it's about time

Technical Whitepaper

Working with sym files

Date March 2019

Author Paula Clarke is a kdb+ consultant for Kx who has worked for some the world's largest financial institutions. Currently based in New York, she is working for a major investment bank where she is part of a team responsible for designing, developing and maintaining a reporting and global tick-capture kdb+ system.

Contents

Working with sym files

As kdb+ developers we often find ourselves working with some of the largest and most important real-time and historical data sets in the world. We capture data, parse it, analyze and store it. Managing such data sets can involve consuming feeds from multiple sources in parallel, performing in-memory analysis, and storing data on disk for future reference.

Within a historic kdb+ architecture the sym file is literally a key component, hence it should be treated with care and consideration when performing database management. Correct procedure when creating and modifying a historical database (HDB) should be followed to ensure that this file does not become corrupt or bloated.

In this whitepaper, we discuss what the sym file is, suggestions for table setup, the maintenance of the sym file and safety measures that can be taken to protect the sym file.

All tests were run using kdb+ version 3.6 (2018.05.17). Code sections that have a distinction between versions earlier than 3.6 have been included in the Appendix.

Enumeration

In kdb+ an enumerated type is a way of associating a series of symbols with a corresponding set of integral values. The specific set of symbols is called the *domain* of the enumerated type, and this domain name identifies the enumeration.

The idea behind an enumeration is that only distinct names are stored, with every instance of the symbol then being referenced from its position within the string pool by an integer value (the symbol's index). This process of converting a list of symbols to the equivalent list of indices is called enumeration in q.

There are two methods of enumeration: `$` (Enum) and `?` (Enum Extend).

Given a list of tickers `p`, we can perform an enumeration using the Enum operator as follows:

```
q)p:`AAPL`AAPL`GE`IBM`GE`IBM`GE`GOOG`GE`IBM`AAPL
q)x:`u#distinct p / apply the unique attribute
q)e:`x$p
q)e
`x$`AAPL`AAPL`GE`IBM`GE`IBM`GE`GOOG`GE`IBM`AAPL
```

The domain of `e` is `` `x ``. When working with a HDB created with `.Q.en` or `.Q.dpft`, `` `sym `` is the domain of the sym file. (`.Q.en` and `.Q.dpft` are discussed in the next section).

Values can now be stored on disk as a list of integers, greatly reducing the amount of space required.

```
q)"i"$e
0 0 1 2 1 2 1 3 1 2 0i
q)type e
20h
```

Storing the data on disk as integers (mapped to symbols via enumeration) allows queries that use comparison functions i.e. `in`, `=`, `<>` etc. to execute more efficiently. This is because it is quicker for q to perform integer comparisons than string comparisons. In this way enumeration normalizes the data, making it more compact and, in turn, lookup times much faster. It also saves memory by using fixed-size integers multiple times rather than strings, which can have varying sizes.

When using `$` (Enum), the entire domain of the enumeration must be in `x`; otherwise, a *cast* error will be signaled. To expand the domain, use `?` (Enum Extend) to fill in any missing items in `x`.

```
q)p,:`FB
q)e:`x$p
'cast
q)x
`AAPL`GE`IBM`GOOG
q)e:`x?p /fills in the missing value
q)e
`x$`AAPL`AAPL`GE`IBM`GE`IBM`GE`GOOG`GE`IBM`AAPL`FB
q)x
`u\#`AAPL`GE`IBM`GOOG`FB
```

Enums and linked columns now use 64-bit indexes, and because of this, there are a number of points to note if upgrading a system to V3.6+ from an earlier version of kdb+:

- 64-bit enums are type 20 regardless of their domain.
- There is no practical limit to the number of 64-bit enum domains.
- 64-bit enums save to a new file format, which is not readable by previous versions.
- 32-bit enums files from previous versions are still readable; use type space 21 through 76 and all ops cast them to 64-bit.

Persisting tables on disk and the sym file

Kdb+ enforces enumeration on a table being splayed to disk if it has a column of type symbol. Attempting to splay a table with columns of type symbol without enumerating first will result in a type error.

When saving tables to disk, enumerations can be done manually, however, this can be cumbersome to maintain. Fortunately, there are functions within the `.Q` namespace that will do the enumeration for us: `.Q.en` and `.Q.ens`.

.Q.en

`.Q.en` will enumerate to the sym domain by default. The arguments for `.Q.en` are the directory path that the sym file will be stored in and the table itself.

