Indexed Sequential files in Ada.

A didactical example.

By Marc A Gobin

1. Objective.

The working of a data base management system can informally be explained in terms of an interrelated set of indexed files, together with the necessary routines to access all the elements of the file.

The routines, one can find in a DBMS, should be implementable in a straightforward way from the routines available in an indexed file system. This imposes some requirements on the indexed file system to be built.

2. Requirements.

Th indexed file system should be implemented in such a way that following requirements hold.

- Each (logical) record of the file is identified by one key. This is the main key of the file, sometimes referred to as key 0. Each data record has a different key. The key is part of the record. An example of such a uniquely defined key is the personal number given to each person of a company.
- Defining a certain number of secondary keys must be possible. These keys are not necessarily unique. If the name of a person is considered as such a secondary key, it is very well possible that two persons have the same name. It is even possible that all keys have the same value (a not very interesting key).
- The keys may be of any type or combination of types.
- Direct access to any record should be possible when one of the keys is given. In the case of a secondary key, where several records can have the same key value, the first of these records is given.
- Sequential access must be possible. The sequence is determined by one of the keys and the data records are retrieved in ascending order of the specified key. If more than one record has the same key, they are retrieved in ascending order of the main key.
- Updating a record must be possible. Even the secondary keys can be modified. The main key can not be changed. If this is needed, the record with the old key must be deleted and the record with the new key must be written to the file.
- In order to achieve its didactical purpose, the specification of the package should be as close as possible to the specification of the package Ada.Direct_io.

- The implementation should be simple enough so that it can be explained in a minimum of time and with a minimum of knowledge. Yet the implementation must be efficient enough so that potential users can be satisfied.

3. The specification.

```
with io_exceptions, aux_io_exceptions, Ada.Direct_io ;
generic
   type element_type is private ;
package indexed_io is
   type file_type is limited private ;
   type file_mode is (in_file, inout_file, out_file) ;
   type relation_type is (greater_equal, equal, greater) ;
   procedure create (file : in out file_type ;
       mode : in file_mode := inout_file ;
       name : in string := "" ;
       form : in string) ;
   procedure open (file : in out file_type ;
       mode : in file_mode ;
       name : in string ;
       form : in string := "") ;
   procedure close (file : in out file_type) ;
   procedure delete (file : in out file_type) ;
   procedure reset (file : in out file_type ;
       mode : in file_mode ;
       key_number : in integer := 0) ;
   procedure reset (file : in out file_type ;
           key_number : in integer := 0) ;
   function mode (file : file_type) return file_mode ;
   function name (file : file_type) return string ;
   function form (file : file_type) return string ;
   function is_open (file : file_type) return boolean ;
   procedure read (file : in out file_type ;
              item : out element_type) ;
   generic
       type key_type is private ;
       default_key_number : integer := 0 ;
   procedure read_by_key (file : in out file_type ;
            item : out element_type ;
            key : in key_type ;
            key_number : in integer := default_key_number ;
            relation : in relation_type := equal) ;
   procedure write (file : in out file_type ;
            item : in element_type) ;
   procedure update (file : in out file_type ;
            item : in element_type) ;
   procedure delete_element (file : in out file_type) ;
   function end_of_file (file : file_type) return boolean ;
   status_error : exception renames
                io_exceptions.status_error ;

   mode_error : exception renames
```

```
                io_exceptions.mode_error ;
   name_error : exception renames
                io_exceptions.name_error ;
   use_error : exception renames io_exceptions.use_error ;
   device_error : exception renames
                io_exceptions.device_error ;
   end_error : exception renames io_exceptions.end_error ;
   data_error : exception renames
                io_exceptions.data_error ;
   key_error : exception renames
                aux_io_exceptions.key_error ;
   existence_error : exception renames
                aux_io_exceptions.existence_error ;
private
              -- implementation defined
end indexed_io ;
```

Notes :

The types, most procedures and all the functions are similar to those defined in the package Ada.Direct_io. They have also the same meaning.

