

Btree.Read Subroutine

Description

There are times when you need direct access to indexed values. Much of the time, the system subroutine, [Btree.Extract](#) will perform this job quickly and efficiently, returning a sub-set of record keys that matches the required specifications. However, while [Btree.Extract](#) is powerful, versatile, and easy to use, it has one limitation:

- It returns only the record keys, not the index values.

You can use system subroutine Btree.Read to address these limitations. Btree.Read returns the complete index record that contains the searched-for value. Since each index record contains the record keys to the next and previous records in the index you use the record returned by Btree.Read as a start and follow pointers from there. See the example below.

Btree.Read reads a node of an indexed file (a **!** or **bang file**) into a dimensioned array. It can be called in the [Char event](#) of a window to return an entry in the **!** file (such as the next entry starting with a character or group of characters) without having to perform an index read, as a [Btree.Extract](#) call would. See the example below.

Syntax

Btree.Read (**bang_file**, **indexed_column_name**, **search_data**, **sort_order**, **node**, **pos**, **separator**, **found**)

Parameters

BTree.Read finds the starting position within the node. Once the starting position is found, you can traverse the index based on your search criteria. The Btree.read subroutine has the following parameters.

Parameter	Description
bang_file	Name of the indexed file (such as <i>!CUSTOMERS</i>) to search. This is the actual index file name, not the file variable.
indexed_column_name	Name of the indexed column (such as <i>CUSTOMER_NAME_XREF</i> or <i>CITY</i>) to search. Note that the bang file contains all the indexes for the referenced table.
search_data	The data to search for. This is a single value, not a range. If a partial word is passed, for example <i>SM</i> , a "starting with" search is implied. The format of the search data must match the format of the data stored in the index. If data is stored in internal format, you must convert it to its internal representation before passing it to Btree.Read.
sort_order	The order (' <i>AL</i> ' or ' <i>AR</i> ') in which to search, indicating either a left-justified or a right-justified (numeric) sort order in the index.

node	<p>A dimensioned array of 5 elements containing the node data returned by the read. Before passing, initialize to null. BTree.Read returns an index leaf record in node. The index leaf record structure is as follows:</p> <table> <tr> <th>Field Number</th><th>Description</th></tr> <tr> <td>1</td><td><i>Node Flag</i>. Contains a value of either 0, 1, or 2, where 2 denotes that this record contains index values and keys. The other values indicate that the record contains branch information and no actual index values.</td></tr> <tr> <td>2</td><td><i>Forward Pointer</i>. Contains the record key to the next record in the index, assuming ascending order.</td></tr> <tr> <td></td><td>The index record keys take the form of:</td></tr> <tr> <td></td><td><fieldname>*<optional-identifier>*<separator value></td></tr> <tr> <td></td><td>The optional identifier is used when there is more than one index leaf record that contains a single indexed value, common in large files with relatively few index values for a field, i.e., a status code field that can contain only a few possible values. The separator value is the same as the value returned by Btree.Read in separator.</td></tr> <tr> <td>If null, then this is the last leaf record in the index.</td><td></td></tr> <tr> <td>3</td><td><i>Backward Pointer</i>. Contains the record key to the previous record in the index, assuming ascending order.</td></tr> <tr> <td></td><td>If null, then this is the first leaf record in the index.</td></tr> <tr> <td>4</td><td><i>Indexed Values</i>. Contains a value-mark delimited list of index values, sorted either in AL or AR. The pos argument references a position in this list.</td></tr> <tr> <td>5</td><td><i>Record Keys</i>. Contains a value-mark delimited list of keys. Each value is associated with the index value in the same position as Field 4. If more than one key is associated with an index value, then the keys are delimited with sub-value marks.</td></tr> </table>	Field Number	Description	1	<i>Node Flag</i> . Contains a value of either 0, 1, or 2, where 2 denotes that this record contains index values and keys. The other values indicate that the record contains branch information and no actual index values.	2	<i>Forward Pointer</i> . Contains the record key to the next record in the index, assuming ascending order.		The index record keys take the form of:		<fieldname>*<optional-identifier>*<separator value>		The optional identifier is used when there is more than one index leaf record that contains a single indexed value, common in large files with relatively few index values for a field, i.e., a status code field that can contain only a few possible values. The separator value is the same as the value returned by Btree.Read in separator.	If null, then this is the last leaf record in the index.		3	<i>Backward Pointer</i> . Contains the record key to the previous record in the index, assuming ascending order.		If null, then this is the first leaf record in the index.	4	<i>Indexed Values</i> . Contains a value-mark delimited list of index values, sorted either in AL or AR. The pos argument references a position in this list.	5	<i>Record Keys</i> . Contains a value-mark delimited list of keys. Each value is associated with the index value in the same position as Field 4. If more than one key is associated with an index value, then the keys are delimited with sub-value marks.
Field Number	Description																						
1	<i>Node Flag</i> . Contains a value of either 0, 1, or 2, where 2 denotes that this record contains index values and keys. The other values indicate that the record contains branch information and no actual index values.																						
2	<i>Forward Pointer</i> . Contains the record key to the next record in the index, assuming ascending order.																						
	The index record keys take the form of:																						
	<fieldname>*<optional-identifier>*<separator value>																						
	The optional identifier is used when there is more than one index leaf record that contains a single indexed value, common in large files with relatively few index values for a field, i.e., a status code field that can contain only a few possible values. The separator value is the same as the value returned by Btree.Read in separator.																						
If null, then this is the last leaf record in the index.																							
3	<i>Backward Pointer</i> . Contains the record key to the previous record in the index, assuming ascending order.																						
	If null, then this is the first leaf record in the index.																						
4	<i>Indexed Values</i> . Contains a value-mark delimited list of index values, sorted either in AL or AR. The pos argument references a position in this list.																						
5	<i>Record Keys</i> . Contains a value-mark delimited list of keys. Each value is associated with the index value in the same position as Field 4. If more than one key is associated with an index value, then the keys are delimited with sub-value marks.																						
pos	After the call to Btree.Read, pos will contain the index into the value mark delimited arrays of values and associated record keys returned in the node array. The position of the row which is either an exact match or the next row, depending on the sort order. Set to 0 before the call.																						
separator	After the call to Btree.Read, separator will contain the value that all values in the record are less than or equal to.																						
found	After the call to Btree.Read, found will contain either True (1), indicating that pos points to an exact match on search_data, or False (0), indicating that pos points to the position in the index where the value would be located if it existed. Also, if false, and pos points to a value, this is the first value greater than search_data.																						