After running `.Q.en` on our table, both the sym file on disk and sym variable in memory are created.

```
q)t:([]col1:`AAPL`IBM`GE`GOOG;col2:4?100.;col3:(string 4?`2))
q)`:db/t/ set .Q.en[`:db] t
`:db/t/
q)system"ls ~/db"
"sym"
,"t"
```

The sym file is the file that contains a string pool in which the distinct symbol values are stored.

```
q)get `:db/sym
`AAPL`IBM`GE`GOOG
// be wary of running get if you have a large sym file
```

By placing the sym file in the database root directory, it ensures the file will be loaded into memory automatically when the database is loaded in a process, as shown below.

```
q)\l db
q)\a
,`t
q)sym
`AAPL`IBM`GE`GOOG
```

Backing up the sym file after .Q.en

The sym file should be backed up frequently, at the very least after it's been successfully updated as part of the EOD processing. Copies should be stored outside the root of the database, and older copies can be updated using a program like `rsync` that very quickly performs incremental updates.

Modifying the example in `.Q.en`:

```
q)t:([]col1:`AAPL`IBM`GE`GOOG;col2:4?100.;col3:(string 4?`2))
//Splay table, rsync sym to backup dir, output same as .Q.en
q)qenBackup:{[backupdir]`:db/t/ set .Q.en[`:db;t];system"rsync db/sym ",backupdir;(),`t}
// Backup to a tmp dir for example
q)qenBackup"/tmp/bkup/"
,`t
```

Checking the `/tmp/bkup/` folder shows a file called `sym` which we can investigate:

```
q)get`:/tmp/bkup/sym
`AAPL`IBM`GE`GOOG
```

Updating the table and splaying again:

```
q)`t upsert `col1`col2`col3!(`FB;23.1;"xq")
q)qenBackup"/tmp/bkup/"
,`t
```

The new `sym` in the database is carried over with the `rsync`:

```
q)get`:/tmp/bkup/sym
`AAPL`IBM`GE`GOOG`FB
```

.Q.ens

`.Q.ens` allows enumeration against a domain other than sym. It takes the same arguments as `.Q.en` as well as the name of the domain to enumerate to, i.e. `.Q.ens[dir;table;enum]`.

```
q)`:db/t/ set .Q.ens[`:db;t;`symt]
q)system"ls ~/db"
"symt"
,"t"
q)get `:db/symt
`AAPL`IBM`GE`GOOG
```

Using multiple sym files

Depending on the system setup, it could be beneficial to enumerate data against multiple sym files within the one process, i.e. within one database there could be two tables, each with their own sym files.

This functionality is provided using the aforementioned `.Q.ens` and `.Q.dpfts`. `.Q.dpfts` functions the same as `.Q.dpft`, except we can specify the domain we want to enumerate to.

```
.Q.dpfts[directory;partition;`p#field;tablename;enum]
```

Depending on the table name, the code below will enumerate against that table's sym file. If no sym file exists for that table, it will create one.

Using `.Q.ens`:

```
q)t:([]col1:`AAPL`IBM`GE`GOOG;col2:4?100.;col3:(string 4?`2))
q){hsym[`$string[.z.D],"/",string[x],"/"] set .Q.ens\[`:.;value x;`$"sym",string x]}each tables`
,\`:2018.10.23/t/
```

Or using `.Q.dpfts`:

```
q)t:([]col1:`AAPL`IBM`GE`GOOG;col2:4?100.;col3:(string 4?`2))
q){.Q.dpfts[`:.;.z.D;`col1;x;`$"sym",string x]} each tables`
,\`t
```

In both cases, the new sym file `symt` is created for table `t`.

Copying data between databases

Copying data between two kdb+ databases is not as simple as copy and paste. When dealing with historical data, users need to be aware that the sym files are likely to be very different between databases. This means migrated data must be re-enumerated against the working sym file of the destination database.

The inbuilt functions `.Q.dpft` and `.Q.dpfts`[1] handle this efficiently and can be used programmatically for each date. Ensure the sym file is loaded into memory when using either function, otherwise a new file will be created.

Note, when saving data to disk using `.Q.dpft` or `.Q.dpfts`:

- The table cannot be keyed.
- An unmappable error will be signaled if there are columns that are not vectors or simple nested columns (e.g. char vectors for each row) present. A helper function[2] can identify the offending table column/s.
- The table name is passed in by reference so the updated data must be called the same name but only contain data for that particular date.

