jBASE Indexing

Contents

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
BOLD	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates JBASE commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates JBASE identifiers such as filenames, account names, schema names, and Windows NT filenames and pathnames.
UPPERCASE *Italic*	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, filenames, and pathnames.
Courier	Courier indicates examples of source code and system output.
Courier Bold	**Courier Bold** In examples, courier bold indicates characters that the user types or keys (for example, <Return>).
[]	Brackets enclose optional items. Do not type the brackets unless indicated.
{}	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
ItemA |.itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
. . .	Three periods indicate that more of the same type of item can optionally follow.
⇒	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File** ⇒ .**Exit**" means you should choose **File** from the menu bar, and then choose **Exit** from the File pull-down menu.

Syntax definitions and examples are indented for ease in reading.

All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.

Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

INTRODUCTION

In order to speed up retrieval of records by attributes rather than by key, jBASE provides the facility to build secondary indexes on these fields. These indexes are stored as binary trees (see Appendix A for details) which are compiled and stored in a file with the same name as the data file with "]I" appended onto the end. For example, an index created on JCUSTOMERS will be stored in a file called JCUSTOMERS]I.

Indexes will have a positive effect on SELECTs on the indexed fields themselves but the maintenance of the index will give a slight overhead on record updates as the index must be maintained. Thus a balance must be sought between improvements in query performance versus the overhead on other file access.

CREATE-INDEX

The CREATE-INDEX command will build a secondary index on an attribute or combination of attributes in the specified file.

COMMAND SYNTAX

```
create-index -Options filename indexname index-definition
```

SYNTAX ELEMENTS

Option	Description
-a	Adjourn the rebuild of the indexes until rebuild-index.
-c	Make the index case insensitive
-d	Debug of the index code allowed
-k	Slow coalesce mode
-lcode	Index lookup code
-n	Any indexes that produce null strings are ignored
-o	Overwrite any existing definition
-s	Write out a pseudo source files
-v	Verbose display, display on period for each 1000 records rebuilt
-m	Suppress multivalue index build use multivalues as single key
-w	Permanent write mode
-N	Synonymous with -n option. (compatibility option)
-S	Silent operation (compatibility option)
-M	Synonymous with -m option. (compatibility option)
-Vnn	Limit multivalue used in index key. e.g. V0 = M option
-X	Exclude existing records when building the index
-Zlocale	Create index in locale. Defaults to JBASE_LOCALE if none specified

This command can be used to create a new index definition. By default the index is then rebuilt to include any existing records.

Option -c means the indexes created will be done in a case insensitive fashion. For example "Fred" and "FRED" will be the same index. This is used automatically in the key-select or query-index command. However if a jQL command such as SORT or SELECT wants to use the index, then the command must be

done in such a way that the jQL command would also be case insensitive (for example, attribute 7 of the DICT item is MCU and the selection criteria is all upper case).

Option -d means the pseudo-code created to build the index key can be debugged. This assumes that the debugger is enabled for the rest of the jBC code anyway.

Option –k and –w are advanced tuning options see Appendix A for a description of their use.

Option -l is the lookup code. It is used with key-select and query- index. What happens is that the selection criteria will be converted using an ICONV call before being used. For example if you create a right justified (numeric) index on say attribute 6 of all items, this could be a date field in internal format. If you want to look at a range of dates then instead of doing this:

jsh-> key-select ORDERS WITH PLACE-DATE > 10638

where 10638 is a date in internal format, then by using the option "-lD" we will perform an ICONV on the selection criteria of format "D" thus translating the date in external format to internal format, and so your command line would be:

jsh->key-select ORDERS WITH PLACE-DATE > 14-feb-1997

This also applies to selection criteria passed with a jQL command such as LIST or SELECT.

Option -n shows that any index keys that are created as zero length strings will not be entered into the index. This is useful for suppressing unwanted index keys. It is especially useful when used in conjunction with the CALL statement in the index definition (see Appendix A) so that if the subroutine decides the index is not interested in being stored, it creates a null index key.

Option -o will overwrite any existing index definition. Without this, if you attempt to re-define an index definition you will get an error. If the -o option is used to re-define an existing index definition, then all the index data for the definition previously associated with the index will be deleted.