The procedure *create* has one important modification. The form parameter is not an empty string, but contains the information concerning the record length, the length and position in the record of each of the keys. This position and the length are expressed in the number of bytes.

Here is an example of the contents of a form parameter :

```
"Record ; size 200 ;" &
"Key 0 ; length 6 ; Position 0 ;" &
"Key 1 ; length 20 ; Position 6 ;" &
"Key 2 ; length 4 ; Position 64 ;"
```

The data records have a length of 200 bytes, the main key has a length of 6 bytes and starts at position 0 of the data record. There are two secondary keys with lengths 20 and 4 respectively and positions 6 and 64. A maximum of 9 secondary keys is allowed.

The procedures *reset* contain a key_number which allows a user to position the system at the first record according to the specified key.

There are three new procedures :

- *Read_by_key* is a generic procedure that performs a direct read from the data file for a given key. The type of the key and the key number are the generic parameters. The procedure itself has five parameters :
  - The file from which the record is to be read.
  - The zone in memory that will receive the retrieved record.
  - The value of the key.
  - The key number.
  - An indication equal, greater or greater_equal, allowing to search for a key that is =, > or >= than the specified key.
- *Update* is a procedure that allows an existing record to be updated. If there is no record with the same main key, the exception *existence_error*

is raised. For *write* there may be no record with the same main key, otherwise *key_error* is raised.

- The procedure *delete_element* allows the suppression of the last retrieved record.

4. The physical implementation.

On creation, there will one data file and one file for each of the keys declared in the form parameter of the *create* instruction. All these files are of the same type i.c. *Ada.Direct_io.-file_type*. The corresponding package, an instantiation of Ada.Direct_io is declared with an element_type of a string of 512 characters.

In the data file all records (of the own generic type *element_type*) are split and/or glued together as to fit succeeding blocks of 512 characters. The generic type *element_type* is a fixed length type. The order of the records in the file is of no importance. They are put in the order of arrival and there is a linked list of deleted records so that new records are first used to fill up the gaps that are left by formerly deleted records.

A block in the key files contains following information :

	key ptr	key ptr	key ptr	

The first field contains the number (in two bytes) of the first free byte in the block. All other fields are composed of a key and a four byte pointer. The meaning of these pointers depends on the number of records in the file and on the length of the key.

1. The number of records is so small that all keys can fit in one block. In this case all keys are kept in ascending order in this unique block and the pointers point to the data record in the data file (record number and position of the first byte). The number of records that can be serviced depends on the length of the key. This number is equal to 510 / (l + 4) where l is the key length. There must be at least two records in one block, which means that no key can be longer than 251 bytes.

2. A two level system is used when the number of records exceeds the number of records of the previous paragraph. Let us consider a key length of 10 bytes. This means that 36 keys can fit in one block. If there are 1000 records, they cannot fit in one block. They are spread over some 30 blocks. In each block the keys are in ascending order. One additional block contains an ordered list of the 30 last keys of each block. This last additional block is the level one block. In a search proces the first level block is read and the first key encountered that is larger than or equal to the searched key gives us a pointer to the block containing the right key. This one points to the data record.

3. If the number of data records increases, the number of levels may increase also. Remark that for n levels the total number of records can be as large as k ** n, where k is the number of keys per block. For a key length of 10, k is equal to 36. The number n is limited to 20. Note that 36 ** 20 = 1E31. Even for keys of length 251, k is equal to 2 and 2 ** 20 = 1000000.

5. Searching for a record with a given key.

To find a record with a given key (K), we start reading the first record of the appropriate key file. This record contains a list of keys in ascending order, together with pointers to records in the same file, but for the next level. The first record is searched through in order to find the first key greater or equal to K. The corresponding pointer is used to retrieve a record of the next level.

This operation is repeated for each level. The pointer in the last level points to a place in the data file (record number and position in the record). In this way the data record can be retrieved.

To function properly the keys must be in ascending order within each record. But the records themselves may be in any order. The only record of the first level is the first record of the key file.

