relation is open, its descriptors and their record numbers in the respective catalogs are kept in the open file table.

Two modules will be developed in this part of the project, called *schema* and *relation*. Fragments for the modules are in files with the same names. The module *database* also belongs to this part but is already provided. It is recommended that you look at this module to understand how the database management system is booted.

The *schema* routines assist the *relation* and *database* modules (and other modules to be developed later) by copying relation and attribute descriptors from the open file table to the catalogs and back using the record routines described above. Since the record routines require record numbers, such numbers of relation and attribute descriptors in the catalog files are also kept in the open file table. The *relation* routine only manipulate the open file table and never touch the catalogs (other than by calling *schema* routines).

(1) *save_schema (file_no, rel_desc, attr_desc)* copies in-memory schema descriptors given in the arguments into the open file table. (Hint: only in-memory copies are required.)

(2) *append_schema (file_no)* is used to append the descriptors of a newly created open relation from the open file table to the catalog files, saving the new record numbers in the open file table. (Hint: use the provided routines *alloc_record()* and *dealloc_record()*, as well as *append_record()*.)

(3) *find_schema (name, file_no)* is used to load descriptors from the catalogs into the open file table, saving the new record numbers in the open file table. This one is a bit tricky, so think before you code. It calls the *find_record* routine once on the relation catalog and once for each attribute on the attribute catalog. (Hint: use the provided routines *alloc_schema(), alloc_record(), and dealloc_record().*

(4) *put_schema (file_no)* overwrites the descriptors in the catalog files with the descriptors in the open file table using the saved record numbers. This routine uses *put_record* after ensuring true "valid" flags in the record headers.

(5) *destroy_schema (file_no)* is used to erase the descriptors of an open file from the catalogs. This routine uses *delete_record*.

The following routines are needed in *relation.c*:

(1) *open_relation (name, file_no)* opens an existing relation's file. If *open_count* is 1, descriptors are read from the catalogs into the open file table. (Hint: if one of the catalogs is opened, flush the schemas of all open files to disk after opening the catalog.)

(2) *close_relation (file_no)* closes a relation's file after saving the descriptors.

(3) *create_relation (argc, argv)* creates a relation by creating a file and appending descriptors to the catalogs. The relation is closed on return. The number of arguments are given in *argc*, and *argv* points to strings describing the relation name (*argv [1]*), the attribute names (*argv [2i+2]*) and the attribute type and length (*argv [2i+3]*). Make sure that the attribute offsets as well as the record length come out to be multiples of *WORD_SIZE*, given in *defs.h*. For example, for the binary relation *my_table* with integer attribute *left* and character attribute *right*, *create_relation* is called like
create_relation (6, ["create relation", "my_table", "left", "i", "right", "c50"]). (Hint: create_relation simply calls *create_relation1*, which does all the work. Another hint: use the routine *typed_def* (in type.c) when building attribute descriptors for new relations.)

(4) *destroy_relation (argc, argv)* destroys the named relation, including its catalog entries. The relation name is given in *argv [1]*. This function does not allow to destroy a catalog relation.

Finally, routines to manage an entire database are needed. Each database is kept in a separate UNIX directory, the name of the currently open database is kept in the global variable *db_path*. You may need to modify *file.c* to translate relation names into UNIX paths.

(1) *create_db (argc, argv)* creates and opens a new database, i.e., new catalogs which contain entries describing the catalogs.

(2) *destroy_db (argc, argv)* destroys the closed database, both the data relations and the catalog relations.

(3) *open_db (argc, argv)* opens the existing database by opening the catalogs.

(4) *close_db (argc, argv)* closes all open relations, including the catalogs.

3. Utilities

In the third part, some basic database utilities are developed. Fragments for the routines are given in the files *append.c*, *print.c*, *help.c*, *load.c*, *dump.c*, *delete.c*, and *update.cP (one file per routine listed below).*

(1) *append_tuple (argc, argv)* appends a tuple to the relation named in *argv [1]*. The attributes named in *argv [3i+2]* are to receive the values given as text in *argv [3i+3]*. Each attribute must receive exactly one value. Numbers are given in ASCII, not binary (use *atoi()* and *atof()* for conversion), and input strings do not contain spaces. You may assume that type checking of attributes and values has been done beforehand; however, attributes may be given in an arbitrary order.