Option -a means the index data will not be built once the index definition is created. The default action is to build the index, depending upon the size of your file data, this could be a lengthy operation ! Once the index data is built it then becomes in-sync with the file data and is available to all the index commands such as key-select and available to jQL commands such as SELECT to improve the performance of them. With this option you need to execute a rebuild-index command to rebuild the index data.

Option -s causes some pseudo source code to be created. This is used with option -d so that you can debug complex index definitions.

Option -v is the verbose mode. It will display a period character for every 1000 records in the file that are rebuilt.

EXAMPLES

Example: Create an index based on attribute 1 , concatenated with attribute 2.
jsh--> create-index filename indexname by 1 : 2

Example: Create an index on the attribute given in the DICTionary definition NAME and a second attribute number 3 , but in descending order
jsh--> create-index filename indexname by NAME by-dl 3

Example: Create an index on attribute 4 which is normally an internal date. You want to be able to specify dates in external format when doing selections against it. Additionally if the field is a null string, then don't store any index information.
% CREATE-INDEX -lD -n ORDERS BY-AR 4

Example: Create an index using three attributes.
sh-> create-index FILENAME INDEXNAME BY 1 BY 2 : "*" : 3

In the above definition the index key is built out of three attributes. Should these attributes all have differing numbers of multi-values it makes it difficult to create an index key that is logically consistent. Therefore in the above example it would fail to create the index definition.

Option -N is synonymous with the -n option on create-index. When used, any index keys that equate to a null string will not be stored.This is a compatibility option.

Option -S is a compatibility option which provides for silent operation when an index is created.

Option -M or -m option suppresses creating individual index keys for each mutlivalue, in other words all mutlivalues are used to create the index key.

For example, if an index is generated on a record with an attribute as follows:

PIPE
001 123]456]789

Then by default three index values based on "123" , "456" and "789" will be created. With the -m or (M) option on create-index, we will build a single index value based on "123]456]789".

Option -Vn. This option provides compatibility and is used to limit the number of multivalues used to generate an index key. Without this option then ALL multi-values will each generate an index definition. This option restricts it to the first nnn values. A special case of (V0) exists. In this case where the multi-value count is set to 0, we assume no multi-values are required and so we don"t split the attribute into multi-values but treat the entire attribute as a single entity -- in effect then (V0) option is identical to the (M) option.

Remember the jBASE syntax already allows an individual value to be used instead. For example

CREATE-INDEX FILENAME INDEXNAME BY 4.3

means just use the third multi-value in attribute 4.

Option -X. This option on CREATE-INDEX will set-up the index, but not run the existing file through it - in other words, it doesn"t make any attempt to index what is already in the file. The file will still me marked as "in-sync". The net result is that you get an index later with only newly-written or modified records - very nice when you're dealing with huge files and you only want to process what's changed or created since the index was set-up.

In addition to the above syntax a compatible form of the CREATE-INDEX command can also be used.

CREATE-INDEX FILENAME ITEM

This creates an index called ITEM and the index definition is based on the dictionary item ITEM.

jBASE supports this by converting on-line the syntax to the jBASE syntax and notifying the user of the equivalent converted command (unless the (S) option is used).

When this happens the dictionary definition is used as follows:

Attribute 2 tells us what attribute number to extract
Attribute 7 tells us any lookup code i.e. what to convert any matching string using.
Attribute 8 tells us any conversions to apply when building the index data
Attribute 9 tells us if it is a left or right (numeric) sort.

jBASE allows indexes created in this manner to be used with some jQL commands like SELECT or SORT. An index which is not created via a dictionary item must query the index with KEY-SELECT or QUERY-INDEX.

If a complex definition exists in attribute 8, then the conversion will fail and the user will have to use the jBASE syntax.

This example shows a DICT item in jBASE and how , if you run the create-index command against it, it will be converted to the jBASE syntax and run.

```
INDEX1
001 A
002 3
003 Description
004
005
006
7    D2
8    MCU
9    R
010 10
```

For example
CREATE-INDEX -a filename INDEX1
Notice: Command converted to "CREATE-INDEX -lD2 filename BY-AR OCONV(3,"MCU")"
Index definition "INDEX1" created successfully
Warning: You now need to rebuild the index data for file "filename" using rebuild-index.

DELETE-INDEX

This command is called to delete one or more index definitions that are associated with a file. All the space taken by the index is released to the file overflow space (but not necessarily the operating system file free space).