In the generic procedure *read_by_key*, the last parameter can be “greater” or “greater_equal”. If this is specified, than the key that is searched for will be strictly greater or greater or equal than K. This allows e.g. to look for a name for which only the first letter is given.

6. Inserting a new record.

The different keys are inserted in each of the key files. For each key to be inserted we initiate the search procedure and we keep in memory the different blocks for each level. In the data file the data record may be inserted anywhere. Normally it is written at the end of the file, but after the deletion of one or more records we put the new record in the same place as one of these deleted records. For each deletion we keep in fact a linked list of the available places.

In the last level of the key files, the new key is inserted if there is space left in that block. If the new key is larger than all existing keys, the new key is simply added at the end of the record and the record of the higher level is adapted accordingly. If there is no place left in the record at hand, than this record is split in two and the higher levels are adapted accordingly. If the splitting takes place on the highest level, a new level is added the highest level containing only the two blocks resulting from the splitting.

The splitting can be visualised as follows (*a* and *b* are block addresses) :

a : key 1 key 2 key 3 key 4 key 5

On inserting a new key between key_3 and key_4, the modified blocks are :

a : key 1 key 2 key 3
b : newkey key 4 key 5

In the higher level the indication “key 5 a” is replaced by two indications : “key 3 a” and “key 5 b”. If there is not enough place for the insertion this block is split up also, giving raise to an insertion in the higher level. If the highest level is to be split, a new level is added to the system.

The splitting of records is not done in two halves if the key to be inserted is larger than all the existing keys. This is done because on creation the keys should be given in ascending order filling up the whole available space. Of course this cannot be done simultaneously for all keys. It is done for the main key only.

7. Deleting records.

In the data file the record is marked as deleted and added to the list of deleted records.

In the key files, the key and its pointer are removed. If it is the highest key in the record, the indication in the higher level is adapted. This may be repeated, if it is the highest key on this level.

If the whole block becomes empty, the block is added to the list of deleted blocks. On the higher level the indication is removed.

If the record of the first level contains only one key and if there is more than one level, the number of levels is reduced by one.

8. The current key.

On sequential access, the read command reads the next record according to the key sequence of the current key.

When the file is opened, the current key is the main key (key 0). When issuing a *reset* or a *read_by_key* command the current key can be changed. At every moment in the program a sequential read may be used. It will take the record following the last record read, according to the current key. After the opening of the file or after a reset, the first key is taken.

9. Possible improvements.

- Portability.

The package has been written for the gnat compiler running in PC. The only changes that need to be made to make the package compatible with other systems and with any Ada83 compiler are the following ones :

  - instead of integers one should use its own integers defined as 32 bit integers.
  - The predefined packages used are now defined as child packages of the package Ada. For the use with Ada83 compilers the prefix “Ada.” should be removed in the context clauses from the packages “Ada.Direct_io” and “Ada.Unchecked_-conversion”. The package Ada.Integer_Text_io should be replaced by ones own instantiation of Text_io.Integer_io.

- Variable length records.

In the data file the logical records are now for fixed length records. It is not difficult to allow for variable length records. One has to keep track of the length of each record, which has to be registered together with the record itself. The routines that perform the linking of free space will get more complex.

- Block length.

The length of each physical block is now fixed at 512. This limits the length of the keys to 166. The block length can be increased or can be made a second generic parameter of the whole package.

- Key types.

The keys are now treated (by Ada.Unchecked_conversion) as strings. The comparisons are done as for strings. This means that e.g. integers will be compared byte per byte and this sequence does not correspond to the sequence of integers. To search for a given key does not cause any problem, but the sequence is not the expected one. A solution could be to add more generic parameters, together with a generic function for comparing keys. This is in conflict with the didactical simplicity we were looking for.

10. Conclusions.

The proposed package is a useful didactical instrument that can be used in non professional applications. It is sufficiently efficient and satisfies most of the criteria set forth at the start of the project.

The package must be considered as freeware. It is available for free, but copywriting the package or parts of it is not allowed.