Given two date-partitioned databases, `db1` and `db2`, we want to move the trade table from `db2` into `db1`. The sym files for the two databases are different so we can't just copy and paste the data from one database to the other. We must enumerate against the working sym file in `db1`.

```
q)(symDB1:get`:db1/sym)~get`:db2/sym
0b
q)count symDB1
15
```

First, we load our data from both `db1` and `db2` into separate q sessions, with the second open on port 5000. We open a handle to the `db2` process from `db1` and store the dates we're interested in from `db2` in a variable `d`. This variable is used to iterate over the data in `db2` and save in `db1`. Using IPC to copy over the trade table for each date into the new database process ensures that we are updating the correct sym file, as `.Q.en` updates the *local* sym variable when it's called.

1. https://code.kx.com/v2/ref/dotq/#qdpft-save-table

2. https://code.kx.com/v2/ref/dotq/#qdpfts-save-table-with-symtable

```
q)h:hopen`::5000 /open connection to db2
q)d:2018.10.04 2018.10.05
q){[dt](hsym `$"db1/",string[dt],"/trade/") set
  update `p#sym from
    .Q.en[`:db1] delete date from
      h({`sym xasc select from trade where date=x};dt)
  }each d
`:db1/2018.10.04/trade/`:db1/2018.10.05/trade/
q)sym /new symbols have been added to the enum
`AAPL`AIG`AMD`BAC`BRK`DELL`DOW`GOOG`IBM`INTC`MSFT`ORCL`SBUX`TXN`V`PEP`RL`T`TIF
```

We now see the trade table in `db1` for the given dates and our sym file has been updated.

```
q)count get`:db1/sym
19
```

Updating data and enumerations on disk

When adding a column of type symbol to a database we must enumerate this column to prevent a type error when writing to disk. The `dbmaint.q` script makes this very easy for developers, as it already has predefined functions that consider the type of the column being added.

The main function to consider is `addcol`. Its functionality is outlined below.

- Firstly, the function checks that the name of the column being added isn't a reserved keyword – this is not allowed.
- It then calls the `add1col` function with table directory, column name and the default value as parameters.
- The default value is passed through the `enum` function to determine if enumeration needs to take place – this is the function that is doing the heavy lifting when it comes to the enumeration.
  - A check is done to see if the default value given for the new column being added is a sym. If it's not, it returns straight away.
  - If it is, the function gets the path to the sym file using `.Q.dd`, which is shorthand for `(`)sv x,`$string y` and conditionally enumerates back to the sym file on disk using `?` (Enum Extend).
  - The return value of `enum` is the default value passed in.

- `add1col` needs to be called for each of the paths. The `allpaths` function assists with this.

Compressed data

Typically, if a column contains many repeating values in contiguous order, it will compress well (parted attribute, `` `p# ``) on disk. However, while columns of type sym can be compressed, you must not compress the sym file itself. While you can still read your data with a compressed sym file, you will not be able to write to the sym file. A *no append to zipped enums* error is displayed if this occurs.

```
q)-19\!(`:db/sym;`:db/sym;17;2;6)
`:db/sym
q)t:([]col1:`FB`MSFT`AMZN`WFC;col2:4?100.;col3:(string 4?`2))
q)get `:db/sym
`AAPL`IBM`GE`GOOG
q)`:db/ set .Q.en[`:db] t
'no append to zipped enums: db/sym
  [0] `:db/ set .Q.en[`:db] t
            ^
```

For more information see the white paper ‘Compression in kdb+’ [3].

3. https://code.kx.com/v2/wp/compression_in_kdb.pdf

Datatype considerations

Symbols that are not used often map poorly to sym files. To avoid this, there are a number of considerations when deciding the different datatypes to use in kdb+ databases, discussed below.

Symbols vs strings

Use symbols where:

- Data is repeated often.
- Used in comparison checks.

Use strings where:

- Data is non-repeating.
- It's unlikely the qSQL constraints will be based around this data and, hence, queries won't benefit from symbol lookup times.

Using type symbol reduces the space required when storing data to disk if a value occurs in multiple locations in the database. Each instance of the value doesn't need to be stored, just its position in the string pool.

Guids

If queries against an ID column are required, use the guid type. A guid is a 16-byte type, and can be used for storing arbitrary 16-byte values. As data is persisted to disk,the ID field can be converted to guid and persisted along with the original ID. IDs requiring lookup should not be char vectors. Guids are faster (much faster for =) than the 16-byte char vectors and take 2.5 times less storage (16 per instead of 40 per).