(2) *print (argc, argv)* prints the table (relation) named in *argv [1]* "prettily" on the screen, i.e., with proper column alignment and column headers, using the *type_min_print_width* and *typed_print* routines. Each row (tuple) should occupy one output line.

(3) *help (argc, argv)* can be called with or without an argument. If called without an argument (*argc* = 1), it prints the relation catalog. If called with a relation name as argument (*argc* = 2), it prints the catalog entries pertaining to the relation, both relation catalog and attribute catalog entries.

(4) *load (argc, argv)* loads the file named in *argv [1]* into the relation described in *argv [2], ..., argv [argc-1]* after creating this relation. The description for the new relation is equal to that used for *create_relation*. Numbers are given is ASCII, not binary, and input strings do not contain spaces, and fields in the input are separated by one or more spaces and/or newlines. *Load* uses the *typed_scan*. If *argc* is 3 and *argv[0]* is "add," a file is loaded (added) into an existing relation. It is assumed that the input file conforms to this relation’s schema, i.e., the right number of columns are present in the file.

(5) *dump (argc, argv)* saves the relation named in *argv [1]* in a new file named in *argv [2]*. *Dump* uses *typed_print* to format a data value. Datafiles written by *dump* must be readable by *load*.

(6) *delete (argc, argv)* deletes selected tuples from the relation named in *argv [2]*. If *argv [1]* contains a string, a new relation with this name is created and the deleted tuples are moved into the new relation. The selection predicate is given as for *select*, described below in the query processing section.

(7) *update (argc, argv)* updates selected tuples in a relation named in *argv [2]*. It scans through the file, selects records by a predicate similar to a selection predicate, and then updates selected records. There are three kinds of updates: assignments (indicated by "="), increments ("+=") and decrements ("-="). The latter two are only permitted for arithmetic attrbute types, i.e., not for character strings. The order of selection predicates and updates and their organization in *argv* is described in the *update()* function in *update.c*.

4. Query Processing

The processing routines allow evaluation of complex queries composed select, project, sort, and join operations. Each of these routines produces a new relation. The provided interpreter takes care of deleting temporary relations created while evaluating complex queries by calling *destroy_relation*.

For the query processing routines, it is recommended to transform the arguments (predicates, attribute lists) from the argument vector into a data structure that allows faster interpretation. Instead of inspecting the input relation’s attribute descriptors to find type, offset, and length of an attribute for each tuple, this information should be collected once and saved such that it can be easily accessed repeatedly.

(1) *select (argc, argv)* creates a new relation named in *argv [1]* and copies selected tuples from the relation named in *argv [2]* into the new relation. The selection predicate is the conjunction of the clauses given; each clause consists of an attribute name in *argv [3i+3]*, a comparison operator in *argv [3i+4]*, and a constant in *argv [3i+5]*. The selection predicate may be empty.

(2) *project (argc, argv)* creates a new relation named in *argv [1]* and copies selected attributes from all tuples from the relation named in *argv [2]* into the new relation. The projection list is given in *argv [3]*, *argv [4]*, etc.

(3) *join (argc, argv)* creates a new relation named in *argv [1]* by joining the relations named in *argv [2]* and *argv [3]* on the attributes named in *argv [4]* and *argv [6]*. *argv [5]* is always "==". All attributes of both relations are preserved except the one named in *argv [6]*. A number of join strategies are possible, namely nested loops join, merge join, and hash join.

Since the same algorithms used for join can be used for other binary matching operations, our join module implements (left) semi-join, (left) outer join, (left) anti-semi-join as well as intersection, union, and (left) difference. Set operations require union-compatible inputs, i.e., same arity and attribute types, lengths and offsets. Equal names are not required. To organize the algorithms such that the same algorithms can perform all these operations, consider (a) left input tuples for which a matching right input tuple was found, (b) left input tuples without a match, (c) whether or not all right input tuples belong in the output, and (d) whether or not (left) input tuples must be extended with values from right input tuples. Do not write separate code for each binary matching operation.