COMMAND SYNTAX

delete-index -Options filename { {indexname {indexname ...}} | *}

SYNTAX ELEMENTS

Option	Description
-a	Delete ALL indexes in the file
-S	Silent option for compatibility

Option -a causes all index definitions to be deleted. Note: For J4 files this does not physically delete the index file. If you have no further need for indexes then the filename]I can safely be deleted.

Without option -a you need to specify one or more index name on the command line.

EXAMPLES

Example: Delete ALL the index definitions for file PRODUCTS

jsh--> DELETE-INDEX -a PRODUCTS
jsh--> DELETE-INDEX PRODUCTS *

Example: Delete the index name "value" in the file PRODUCTS

% delete-index PRODUCTS value

KEY-SELECT / QUERY-INDEX

This command allows you to select or count a list of record keys. Note that file updates that cause a change to an index will wait for a query-index to complete. The first structure of the query-index command allows you to select all record keys sorted by the index definition. For example, to select all customers sorted by their lastname:

jsh-> query-index CUSTOMERS lastname

COMMAND SYNTAX

query-index -Options filename index_name
query-index -Options filename {IF/WITH} iname {Op} "startpos"
query-index -Options filename {IF/WITH} iname {Op} "startpos" {AND} {Op} "endpos"

SYNTAX ELEMENTS

Where Op can be: LT < less than
LE <= less than or equal
GT > greater than
GE >= greater than or equal.

Option	Description
-c	COUNT the records (default is to select records)
-inn	Indexes used restricted to approx nn indexes
-mTYPE	Match algorithm – either "REGEXP", "JQL" or "DEFAULT"
-rnn	Record count restricted to approx nn records

The second structure of the query-index command allows you to specify a single conditional parameter. You can make this query less than, greater than etc. to the parameter. If you don't specify LT,GT,etc. then it defaults to equals.

EXAMPLES

Select all customers whose name begins with "KOOP"

jsh-> QUERY-INDEX CUSTOMERS IF lastname "KOOP"

Note that in this case the double quotes will be ignored , as would single quotes. The IF token is a throwaway token, and is used simply for clarity. WITH can also be used to the same effect.

Another example is to select all customers whose date of birth is before 25 July 1956

jsh-> QUERY-INDEX CUSTOMERS WITH dob LT 25-JUL-1956

The third structure of the query-index command allows you to specify a range of values. This means the operators must be either GT or GE followed by LT or LE. If the operators are not specified the command defaults to GE and LE.

Example: Count all the customers whose last order was placed between 19-DEC-1996 and 23-DEC-1996 inclusively.

jsh--> query-index -c CUSTOMERS IF order.date >= "19-DEC-1996" and <= "23-DEC-1996"

Option -c means a count of record keys is done instead of producing a select list.

Option -m allows the pattern matching algorithm to be changed :

When the command is performing its test, it does so using a simple string or numeric test. If you want to make the test more akin to say a jQL command, then you would use the -mJQL option. For example, if you want to select all customers whose name ends in PATEL, you could do this:

jsh-> query-index -mJQL CUSTOMERS IF lastname EQ "[PATEL"

Similarly you can use the -mREGEXP to use a pattern matching algorithm called "Regular Expressions". This allows complicated patterns to be searched for. See the "Regular Expressions" chapter in this document. As an example, the following command will select all products whose description begins with the letter A is followed by any number of characters before the sequence PIPE is found:

jsh-> query-index -mREGEXP PRODUCTS IF description EQ "^A.*PIPE"

Option -i can be used to restrict the number of indexes used to create the list of record keys. This can be useful to restrict a search to a smaller subset.

Option -r is similar to -i except it restrict the number of record keys.

NOTES

QUERY-INDEX should only be used to generate select-lists for READNEXT KEY and READPREV KEY statements. It should not be used to generate select-lists for a subsequent SELECT, the READNEXT statement or for use with other jQL commands, e.g. LIST; the jQL SELECT command should be used for this purpose.

LIST-INDEX

This command is used to display to the screen details of all the current definitions. A format similar to jQL is produced.