Encoding/decoding

The functions `.Q.j10` and `.Q.x10` (encode/decode binhex) and `.Q.j12` and `.Q.x12` (encode/decode base64) return a string encoded or decoded against restricted alphabets. The main use of these functions is to encode long alphanumeric identifiers (CUSIP, ORDERID, etc.) so they can be quickly searched – but without filling up the sym file with vast numbers of single-use values. The numbers at the end of the function indicate the max length of string that can be used for each function. Below is an example of encoding/decoding using base64:

```
q)t
col1 col2     col3 col4
-----------------------------
AAPL 17.80839 "mi" "369604103"
IBM  30.17723 "ig" "256677105"
GE   78.5033  "ka" "778296103"
GOOG 53.47096 "ba" "18581108"

q)update gID:.Q.j12 each col4 from t /encode
col1 col2     col3 col4        gID
--------------------------------------------
AAPL 17.80839 "mi" "369604103" 8953468732947
IBM  30.17723 "ig" "256677105" 6047476211669
GE   78.5033  "ka" "778296103" 20313869089683
GOOG 53.47096 "ba" "18581108"  96094238552

q)update gxID:.Q.x12 each gID from t /decode
col1 col2     col3 col4        gID            gxID
-----------------------------------------------------------
AAPL 17.80839 "mi" "369604103" 8953468732947  "000369604103"
IBM  30.17723 "ig" "256677105" 6047476211669  "000256677105"
GE   78.5033  "ka" "778296103" 20313869089683 "000778296103"
GOOG 53.47096 "ba" "18581108"  96094238552    "000018581108"
```

If the string is not of length 12 when decoding from base64, it is padded with leading zeros.

The datatype decision should be something that is carefully considered during the design phase of a database, as a poor design could be troublesome in the long run where we could need to run maintenance on the sym file.

Periodic checks on the size of the sym file to measure its rate of growth can help detect columns which were poorly chosen as symbol type, providing an early-warning system to prevent bloating.

Compacting bloated sym files

A sym file becomes bloated when a significant number of symbols in the sym domain are no longer in use. This section will explore two ways of compacting the sym file. Code is discussed in detail where applicable. Compacting a sym file involves unenumerating data and then re-enumerating once the data has been tidied up.

Common situations where compacting a sym file could be considered are:

- If the size of the database's sym file is becoming large and it's taking some time to load the file into the HDB process.
- Saving to disk is taking a long time, as kdb+ has to enumerate against a bloated sym file. This can become problematic especially during intraday write-downs.
- Syms that were saved in an earlier version of the system may not be used in the dataset any longer.
- Columns were erroneously typed during the design phase and need to be updated.

Before compacting the sym file it's a good idea to consider deleting older partitions that are no longer required, and/or updating the schema and running `dbmaint` on all partitions to ensure that we are compacting the file efficiently.

Back up the database and sym file before performing any of the below operations.

Single-threaded sym rewrite

The function[4] that we're going to discuss in detail below was written by Charles Skelton and can be used as a template for compacting a sym file. Changes have been made where needed for V3.6 of kdb+, with comments added where necessary. The function assumes a single enumeration (sym), and only splayed tables in a date partitioned database are present.

The first step in the process is to move to the root directory of the database and start a q session. *Note that we haven't loaded any data into the process.* Create a backup of the current sym file by moving the old sym file to an alternative file name and create an empty sym file. We need to use a file named `sym` as this is the default domain that will be used by `.Q.en`.

4. https://code.kx.com/v2/kb/compacting-hdb-sym/

Next, store a list of files in the directory by keying the directory path. From this variable, we are able to extract all the date files.

```
system "mv sym zym";
`:sym set `symbol$(); / create a new empty sym file
files: key `:.;
dates: files where files like "????.??.??";
```

For each of the dates that we've extracted from the filenames we will iterate over the function below.

```
{[d]
  root:":",string d;
  tableNames:string key `$root;
  tableRoot:root,/:"/",/:tableNames;
  files:raze {`$x,/:"/",/:string key `$x}each tableRoot;
  files:files where not any files like/: ("*#";"*##");
  // indicates compound and anymap files, we don't want to work on these files
  types:type each get each files;
  enumeratedFiles:files where types=20h; /all 64-bit enums type 20h
  // if we have more than one enum better get help
  if[not 1=count distinct{key get x}each enumeratedFiles;'"too difficult"];
  {
    `sym set get `:zym;
    s:get x;
    a:attr s;
    s:value s;
    `sym set get `:sym;
    s:a#.Q.en[`:.;([]s:s)]`s;
    x set s;
    -1 "re-enumerated ", string x;
    }each enumeratedFiles;
  }each dates
```