Multi-User Access

BTREE.READ does not perform any locking. If your index file is being updated by multiple users simultaneously these problems could result:

- The forward or backward pointer in record could point to a nonexistent record. The values in the index could change as you traverse the index. To prevent the index from changing as you traverse the index, you should lock the root node of the index (the *FieldName*ROOT* record in the indexing file) before accessing the index. Users will still be allowed to update the data file and to post new index transactions. Be sure to unlock the root when you're finished with the index. If you must allow updates while traversing the index use this strategy to account for a shifting index:

If your read of the next index record fails, reread the current record from the index. This will give you an updated version of the record with new pointers. If the new pointer fails (the read still doesn't return a record) call Btree.Read again with using the last index value in search_data. This will give you a new starting position within the index from which to start traversing.

Caution: *Both of these techniques could result in your program missing records or in your program processing the same record twice.*

See also

[Btree.Extract](#), [Extract_SI_Keys](#), [IXLOOKUP event](#), [Update_Index](#), [Collect.IXVals\(\)](#)

Example: Traversing the Index

```
/* This program searches the CUSTOMER_NAME_XREF index using BTREE.READ
and returns all records that have an indexed value between "CASH" and "THOMPSON". */
DECLARE SUBROUTINE BTREE.READ, FMSG
EQU TRUE$ TO 1
EQU FALSE$ TO 0
```

```

EQU NULL$ TO ""
DIM I_RECORD(5)
MAT I_RECORD = NULL$
I = NULL$
SEP = NULL$
FOUND = NULL$
I_FILEVAR = NULL$
FIELD = "CUSTOMER_NAME_XREF"
VALUE = "CASH"
MAX_VALUE = "THOMPSON"
/* Get the sort information from the index.*/
SORT_METHOD = XLATE("!CUSTOMERS", FIELD, 1, "X")
OPEN "!CUSTOMERS" TO INDEX_FILE ELSE
  FMSG()
return 1
END
OPEN "CUSTOMERS" TO CUSTOMER_FILE ELSE
  FMSG()
  RETURN 2
END
LOCKED = FALSE$
LOOP
  LOCK INDEX_FILE, FIELD:"*ROOT" THEN
    LOCKED = TRUE$
  END
  UNTIL LOCKED
REPEAT

BTREE.READ(INDEX_FILE, FIELD, VALUE, SORT_METHOD, MAT I_RECORD, I, SEP, FOUND)

/* Find the starting position in the key list in element five of the index leaf record. */
IF I > 1 THEN
  POS = INDEX(I_RECORD(5), @VM, I-1) + 1
END ELSE
  POS = 1
END

FLAG = NULL$
DONE = FALSE$
LOOP
/* Get the next key to process */
REMOVE @ID FROM I_RECORD(5) AT POS SETTING FLAG
READ @RECORD FROM CUSTOMER_FILE, @ID THEN
  GOSUB PROCESS_RECORD
END
IF FLAG = 3 THEN
/* REMOVE sets FLAG to 3 when a value mark is found.
   Finding a value mark indicates the indexed value has changed. */
  IF I_RECORD(4)<1,I> > MAX_VALUE THEN
/* No more valid values to process. */
    DONE = TRUE$
  END
END ELSE
  IF FLAG ELSE
/* If flag = 0 then we have reached the end of the current leaf record
   and it is time to read the next record or stop. */
    IF I_RECORD(2) THEN
/* There is a pointer to another record. */
      MATREAD I_RECORD FROM INDEX_FILE, I_RECORD(2) THEN
/* Start at the beginning of the list of keys in the new record.*/
        POS = 1
        I = 1
      END ELSE
/* Recovery logic goes here if you're not locking the index. */
        DONE = TRUE$
      END
    END ELSE
/* There are no more index records. */
      DONE = TRUE$
    END
  END
END

```

```

END
UNTIL DONE
REPEAT
UNLOCK INDEX_FILE, FIELD:"*ROOT" ELSE
    FSMMSG()
    return 4
END
return 0

PROCESS_RECORD:
/* Logic to process records goes here. */
RETURN

RETURN 0

```

Example: In the Char Event

```

/* The following code, in the Char event of an edit control called CITY,
returns the name of the city closest to the letter entered by the user, in the CITY control.
The program assumes that the CITY column in the CUSTOMERStable has a BTree index.*/
declare subroutine btree.read , set_property
dim node(5)
column = 'CITY'
entered_data = get_property( @window : '.CITY','TEXT')
open '!CUSTOMERS' TO vBangCustomers Then
    ReadV SortOrder from vBangCustomers, column, 1 Then
        pos = 0
        separator = ''
        found = 0
        call btree.read(vBangCustomers, column, entered_data, sortOrder, mat Node, Pos, separator, found)
        returned_data = node(4)<0,pos>
        returned_keys = node(5)<0,pos>
        set_property( @window : '.CITY', 'TEXT', returned_data)
    end
end
RETURN 0

```