COMMAND SYNTAX

list-index -Options filename {Same as jQL options}

SYNTAX ELEMENTS

Option	Description
-ffile	Use the filename "file" instead of a temporary file
-m	Output is in machine readable format
-a	Verbose output – all fields will be displayed

Option -f allows you to specify your own file name instead of list-index creating a temporary file. This way you can find out what DICTionary items are created by list-index, and if you want to you can modify them and pass them on the command line. Using this option therefore allows you to define your own output format for the command.

Option -m produce "machine readable" displays. In other words the detail is displayed simply as a series of lines, one line per index definition, with a tab character (CHAR(9)) to delimit the fields. This makes it easily parsed by another program or UNIX script.

Option -a is for verbose mode -- all details will be printed instead of a smaller selection.

EXAMPLES

Example: Display all the index definitions, in full, for file CUSTOMERS, and send the output to the printer.

jsh--> LIST-INDEX -v CUSTOMERS (P

NOTES

Machine readable format broken down as follows;

Tab position	Description / INDICES() attribute reference
1	Index name <0>
2	Not used
3	Sort criteria <1>
4	Base FID of index
5	Always 1
6	Index type <2>
7	Creation time <5>
8	Last modified time <6>
9	Index number (zero based)
10	DEBUG flag <7>
11	NULL flag <8>
12	CASE flag <9>
13	MULTI-VALUE flag <10>
14	SYNC flag <11>
15	VALUE MAX count <12>
16 – 19	Not used
20	INDEX definition <3>
21	Lookup definition <4>

REBUILD-INDEX

This command will rebuild the index definitions. It can be used in the following circumstances:

- Following the corruption of the index file
- Following a file restore using jrestore . This is detailed later.
- Following the creation of an index using create-index -a.

By default create-index will build the index and will not require a rebuild-index command to be performed.

COMMAND SYNTAX

rebuild-index -Options filename { {indexname {indexname ...}} | *}

SYNTAX ELEMENTS

Option	Description
-a	Rebuilds ALL indexes in the file
-r	Rebuild ALL indexes in the specified directory
-v	Verbose mode; displays a full stop for every 1000 records

Option -a means you want to rebuild all the indexes defined for the file. This can also be achieved by specifying * as the index name. Otherwise you must specify on the command line one or more index names to rebuild.

Option -r will rebuild all files in the directory name specified. This is a useful operation after using, for example, jrestore to restore your database and then you can use the option -r to rebuild all files in a certain directory.

VERIFY-INDEX

This command will verify the integrity of an index definition, in so far as it looks for internal corruption. It doesn't actually verify that the index data actually correctly cross references the data file records.

COMMAND SYNTAX

verify-index -Options filename { {indexname {indexname ...}} | *}

SYNTAX ELEMENTS

Option	Description
-a	Verifies ALL indexes in the file
-l	Display leaf information
-o	Display overflow frame information
-r	Display record information
-v	Verbose mode

Option -a means all indexes will be verified and this can also be achieved by using * on the command line for the index name. Without the -a option (or * as index name) you must specify on the command line one or more indexes to verify.

Option -l means information about each leaf of the index is displayed.

Option -o causes the overflow table information to be displayed.

Option -r causes all the record information to be displayed. This is the index key followed by all the record keys that share the same index value.

Option -v is the verbose mode and causes extra information to be displayed.

NOTES

CAUTION While this command is active a lock on an entire index is taken. Should an application try to update that index then the application will wait until the lock is released, which isn't until the verify-index command has completed for that particular index. This means scenarios such as the one below should be used only with caution as the piping of the output into "more" means the lock will be retained until the display has completed.

% verify-index -avrl FILENAME | more

Using jQL Commands

jBASE supports a limited mechanism whereby jQL commands such as SORT or SELECT can automatically use any valid secondary index to reduce the search time. This does not involve creating a specific DICTionary item. If for any reason the index cannot be found, or is not up to date (e.g. awaiting a rebuild-index command) then the jQL command will ignore the secondary index and retrieve the information in the usual manner.

