```
(********************** SplitBase Data Management Systems ********************
 *                                                                           *
 *                        Copyright (c) 2001 Leon O. Romain                  *
 *                                                                           *
 *                                  leon@kafou.com                           *
 *                                                                           *
 *****************************************************************************)
```

HISTORY

I conceived SplitBase about 8 years ago as an alternative to the available Database Management Programs and also to alleviate some of the burden imposed on programmers by Borland's BDE. During that time, I have been involved on projects that either did not require database programming or those for which the clients requested specific systems. I had also hoped that by now the public domain and/or the open-source communities would have been flooded with either good or excellent alternatives to both the BDE and commercial databases. Although there have been many outstanding implementations in that direction, none of them had addressed all the issues I am, to this day, still concerned with. Now, being faced with the prospect of many data management projects in my immediate future, I decided to finally start working on my ideas, and that is how I came up with SplitBase.

THE CONCEPTS AND PHILOSOPHY

Simplicity, efficiency and speed are the guiding principles behind SplitBase, and the reasons I prefer to program in Delphi. Delphi allows the creation of powerful applications with all their functionality confined in a single small executable file. In many other popular programming environments, a “Hello World” program necessitates a complex multi megabytes sized installation program in order to run in a user’s computer. Dynamic link libraries are a wonderful concept that ran out of control. They often create serious conflict or even worse, overwriting them sometimes may cause other programs to crash or behave erratically. I wanted to develop a structure that was simple enough for average programmers to comprehend and modify easily according to their needs. That structure should be efficient, fast and also powerful enough to be used in real world situations. By efficient I mean it should be able to manage substantial amount of data (at least one million records) safely and accurately. Fast refers to the ability to execute available transactions instantaneously (one fifth of a second or less). Efficiency and speed should not rely on the power of today's processors and other sort of equipment, but rather should be achieved on minimal machines such as the original IBM PC XT and AT with all their relative limitations. I believe that with SplitBase I have achieved my goals and I humbly present it to your judgment while hopping that it might be useful to other programmers and that they may improve upon the quality and functionality of SplitBase according to the preceding guidelines.

DESIGN

In its current incarnation, SplitBase is a multilevel indexed file structure. Simply put, it is composed of abstract containers of index data or keys as they are more commonly referred to. These keys belong to a structure that attaches them to a simple pointer that relates them to the corresponding records or other lower level containers. When a container reaches its full capacity, it splits into two half filled containers. It is very similar to the B-Tree structure used in many Database indexes, but it is simpler to manage and store. When full, the top-level container, that never splits, indicates that the Database has reached its maximum capacity and cannot accommodate any more records.

In its present phase, SplitBase consists of two levels of key containers: a top level with a single container and a secondary level with multiple containers. The top level contains a duplicate of the first key of each secondary container. Attached to that key is a pointer to the physical position in the database file of the related secondary level container. When a secondary container splits or when its first datum changed, that information is updated in the top container. Secondary-container keys are attached to pointers to the location of the actual records within the file. The top-level container always resides in random access memory (RAM) when a database file is active. That is whenever it is opened or created. This reduces the number of disk access to only two to locate and retrieve any record. To write new records, disk access is also limited to an average of two or a maximum of three if the top-level container must be updated.

At this point, it is important to note that the maximum number of records that can be entered in a two level container system as described above, is equal to the square of the full capacity of a container. In the worst case scenario, when all secondary-level containers are only half full and the top-level container reaches full capacity, that number is reduced by half, thereby fixing the guaranteed minimum number of records that can be entered in the SplitBase system. New data such as containers generated by a split and all new records are always written to the end of the database file and a pointer to their position within that file is attached to the corresponding key. A SplitBase file is therefore a moderately complex linked list of containers and records. In its original state when the file is created, it only contains an empty top container and a header that provides the manager with information about the file structure such as length of records and of fields or the number of deleted records.

For the sake of simplicity, I have decided to save all fields in a record as strings that can be easily converted, in most programming languages, to other data type such as numbers, currency, date etc. Data within the containers are automatically sorted in ascending order with a simple insert sort algorithm. This allows for extremely fast binary search of data in RAM. For every search and insertion request, the manager looks up the top container to determine which secondary container holds the position for that specific key. It then uses the corresponding pointer to locate and load that container into RAM. New data are then inserted into their correct position and found data provide the pointers to their actual records for retrieval. The whole system is then updated if necessary with very minimal disk access.

