

Implementation of a Rudimentary Database Management System

Michael Huynh

AP Computer Science, Upper Darby High School

November 2003

Objective: To develop a database management system utilizing sequential access in flat-file databases in the C++ programming language.

1. BACKGROUND DISCUSSION

Moving at the speed of light, millions of gigabytes of data are shuffled around every day. In its raw state, data is useless. It is not coherent and has little relevance. It is therefore imperative that this data be arranged and organized into a database for ease and speed of search and retrieval¹.

The tedious task of maintaining and processing data has been delegated to computers today. Databases are crucial to the daily operations of the government and businesses. They store information about people, transactions, and maintain huge lists of inventory and other information. The applications of a database are endless.

Monstrous corporations such as Oracle, Microsoft, and IBM realized the profit potential in the billion dollar database market and have written database management system (DBMS) software. Their DBMS solutions are very popular and are used by many major institutions. For the smaller business or organization that cannot afford the high price of one of the high end database license, there are also low-cost or free DBMS such as MySQL and PostgreSQL offered by the open source community that is just as effective as its expensive counterparts.

The general functions of a DBMS are simple. A basic system offers the ability to view, create, edit and delete records as well as the capacity to search for records. Lying under this seemingly simple interface, however, are more foundation functions that perform input and output functions to the database file, extract individual fields from records, and keep the database records in a sorted manner.

2. PROGRAMMING PROBLEM SOLUTION

The development of a DBMS must start from the foundations. Since the DBMS objective was to model basic functionality, a flat-file was used to store the database in ASCII encoded text. In order to reflect practical application, the database

managed various computer information for a college or a corporation. There were five fields in this database that stored: five-digit id numbers, three character computer types, ip addresses, mac addresses, and a one character building location. All of the fields were encoded as a single string record separated by a delimiter [p5 279]. For this database, the delimiter was the character: '|' [p1 46], but it could have been any character that did not conflict with the stored information. Using a delimiter enabled the DBMS to quickly deconstruct a record into its individual fields by using string commands and control structures [p9 25-50]. Each string represented a database record and looked like [p14]:

```
00003|CPQ|192.168.1.2|4F:66:A3:14:5B:89|A|
```

A collection of these strings in a one dimensional array constituted the database [p1 55]. Storing the database on media meant streaming the array of strings into a file with each string on a new line [p10 88-104]. Reading a database from media reversed the process: the file was opened and its contents were placed in an array of strings [p9-10 53-85].

With the foundations laid, database functionality was added next. To view the database, the database array was sent to output with formatting [p4-5 191-256]. Filling a new element of the database array with user entered data created a new record [p5 258-286]. Editing a record meant changing the fields of a deconstructed record, reconstructing it, and updating the database with the modified record [p5-6 288-334]. To delete a record, the database array is reorganized and the record is omitted from the new array (Another way to delete a record would be just to flag it for remove during the writing process) [p6-7 336-362].

A DBMS would be very shallow if it did not include the ability to search a database for certain records. One of the fastest searching algorithms is binary search which uses a divide and conquer approach on a sorted array to find a matching value

[p13]. Binary search is particularly advantageous for large databases since the searching time in worst case conditions can be expressed as $O(\log n)$ where n , the number of elements, is a function of Big-O that measures the algorithm efficiency and run time². However, binary search requires that the elements in an array are already sorted. Thus, this DBMS also enlists the use quick sort, one of the fastest sorting algorithms, that also works on a principle of divide and conquer [p11-12]. Although quick sort suffers from a worst case sort of $O(n^2)$, the average-case run times are $O(n \log n)$.

Finally, all changes to the database are kept in memory for efficiency and for the integrity of data. When the interface to the DBMS exits, the contents of the database in memory are then transferred to the flat-file [p3 152-156]. In this way, valuable time is conserved by refraining from constant disk input and output.

3. PSEUDOCODE

```
//Read the entire database into an
//array of strings.
totalrecords =
ReadDatabase(filename, db_array);

void DisplayRecords(database[],
totalrecords) {
    //Print out records
    for(recnum=0;recnum<totalrec
ords;recnum++)
    {
        SplitFields(database[r
ecnum], fields);
        Cout<<fields;
    }
}

void AddNewRecord(database[],
totalrecords) {
    //Prompt for field data
    line=GetFieldData();
    //Place new data in new
    //array element
    database[totalrecords+1]=
        line;
}

void EditRecord(database[],
totalrecords, linetoedit, newdata)
{
    SplitFields(database[linetoe
dit], fields);
    field[]=newdata;
    //Reconstuct the line
    ReconstructLine(field);
}
```

```
}

void DeleteRecord(database[],
totalrecords, linenum) {
    database[linenum]='';
    Reconstruct(database);
}

void SearchRecords(database[],
totalrecords, searchfor) {
    position=BinarySearch(databa
se, searchfor);
    //Found at:
    database[position];
}

int SplitFields(line, field[]) {
    //Loop through the line and
    //split at |
    fieldcount=0;
    for(i=0;i<line.length();line
++) {
        if(line[i]=='|') {
            //Store part of the
            //line in field[]
            field[count]=line.subs
tr(start, i);
        }
    }
}
```

4. CODE

Actual program code can be found attached to this document.

5. VALIDATION

The program was put through multiple test runs certifying each function. Visual verification screen output and file data was also used to ensure that the program output and actions were correct. For instance, if a record was deleted, the database file should not contain the record. If a field was edited, the database file should reflect that change. Quick sort functionality was validated by inspecting output, and binary searching was verified if the search returned the correct match. This program is believed to be free from errors.

5. SUMMARY

The program was successful in serving as a simple DBMS. It provided all the basic functionality that a true DBMS required, and developed solutions to work in a flat-file database. The program also successfully implemented the concepts of sorting and searching. The concepts

used for the simple DBMS can be applied towards understanding a far more complex DBMS.

REFERENCES

1. "Database." The American Heritage® Dictionary of the English Language, Fourth Ed. 2000.
2. Teukolsky, Roselyn. How to Prepare for the AP Computer Science Advanced Placement Examination. Barron's Educational Series, 2001.
3. Borland Turbo C++ function help file. 1994.