The nested loops in this project is the most naive version: the inner relation is scanned for each outer tuple. While this algorithm could be improved, e.g., by blocked nested loops, we shouldn't bother because the other algorithms are still faster.

The merge join is a fast algorithm that works well for large input files if the input are suitably sorted. Since the merge logic is fairly complex, you may use up to three goto labels in the merge join if you name them such that they are self-explanatory. Make sure your logic deals correctly with duplicates in outer and inner inputs, semi- and outer join, and end-of-file, in particular of the inner file.

The hash join is also the most naive version, relying on memory to be larger than the build input.

(4) *aggr (argc, argv)* implements both scalar aggregates (if the relation named in *argv [2]* is unary, i.e., has only one attribute) and aggregate functions. In aggregate functions, all but the last attribute are considered grouping attributes, and the last attribute is the one to be aggregated. Inputs into aggregate functions must be sorted on the grouping attributes. The aggregate function to be performed is given in *argv [0]*. It can be "count," "min," "max," "sum," or "average." If it is "count," the aggregation attribute in the output will be of type "T_INT."

(5) *sort (argc, argv)* sorts the relation named in *argv [2]* into the new relation named in *argv [1]* on the attributes given in *argv [3]*, *argv [4]*, etc. If no attributes are given, it sorts on all attributes. If the keyword in *argv [0]* is *unique*, duplicate records are eliminated while writing the output relation. The sort module uses the *qsort()* routine in the C library to sort memory loads of data. However, since database sets tend to be larger than memory, a merge sort should be used instead. For very large files, this would be a multi-level merge sort.

5. Views

The current software includes the possibility of defined views as relational algebra expressions and using them like sorted relations. The complete module view.c is provided.

6. Buffer Management

The *buffer* module allows fast access of frequently used pages by managing in-memory copies of disk pages using a fixed number of page frames. It is a software layer between the record and file routines and the operating system. Pages are brought into the buffer upon request by the record routines, and they remain in the buffer until their buffer space is needed to satisfy other requests. The buffer manager uses a circular doubly-linked list to implement the *least-recently-used* (LRU) replacement algorithm. Hashing is used to speed up the search algorithm in the buffer, with a circular doubly-linked list for each hash bucket. Routines to manage lists are provided.

A descriptor is maintained for each page frame in the buffer. It contains the file and page numbers, the valid and dirty bits, a fix count, and elements of the lists above. The valid bit indicates that a page frame contains a valid disk page of an open file. All valid pages are entered in a hash chain according to the hash function. When a page is requested from the buffer manager, it is *fixed* in the buffer, meaning that it cannot be replaced. Since a page can be fixed multiple times, e.g. in a multi-user environment, a *fix count* is kept for each page frame in the buffer. When the requesting routine is done with the page, it *releases* the page. The buffer manager is then free to replace the page as dictated by the page replacement algorithm. Pages that are not valid or whose fix count is 0 are subject to replacement, and are kept in the LRU chain. When a page has been modified in the buffer, it is set *dirty*. When a dirty page is removed from the buffer, the disk page is overwritten with the updated copy.

In order to make best use of the buffer, the record routines should be rewritten such that records never span two pages, and copying is avoided when possible. *Get_record* and *put_record* fix the appropriate page in the buffer, copy the requested record, and release the page. Using a single routine to implement *get_record* and *put_record* probably does not pay off anymore. *Append_record* appends an entire page at a time, inserting the record slots not used immediately into the free list. *Delete_record* performs the necessary updates in the buffer instead on relying on *get_record* and *put_record*. *Get_next_record* and *find_record* test records in the buffer, thus avoiding the expense of copying unsuitable records from the buffer.

Copying is done using *bcopy*, a UNIX library procedure with on-line documentation. The rewritten record routines do not use any UNIX system calls; *bu_io* is the only routine to use *lseek*, *read*, *write*.