The decisive factor in determining container capacity has been available memory in our targeted minimal computer, the XT and AT version of the IBM PC. Both machines have been plagued by the 64k block limit, which was chosen as the maximum physical size of a container. On the other hand since only two containers are present in memory during data manipulation this will allocate for more than enough room in the limited memory of these machines (640k max.) for the records, the database manager, the operating system and other programs. In our original model, we propose a maximum key length of 26 bytes, which is appropriate for the majority of real life situations where keys are usually simple and short such as a social security number, a last name, a phone number, a zip code etc. This choice provides a capacity of 2000 key/pointer combinations yielding to databases of a maximum of 4 million records and a minimum of 2 million records if data was entered sequentially resulting in all half full split indexes by the time the top container reaches its full capacity.

If you did not understand anything in what was said in this section, you should not be programming databases anyway. However the functions in SplitBase are few

and much simpler than with other databases; hence you should be able to use it without too much difficulty.

IMPLEMENTATION

SplitBase is built around a small set of Boolean functions. It is composed of 29 structurally identical functions that return true when the function succeeds and false otherwise. Only 20 of those functions are necessary for a programmer to know in order to use SplitBase. Many of these routines are not structural but complementary and were implemented only for the convenience of the programmer and the integrity of the system. All the others are mainly basic but optimized file manipulation routines. All functions are written in simple, object free turbo Pascal syntax with very few exceptions. This simplicity makes it extremely easy to port SplitBase to other programming environments and early versions of turbo Pascal. It also renders the system compatible with current and future releases of Delphi with practically no modifications. It could even be adapted as easily to run in the original Pascal language but I would not recommend anyone to do so simply because Wirth's Pascal did not allow the programmer to access files and memory directly. This would result in a very slow system and would definitely not meet our previously stated standards of speed and efficiency.

In this current version, SplitBase is released in two formats. An include file (splinc.pas) to be included in Turbo Pascal or Delphi programs, and a component (Tspl) that can be added to practically all current releases of Delphi and most likely all future implementations unless they contain some very dramatic structural modification in their basic syntax or functionality.

INSTALLATION

For the include-file version, just add the compiler directive {$I Splinc.pas} at the top of your Turbo Pascal programs or at the beginning of the implementation section of your Delphi projects. For the component version, install the file spl.pas as a new component according to the methods available in your version of Delphi. You can install it in an existing package or a new one preferably Tspl. At any rate, if you follow the right steps you should end up with a color logo of SplitBase in the "Samples" palette of your Delphi environment. Drag and drop that logo into your forms to use the Spl component that will provide you with all the functionality of SplitBase. The file Splinc.pas is located in the "\Splbase\Include" sub directory of the original release of SplitBase. The component Spl.pas is located in the "\Splbase\Class" sub directory of the same release of SplitBase.

PROGRAMMING

SplitBase is very simple to program. The following is a brief description of the necessary functions as well as basic principles that should be observed to safely and easily implement these routines. It is the programmer's responsibility to allocate memory space for the system by using InitSpl and ReleaseSpl at the beginning and end of your program. You must also initialize system variables by using InitBase before creating or opening any database file. You have to ensure that your function calls are successful before proceeding with any database session. This can simply be achieved by checking the return value of all your calls. By following the preceding principles you should enjoy a trouble free experience while programming and using SplitBase. Here are the main functions:

InitSpl: Initializes the Splitbase system by allocating memory to accommodate the top and secondary system containers. Initspl must be called at the beginning of your programs. If it fails, all other calls to the system must be cancelled.

ReleaseSpl: Releases allocated memory back to the operating system. This function must be called at the end of your programs. Failure to do so will prevent other programs to use that portion of memory reserved for SplitBase.

InitBase: Initializes important variables used by Splitbase. Initbase must be called before creating or opening SplitBase files. Failure to do so may generate unpredictable errors or otherwise compromise the stability of the system.

SetSpl: This is a very important function that you must use to tell SplitBase the size of each field in a new SplitBase file. It must be called before creating a new database and uses a string as parameter. This string contains the size of all fields that define the record structure. Each field size is described with three characters starting with the first field in the record. Fields which sizes are less than 100 should be filled with zeroes on the left of its allocated portion. For example, a field of length 3 followed by a field of length 12 should be written as follows: '003012'.