At present only a single search pattern can be used in a jQL command. As an example a file will have an index built onto attribute 23 , the customer last name like this:

```
jsh->create-index CUSTOMERS lastname BY  23
```

Let us assume there exists a DICTionary definition called LASTNAME that looks like this:

```
LASTNAME
001 A
002 23
003 Customer  Lastname
004
005
006
007
008
009 T
010 20
```

Now let us assume we try to select all customers in that file whose last name equals "COOPER]". The jQL statement would look like this:

```
jsh-> SELECT CUSTOMERS IF LASTNAME EQ  "COOPER]"
```

In this example the index definition is "out of sync", awaiting a rebuild-index command to be performed. Therefore the SELECT would achieve the result by looking through the entire file. Now let us run the rebuild-index command as:

```
jsh-> rebuild-index CUSTOMERS  lastname
```

If we now re-execute the SELECT command, then instead of scanning through the entire CUSTOMERS file, it will look through the index definition "lastname" instead and will therefore execute considerably quicker.

Subroutines

It is possible to call a jBC subroutine from an index definition. The subroutine should have five parameters as follows:

Result	This is to return the result of the calculation.
Filevar	File variable of file for which the update is being processed.
Record	Record beingupdated.
Key	Record key of the record being updated.
Field	Field, or attribute, already extracted as part of the index definition

As an example, consider the following index creation

```
jsh-> CREATE-INDEX FILENAME INDEXNAME BY 1 : CALL(2,"INDEX-DEF")
```

When an update occurs the index key is calculated by taking attribute 1 and concatenating it with the output from a call to a subroutine called INDEX-DEF. The source code for this may look something like this:

```
INDEX-DEF
1   SUBROUTINE INDEX-DEF(result , file , record , key , field )
2   IF NUM(field) THEN result = "*1" ELSE result = "*0"
3   result := record<3>
4   RETURN
```

In the above example the result is created in the first parameter, the "result" variable. This is calculated by taking the string "*1" or "*0" and concatenating it with attribute 3 from the record being updated. The choice of "*1" or "*0" depends upon whether the extracted attribute, passed in the fifth parameter as variable "field" , is numeric or not. The index definition was "CALL(2,"INDEX-DEF")" so this extracted attribute will be attribute 2.

Any normal jBC code will execute in these subroutines but you should be aware of the following pitfalls.

The code should always create exactly the same result given the same record. This means you should avoid using functionality that creates a variable value, such as the RND() function, the TIME() or DATE() functions , the users port number and so on. If this occurs then there will be no way jBASE can delete a changed index value and so the index file will continually grow with invalid data even if the number of records remain constant.

These subroutines will be called implicitly from other running jBC code which to its knowledge has merely executed a DELETE or WRITE statement. You should therefore avoid any code that changes the nature of the environment such as using the default select

list, turning echo on or off, turning the break key on or off. There are ways around many of these, for example you can turn the echo on and off so long as your code remembers in all cases to restore it to its original status. Similarly you can do a SELECT so long as it is to a local variable rather than the default select list.

Depending upon your application, these subroutines may be accessed by users other than the account in which the files exist. Therefore all persons who have access to OPEN and update the file must also have access to be able to CALL your subroutine. This can be done in a number of ways.

All users who want to update the file may have the environment variable JBCOBJECTLIST set to include the library where these subroutines were catologed into. For example if the subroutines have been cataloged from account greg then you can set up the JBCOBJECTLIST as follows so that we look in the users current lib directory and failing that in the lib directory for greg (this from the Korn shell)

```
export  JBCOBJECTLIST=$HOME/lib:~greg/lib
```

You can CATALOG into a directory that is common to all users anyway. Such as directory is the lib directory where jBASE is installed. In this example you don't need to set up JBCOBJECTLIST. You do , however, remember to re-catalog all these subroutines when a new version of jBASE is loaded. You change the output directory for CATALOG with the JBCDEV_LIB environment variable. For example, from the Unix Korn shell you would do this:

```
export  JBCDEV_LIB=$JBCRELEASEDIR/lib
CATALOG BP INDEX-DEF
```

Related BASIC Statements

INDICIES	Returns the index information for the specified file.
OPENINDEX	Opens the specified index definition..
READNEXT	Move forward through an index.
READPREV	Move backward through an index.
SELECT	Select an index variable.
SELECTINDEX	Creates an array of record keys based on the specified index.

Documentation for these statements can be found in the BASIC statements and functions manual.