Notice that *close_file* must flush remaining pages from the buffer before closing a file.

Following is a list of routines in the file *buffer.c*. The first four routines are internal to the buffer manager, i.e., they are only called by other buffer manager routines.

(1) *bu_io (io, file_no, page_no, frame_no)* reads or writes a page of an open file from or into the buffer frame indicated by *frame_no*. The *io* parameter points to the required system call.

(2) *int bu_find (file_no, page_no)* attempts to find a page in the buffer pool and returns either the slot number or *-1* in case of failure.

(3) *int bu_alloc (file_no, page_no)* allocates a buffer frame. If the LRU chain is empty, the buffer is overcommitted and an error is returned. If necessary, the frame's old content is written to disk. *Bu_alloc* does not read the page from the disk. It initializes the buffer descriptor and performs the necessary adjustments of the hash lookup structure. On return, the fix count of the new page is *1*. It returns the allocated slot number or an error number.

(4) *bu_initialize ()* initializes the static data structures of the buffer manager. It is called only once when the first buffer request is made.

(5) *bu_new (file_no, page_no, addr)* allocates a frame for a page that was newly allocated to a file by invoking *bu_alloc*. It is an error if the page already exists in the buffer. It passes back the main memory address of the frame through *addr*. This routine is called only by *append_record*. If necessary, it first calls *bu_initialize ()*.

(6) *bu_read (file_no, page_no, addr)* ensures that the desired page is in the buffer pool, and passes back its address in *addr*. If the page already was in the buffer pool, it is removed from the free list if necessary and its fix count is increased. Otherwise, *bu_read* invokes *bu_alloc* to allocate a page frame and *bu_io* to read the page from disk. If necessary, it first calls *bu_initialize ()*.

(7) *bu_setdirty (file_no, page_no)* marks the indicated page dirty. It is an error if the page is not in the buffer or if the page is not fixed.

(8) *bu_release (file_no, page_no)* releases the page in the buffer by decreasing the fix count. If the page becomes unfixed, it is included in the LRU chain.

(9) *bu_flush (file_no)* removes all pages of an open file from the buffer, if necessary after writing them.

Hints

(1) Study the *defs.h* file to understand the data structures.

(2) Write pseudocode for routines before you hack C. This will save you a great deal of time in the long run.

(3) A primitive test program for records and files is provided in *test1.c*. After the record and file routines are developed, you can compile and link them with the test program by typing "make test1". The three upper levels can be tested using the interactive program *dbmgr*. The command language of this program is described in the file *syntax*.

(4) It is possible to implement *get_record* and *put_record* with the same procedure (*io_record()*) using an additional parameter. You are not, however, required to do so.

(5) Many procedures are already implemented.

(6) Use compiler flags (e.g., DEBUG) to control compilation of debugging code and whether or not to use the buffer management routines.

(7) An example for the usage of the condition function of *find_record* is given in *test1.c*.

(8) A buffer slot is in the buffer manager's LRU chain if and only if it is either invalid or it is valid with a fix count of zero. A buffer slot is in the buffer manager's hash buckets if and only if it is valid.

(9) The macro *baseof* can be used to find the address of a buffer descriptor record from the address of one of its fields, e.g. an LRU chain element.

(10) When uncertain whether a problem is a problem with the buffer code or with higher level routines, using the record routines that do not use the buffer module might help decide.

(11) The file *misc.c* contains routines that may help you during debugging.

Software Tools

Make is a UNIX utility with on-line documentation. There is a file *Makefile* provided for your convenience. By default, all files are compiled with *-DDEBUG*. This can be overwritten by using *make C=*. In order to compile with a special compile flag, e.g. *-Ddebug_record*, type *make D=-Ddebug_record*.