CreateSpl: This function receives a string parameter to create a new SplitBase file. The parameter is the name of the file to be created without any extension. The system will add the proper extension to it, currently (.spd), and create the file if possible. The functions InitBase and SetSpl must be called to set the necessary variables prior to calling CreateSpl.

OpenSpl: This function requires a string parameter to open a SplitBase file. The parameter is the name of the file to be opened without any extension. The system will add the proper extension to the name, currently (.spd), and open the file if possible. A call to the InitBase function must be made to set all the necessary variables prior to calling OpenSpl.

AddField: This function uses two parameters, a string and an integer, to insert a field into the record structure of the SplitBase system. The string contains the value of the field and the integer its index. All fields must be inserted before saving a record into the file.

GetField: This function uses an integer parameter to retrieve a field from the record structure of SplitBase. The parameter contains the index of the field. A record must be retrieved from the data file before using this function. Getfield must be called once for each field in the current record. If the function fails, SplitBase returns an empty string.

AddRec: This function saves a record into the SplitBase file. It receives the key attached to that record as a string parameter. All fields must have been inserted into the record structure prior to calling AddRec.

GetRec: This function retrieves a record from the data file. It receives the key attached to that record as a string parameter. If the record is found, all fields are loaded into the record structure of SplitBase.

DelRec: This function deletes the current record from the data file. A record must be active in memory in order to use DelRec. DelRec has no parameters.

ModRec: This function updates the current record in the data file. It actually deletes that record from the file then adds the modified record immediately. It uses a string parameter as key.

FirstRec: Locates and retrieves the first indexed record.

LastRec: Locates and retrieves the last indexed record.

NextRec: Locates and retrieves the next indexed record.

PrevRec: Locates and retrieves the previous indexed record.

RecCount: Returns the total number of records in the data file.

ActiveDb: Returns true if a data file is active or opened.

ActiveRec: Returns true if a record is active or loaded.

DBEmpty: Returns true if the current data file is empty.

Many variables are important in the use of SplitBase. Most of them should never be manipulated directly by the programmer. However, there are a few such as **Splerr** that the programmer should check regularly. **Splerr** returns the last error number and description. These values are returned with the following calls: **splerr.recnum** and **splerr.recstr**. In the component version the properties **ErrorNumber** and **ErrorString** must be accessed in lieu of splerr. The variable **CurDtb** or its equivalent **CurrentDB** contains the name of the current data file. The **allrec** variable contains a lot of valuable information such as the number of fields in the current SplitBase record. You may read it with **allrec.size** or with **FieldCount** from the SplitBase component. Finally **limrec** is a number you can set to limit the number of record that may be entered into the data file. For the component, use **Reclimit** instead.

RELATED PROGRAMS AND TESTS

The original release of SplitBase comes with two test programs Split and Split2. These two programs are identical in looks and functionality except that the former uses the include file and the other the component. These programs allow a user to test the major functions of SplitBase. They contain buttons to create new data file and open existing ones. Other buttons also allocate for adding new records as well as finding and deleting existing data within a splitbase file. They also provide navigation buttons to locate the first, last, next and previous records. Finally a generate button will automatically insert 1 million records into a specific data file. That file is made of a two fields record and is called test.spd. It is automatically created and opened if you click the related buttons in the program. Generate will automatically add 1 million even numbers from 2 to 2 million to that file. For fun and to simulate data entry overhead, generate will convert all those numbers into their English spelling and put the actual number in the first field and its spelling in the second before saving the record in the data file. These two fields are also output in two edit fields for added overhead. An algorithm is used that forces SplitBase to update the top container every four records that in real life applications should only happen on average every thousand records. Test on a 667 megahertz custom built computer equipped with a 5400 RPM Ultra ATA/66 hard drive yield a result of 139 records per second at completion of generate. The function can be stopped at any time by clicking on the 'Stop' button.

FUTURE EXPLORATIONS

Where do we go from here? Well from the basic functionality of the system many routes may be considered. One of the more obvious may be to increase the number of records that the system can handle. The easiest way to do this might be to hold the keys in the top and secondary containers into an ANSI string just like 'rechld' that holds the fields in any given record. The two-gigabyte theoretical

limit claimed by Borland is well beyond anything we might need in any real world situations.