Regular Expressions

Regular expressions are the name given to a set of pattern matching characters. The term derived from Unix environment. The regular expressions can be used to great effect using the query-index command to decide what records to select. A full description of regular expressions can be obtained on Unix systems by entering the command:

```
% man 8 regexp
```

For Windows systems only a limited subset of regular expressions are available. The following characters inside a regular expression have special meaning:

^	The text must match the start of the record key.
$	The text must match the end of the record key
.*	Any number of characters of any value may follow
\x	This escapes the character x meaning it just evaluates to x. This is useful if you want to include say the ^ character as part of your text string rather than a character with special meaning.

For example, on either a Unix or Windows/NT system you could use key-select to find a product description that has the text SLIPPER at the start of the description. This can be done using the jQL format as you might be familiar with using say the SELECT command or by using regular expressions. The two methods are therefore:

```
jsh-> key-select -mJQL PRODUCTS IF description EQ "SLIPPER]"
jsh-> key-select -mREGEXP PRODUCTS if description EQ "^SLIPPER"
```

As a more complicate regular expression, the following example looks for a product that begins with the string BIG , has the words RED somewhere in the text, and then must end with the words ENGINE:

```
jsh-> query-index -mREGEXP PRODUCTS "^BIG.*RED.*ENGINE$"
```

The Unix implementation uses the operating system supplied version of regular expressions and these are far more powerful than the jBASE supplied version of regular expressions on Windows systems. As already mentioned , use man 5 regexp to find more details. The following example looks for a product description that begins with the words PIPE , any number of spaces, then one or more numeric characters follow (including optionally a decimal point), any number of spaces, and finally the characters "mm" , which are case insensitive:

```
jsh->query-index -mREGEXP PRODUCTS EQ "^PIPE *[0-9\.][0-9\.]*[mM][mM]$"
```

The above command would find any of the following:

PIPE 5 mm
PIPE15MM
PIPE 33.3 mm

Backup and Restore

There are three ways of backing up your database.

ACCOUNT-SAVE

This should only be used to transfer your data to a non-jBASE system. The indexing information will be lost.

tar and cpio

These Unix supplied backup utilities will backup ALL jBASE files and not care if they are hashed files, index files, dictionary items and so on. These should be used with caution, and only against databases where there is no update activity. The tar and cpio programs will not respect the locks on a file that jBASE normally uses.

jbackup and jrestore

These are the jBASE supplied backup and restore programs and are the preferred way of backing up your data. They respect all locks and will keep the tapes free from any soft format errors that would otherwise occur if using tar and cpio on an active database.

By default , when you use jbackup to save a database, any indexing information will be saved as just the index definition, not the actual index data. Conversely during a restore using jrestore, the index definition will be restored, but not the index data, whether it exists on tape or not.

So the full scenario, by default

(i) Backup your database:

```
cd $HOME
find . -print | jbackup -v -f/dev/rmt/0
```

(ii) Restore your database:

```
cd $HOME
rm -rf *
jrestore -v -f/dev/rmt/0
```

(iii) Rebuild indexes. The stage (ii) will have restored the database and the index definitions, but not the index data. Now to rebuild the index data for all the files you have just restored:

```
rebuild-index -r $HOME
```

If you have restored files into sub-directories you could use the following Unix command to rebuild the indexes for all files in all sub-directories

cd $HOME

```
find . -print | xargs rebuild-index
```

When you backup a database with jbackup, you can use the "-c" option and this will backup the actual index data as well as the data files. The index data will be dumped as a normal Unix binary file , and so during the restore phase will be restored exactly as-is.

When you restore a database using jrestore, by default it will restore the index information, but will NOT rebuild the index data. This is quite time-consuming and so by default the jrestore works in the quickest mode. The index will need to be re-built at a later stage using rebuild-index. In the meantime, any attempts to use query-index or key-select will fail, and the jQL programs such as COUNT and SELECT will not use the index in order to satisfy the request. During the restore a warning message will be shown against each file that needs re-building. Once the rebuild-index has completed, the index will become available again to the jQL programs and query-index.

You can optionally use the -N option to jrestore and this will result in the indexes being built as the data is restored. This will slow down the restore time, but means the index is immediately available for use without the need to re-build the indexes with rebuild-index.

Appendix A. Internal operation

The following section describes the internal operation of the indexing system. Understanding of this is not a requirement for simple index creation or for indexing of small to medium sized files. It explains 'tuning' operations that can be carried out to improve indexing performance with large (>100K records) files.