It is recommended to use the *revision control system*, a family of UNIX utilities with on-line documentation (rcs, ci, co, rlog, rcsdiff). Creating a directory *RCS* before using any of these commands separates working files from version files. Typing *make ci* saves a version of all files. It can be retrieved with *make co*. For saving or retrieving a particular version, e.g. version 2.1, use *make REV=-r2.1 ci* or *make REV=-r2.1 co*. A log message can be included with the version checked in by supplying the message as follows.

make ci log='record and file routines work ok with test1.c'

To examine the history of a particular file, say record.c, you can type

rlog record.c

If you have edited your programs in such a way that you have lost track of nesting, indentation, etc., you may wish to use the C beautifier *cb* or the indentation program *indent*. You may find the type checker *lint* and a debugger (e.g., *dbx, gdb*) useful when your program's errors prove hard to find otherwise. All these programs have on-line documentation.

Coding Conventions

Type and constants are given in the file *defs.h*. They should be used whenever appropriate. The external variables are declared in *defs.h* and defined in *globals.c*. Some system calls, e.g. *close*, return either 0 or a negative number; their return value should be compared with 0. If a specific return value is to be expected, e.g. from *lseek* or *read*, this value should be tested. In general, numbers instead of named constants are bad programming style.

All routines must return an error code, e.g. *OKAY* or *ERR_EOF*. The only exception to this rule are the routines that are used only within their module, i.e., the routines which are declared ‘static’. The error codes in *defs.h* should be used; if additional error codes seem necessary, numbers less than -100 should be used and declared in the file *my_defs.h*. All return codes must be checked. The code is easier to write, debug, and read if the response to an error follows immediately its detection in the code file.

It is not required to comment the provided routine fragments, e.g. their functionality or arguments. However, the code should include brief comments explaining what each ’paragraph’ (about 2-7 lines) of code does. Separating such paragraphs by empty lines and following a consistent indentation scheme further improves the readability of the code. Using self-explanatory variable names and reusing the same abbreviations everywhere also helps.

The provided C code is unfortunately not ANSI C code.

Project Milestones

Date	Modules working
Fri 15 Mar	files & records without buffer management (4 routines - 40 to 60 lines)
Wed 3 Apr	relation, schema routines (4 routines - 75 to 100 lines)
Wed 10 Apr	project and append routines (3 routines including compile_proj - 100 to 120 lines)
Fri 19 Apr	join routine (1 routine - we’ll only implement hash join - 60 to 80 lines)
Fri 26 Apr	above using buffer management (5 routines - 60 to 80 lines)

Testing and What to Hand In

For the final milestone, you must schedule a time to meet with the class teaching assistant to test your code. For the intermediate milestones, you will submit your source code and a makefile to be tested. You are not required to be present for testing of the intermediate milestones. The intermediate milestones are primarily meant to measure your progress and identify any problems early on.

For the final milestone testing, you must also hand in a hardcopy of your source code. (Remember to write comprehensible code and to document it with comments. If I cannot understand your code, your grade will be lowered.) Please remember that I do not accept late work.

You are expected to work alone on the project. **The final exam will contain questions you will only be able to answer if you have written the required code**.

How To Get Started

Copy the compressed tar’ed file */n/homeserver/w/cs425/1996/dbmgr.tar.Z* into your directory. To untar/uncompress the file, type:

```
zcat dbmgr.tar.Z | tar -xf -
cd studentversion_425
ls
```

When untar’ing the file, it will create a directory called *studentversions_425* in your current directory. You should now have all required files you need. In the root directory created, you will see two subdirectories, *docs* and *src* respectively. This file along with a couple others are under the *docs* directory (read the the documentation prior to making code changes). Next, change directories in to the *src* directory and initiate the version control system with these provided files. While you do not have to use RCS, I strongly recommend it. Type

```
mkdir RCS
make ci
```

ls

The *ls* command will show that most of the files are gone, i.e., they are checked in (hence, the ci) and saved in the RCS directory. To get them back (check out - hence, the co), type

make co
ls

I recommend frequent check-in/check-outs to checkpoint your code.

Now you can start developing code. When you are ready to compile and test it, say with test program *test1.c*, type

make test1

You will have to edit the file *makefile* if you want to use it for other test programs you have written. Later on, when you use the interactive system, you will use

make dbmgr

Finally, in order to save disk space while you are not working, please remove all **.o* files at the end of each day by using the command *make clean*.

Good luck, and have fun with the project.