The LongInt and Int64 Problem

The only other hurdle is the 2-gigabyte limit of the longint type that is used to determine file sizes and to locate records and containers within the data file. This problem may be addressed by using a blocking factor greater than 1 for creating and opening data files. Two hundred fifty-six (256) and one thousand twenty four (1024) may be good values to experiment with. The only problem might be in synchronizing container and record size with the chosen blocking factor. It would be easy to add fillers like in the good old days when Cobol was king. This will certainly add to the waste in a data file. But who cares in times when 80-gigabytes hard drives cost less than 200 dollars in most major computer stores. An easier way would be to use the new int64 type that appeared with version 4 of Delphi. But Borland is mute about using it with the reset, rewrite and seek procedures. Int64 can hold up an integer the size of 2 to the power 63.

What about garbage collection? Much like in other databases, deleted records and containers are not actually erased from the file. They just sit there useless with no actual pointers relating to them. Heavy usage of delete transactions tends to increase considerably the amount of garbage within the data file. A garbage collection or rebuild routine can easily be implemented by using a new data file and add all the live records from the previous file to it.

Multiple indexes data files may be more easily implemented by saving the indexes and the records in different files. It will actually allocate for more records if longint is used. This is also the case if one would like to emulate the functionality of relational databases. The straightforward structure of the data file without the containers will make it easy to emulate most general functions of a relational database system. In the case of a multi-user environment and client/server implementation, a separate manager should be written to handle calls from the different users. The most important issue in multi-user system is probably the ability to lock records. The manager can easily implement this by adding an extra field to records in data files that holds the status of that record. A simple protocol should also be implemented between the manager and the client programs to properly handle transactions.

These were a few suggestions for programmers to modify the SplitBase system to easily achieve their goals. However, we believe that SplitBase is very useful as is and can be adapted to solve a great deal of real life problems. Particularly in the retail industry Splitbase may be easily and effectively used without modification to help manage most problems such as stock, accounting, customers etc… How many times did the clerks used complex data analysis to locate your record or a specific item in the store? They usually type a single key string in order to locate the needed information. Most stores even the major chains hold less than 2 million items (I know I usually go around and count them :-). Many cities in America and around the world have less than two millions citizens, most libraries hold less than two million books and the list can go on and on about real life situation in which SplitBase may be used without modifications.

If you come up with important additions to SplitBase you are free to publish them along with the original distribution of SplitBase with all files present and unchanged. I would recommend that you follow the guidelines stated at the beginning of this file. However, if you have a REALLY important addition that uses some arcane procedures or an extremely complex routine, well go ahead,

publish it with the restrictions stated above as far as keeping original files intact.

COPYRIGHT AND LICENSES

SplitBase is not based on any previous work other than basic computer Science and Data Structure principles. It is not a public domain material but a copyrighted publication of Leon O. Romain. However, it is freeware and is being released under the GNU license agreement. See the file licence.txt for more information. You may use the SplitBase system as you see fit in your personal, professional or commercial programs without paying any loyalty to the author providing you relieve him of any and all liabilities that may result from the use of SplitBase. In fact SplitBase is released as is. The author decline all responsibility and liability from damages either direct or incidental that may result from the use of SplitBase and its accompanying routines and other software.

KNOWN BUGS

There are no known bugs. If any are found, they should be easily corrected due to the simplicity of SplitBase and its modular structure.

ADDENDUM

I checked the int64 type and it worked perfectly with the reset and seek functions on files much bigger than 4 gigabytes. This yields the possibility of creating SplitBase systems capable of accessing billions of records with very little modification to the original program when using Delphi version 4 or later. By changing the type of the necessary variables from longint to int64 and by modifying the index field of the splitbox record to work as an ansistring instead of an array is all that it takes for SplitBase to access those billions of records. I also added an activex version of SplitBase to the original release. It works fine except for the fact that the icon does not appear on the form when selected. The necessary files are located in the activex subdirectory and the file SplBaseXControl1.ocx must be registered before using the control. This opens the door for visual basic, c++ and other programming environments to use SplitBase before the advent of native implementations of the system. OCX controls may be registered using the Regsvr32.exe utility usually located in the windows or system directories.

COMMUNICATION

For any comments, suggestions, corrections, criticism, bug reports and other communications please Email me at leon@kafou.com.


S
P
L
I
T
B
A
S
E