Within the filename]I file each index is stored as a b-tree. By default each node of the tree relates to one 'index value' and contains a list of keys for the records that contain that value. This is efficient for read operation because, for any requested index value, the required node can rapidly be found and the record list returned.

For write operations this is however inefficient. If a given value points to a large list of records then to insert a new record the correct place in the list must be found, the remainder of the list shifted up and the new 'record key' inserted. If a file containing a large number of records is indexed this can be a very slow operation.

To solve this problem 'write mode' was introduced. In this mode instead of storing record keys in simple lists each list is also stored as a b-trees rooted in a node of the existing index value tree. These 'record b-trees' are searched by record key instead of index value. The result is that once the correct index value has been found a new record key can quickly be inserted. However when a read is required the record b-tree must be coalesced back into a list. This is time consuming. So read performance is reduced.

By default when create-index (or rebuild-index) is invoked the system is switched into write mode. All the records are written into the index, the system is switched back into 'read mode' and the record b-trees are coalesced back into lists ready to be read Create-index has some options to fine tune this behaviour:-

Option –k means 'Slow coalesce' mode. In this mode the re-coalescing of b-trees is deferred until each index value is read for the first time. This will give a slightly faster rebuild but at the cost of a delay on the first read of each value.

Option –w means 'Write only' mode. In this mode b-trees are never coalesced into lists. This will reduce the performance of read operations but increase the performance of write operations. If an indexed file is frequently modified, but only occasionally searched, then this mode should be considered.

Note On IndexFile Size

A certain amount of extra file space is required to support record b-trees. Because of this when in write mode a slightly larger filename]I file is required than is the case in read mode. When the system switches

back into read mode less space is required but, since the size of the filename]I file cannot be reduced, it remains the same size. The ‘unused’ space remains available to the indexing system and is re-used as more entries are added to the file and/or new indexes are created.

Appendix B – Sort sequences

When you create an index definition using the create-index command you must specify a description of how the index key is to be built. This example below shows a basic index definition such that the index key is created from attribute 4 of the record

```
create-index CUSTOMERS CUST-NAME BY  4
```

In the above example CUSTOMERS is the name of the file and CUST-NAME is the name of the index definition. The full details of create-index have already been described at the earlier part of this document.

This appendix deals with the description of how to build the index key from the record data, which is "BY 4" in the above example. The syntax of the index definition can be summaries as:

by-expr sort-def {by-expr sort-def {by-expr sort-def }}

Where **by-expr** is one of BY , BY-AL , BY-DL , BY-AR or BY-DR and sort-def will be described later. Each **by-expr** causes another sort sequence in the index key. Consider the following example:

```
create-index filename indexname BY 3 BY  2
```

The means the index key will be made up from attribute 3 of the record , a 0xff delimiter, and finally attribute 2 of the record. The index data will be sorted by attribute 3. Where there are cases when attribute 3 is the same as previous index keys, it will be further sorted by attribute 2.

The keywords by,by-al,by-dl,by-ar and by-dr are all case insensitive.

The keyword BY means the sort sequence is ascending left justified.

The keyword BY-AL same as BY and means the sort sequence is ascending left justified.

The keyword BY-DL means the sort sequence is descending left justified.

The keyword BY-AR means the sort sequence is ascending right justified.

The keyword BY-DR means the sort sequence is descending right justified.

When a sort sequence is described as right justified then this takes on the same meaning as it does in jQL commands such as LIST and SELECT, i.e. it assumes numeric data. Therefore right justified fields will be sorted numerically and left justified fields sorted textually. In the event that a right justified field contains a null string or non-numeric

data, then the sort sequence will be the same as jQL command and have the following precedence:

- Fields that contain entirely numeric values
- Fields that are an empty zero length string
- Fields that begin with a numeric value and then are followed by non-numeric
- Fields that are one or more bytes and are not numeric.

The "sort-def" definition mentioned earlier is the description of how an individual sort sequence is defined. The "sort-def" can be one or more extract definitions , each definition being delimited by a colon. Each sort-def definition is concatenated to each other. For example:

```
by 3 : 4 : 5 by-dr DATE
```

means the index key is made up as follows:

- attribute 3 from the record concatenated with attribute 4 from the record concatenated with attribute 5 from the record concatenated with a 0xff delimiter between the sort sequence
- attribute NN from the record, where NN is described the in DICTionary item DATE.

When the index key is added to the index data, the key will be sorted , in ascending left justified sequence, by the first sort sequence , attributes 3 , 4 and 5. If there are duplicate keys then it is further sorted in descending right justified sequence by attribute NN from the record where NN is described in the DICTionary item.

Note that when using DICTionary items to describe what attribute number to extract the index definition simply extracts the attribute number and forgets the DICTionary name. This way the index remains logically consistent even if the DICT record is later amended. The "sort-def" definition can be more than a simple attribute number. It can be any of the following types.

Numeric This shows to simply extract attribute "Numeric" from the record. If the numeric value is 0 then this means the record key.

Numeric.Numeric This shows to extract a specific multi-value from a specific attribute. For example 4.3 shows to extract multi-value 3 from attribute 4.

OCONV(field , "code") This causes an output conversion to be applied to the defined "field" The conversion to apply is any that can be applied with the OCONV function in jBC code. The "field" is defined using one of the above mentioned definitions of "Numeric" or "Numeric.Numeric".

CALL(field,"subname") This allows a normal jBC subroutine to be called. The mechanism for doing this is defined later.

"STRING" You can specify a string constant.

As a more complex example, consider the following statement:

```
jsh-> CREATE-INDEX CUSTOMERS BY 3.1 : "*" : 2.1 BY-AR  4
```

The above creates an index key with two sort sequences. The first sort sequence is in ascending left justified with multivalue 1 of attribute 3 concatenated with an asterisk and then concatenated with multivalue 1 of attribute 2. The second sort sequence is an ascending right (e.g. numeric) sequence on the fourth attribute. This index could be, for example, a customer's surname , and asterisk and a customer's first name. In the event youhave two customers with the same name it will be further sorted by their date of birth.

The second example shows the use of the CALL function:

```
jsh-> create-index -lD -n CUSTOMERS  by-ar
CALL(3,"TESTDATE")
```

In the above example attribute 3 will be extracted from the record and passed to a user-written subroutine call TESTDATE. This subroutine can amend the index key from what it was passed (e.g. attribute 3) to whatever it likes. We will assume the purpose of this subroutine is to test the attribute passed to it, and if it was a numeric, assume it is a date field and check the date is inside a range. If the date is okay, it passes back attribute 3 . If not, it passes back a null string.

The use of the **-n** option means any null index keys will not be stored in the index data. Hence when used in conjunction with the subroutine, it is a way of using a user-written subroutine to decide what records to put in the index data.

The use of **by-ar** further shows that all index keys (in this case non- null keys) will be stored in ascending right justified fashion, e.g. as a numeric sort.

The use of the **-lD** option shows that when an enquiry is made of the index via query-index , key-select or the jQL commands such as LIST or SELECT, then the string to search for will first of all be converted using an input conversion of type "D". This is primarily used so that these commands can use date (or times) in external format (e.g. 23-jul-1970) but will be compared numerically as internal values against the index data.

Appendix C – Universal Co-ordinated Time

A standard time and date format exists on many operating systems including Unix and Windows. This is called Universal Co-ordinated Time and is the number of seconds since 00:00:00 January 1st 1970. Confusingly it appears to have been given the acronym UTC (instead of UCT as you might expect).

Within a jBASE application the normal convention adopted by jBASE , for the purposes of legacy code compatibility, is that times are stored as seconds past 00:00:00 and dates stored as the number of days since 31st December 1967.

The way to convert from UTC to jBASE time and dates and vice-versa can be demonstrated by the following code segment

```
*
* Convert a UTC value to a displayable time and  date
*
UTC = 866558733
internal.date = INT(UTC/86400)+732
internal.time = MOD(UTC,86400)
CRT "Date = ":OCONV(internal.date,"D")
CRT "Time =  ":OCONV(internal.time,"MTS")
*
* Convert internal time and date to  UTC
*
UTC2 = (DATE()-732)*86400 +  TIME()
CRT "UTC2 = ":UTC2
```

One important aspect to remember is that the UTC is often in the base time of the operating system without any time zone applied. For example on Unix systems you set the time and date of your system in UTC but then individual accounts may have different time zones applied. Thus if you create an index at what appears to be a time of say 10:40:29 then this could actually be a time of 11:40:29 but with a time zone of minus one hour applied.