Below the date directory, we get all the table names by using `key` on the path once again. The variable `tableRoot` will then contain a full path to each of the table names. This is done by using Join Each Right (`,/:`) against the `root` variable. Note the addition of an extra forward slash to ensure a legitimate directory path.

```
root:":",string d;
tableNames:string key `$root;
tableRoot:root,/:"/",/:tableNames;
files:raze {`$x,/:"/",/:string key `$x}each tableRoot;
```

Next, we remove any files that have names which end in # (compound column files) or ## (anymap files) as we won't need to perform any action on these. The type of each of the files is obtained by using keyword `get` to extract the data to perform the type call. The information containing the type of each kdb+ file is stored in the header of the file.

```
files:files where not any files like/: ("*#";"*##");
types:type each get each files;
```

Only files that have an enumerated type should be considered in this function so we store in `enumeratedFiles` only files with type `20h` as these are the types given to all 64-bit enumerations. Next, check how many enum domains we have in our dataset. If the dataset contains multiple domain enumerations it is out of the scope of this function so we exit with a signal that it's too difficult.

```
types:type each get each files;
enumeratedFiles:files where types=20h; /all 64-bit enums type 20h
/ if we have more than one enum better get help
if[not 1=count distinct{key get x}each enumeratedFiles;'"too difficult"];
```

We want to re-enumerate each of these enumerated files against the new sym file. Because it's an enumerated file it contains only the integer pointer values to the sym-file string pool so we read the old sym file into variable `` `sym``. We use `sym` as this is what will be loaded into memory from the sym file if we were to load the database into the q session.

```
`sym set get `:zym;
```

Performing `get` on the enumerated file after reading the old sym file into memory will allow us to store the unenumerated values into variable `s`, and `a` will store the attribute, if any, that is applied to the column. Reset the sym variable in memory to contain the string pool values from our new sym file, as we want to add our re-enumerated data to this new file.

```
s:get x;
a:attr s;
s:value s;
`sym set get `:sym;
```

Then we perform the enumeration, creating a single column table with data `s` and applying the attribute to the data (if any). This is stored back into variable `s` and we set this data back to the enumerated file that we passed in.

```
s:a#.Q.en[`:.;([]s:s)]`s;
x set s;
```

This function is first run over all enumerated files for a particular date, and then all dates. The amount of time this operation takes to run is proportional to the number of partitions and enumerated files in each partition. Benchmarking against one date could help determine how long the process will take for an entire HDB.

Multithreaded sym rewrite

The same recipe from the previous section contains code that allows multi-threading to re-write the sym file (sym domain only). The same precautions as mentioned above should be followed along with the knowledge that this method is more memory intensive, albeit much faster. This multithreaded version can handle partitioned and splayed tables within the same directory, and `par.txt`. Note the loss of the grouped attribute (`` `g# ``), which isn't supported in threads, and must be reapplied later. This method of re-writing the sym file also differs from the one discussed above because we load our data into a process in this instance.

We define a slightly modified version of the `allpaths` function from `dbmaint.q`[5] so we can retrieve the full paths to each of the files that actually have data in them. Note the check in the last line of the allpaths function.

```
system"l ."
allpaths:{[dbdir;table]
  files:key dbdir;
  if[any files like"par.txt";
    :raze allpaths[;table]each hsym each`$read0(`)sv dbdir,`par.txt];
  files@:where files like"[0-9]*";
  files:(`)sv'dbdir,'files,'table;
  files where 0<>(count key@)each files}
```

Next, we need to unenumerate and store the result in variable `sym` as we did before.

```
sym:oldSym:get`:sym
```

We first get the full file paths for columns of type symbol for all partitioned tables only. The check for partitioned tables is done using `.Q.qp`, which will return `1b` for a given table name if it is partitioned, `0b` for a splayed table, and `0` for anything else.

```
/sym files from parted tables
symFiles:raze` sv/:/:raze{
  allpaths[`:.;x],/:\:exec c from meta[x] where t in "s"
  }peach tables[] where {1b~.Q.qp value x}each tables[]
```

5. https://github.com/KxSystems/kdb/blob/master/utils/dbmaint.q

The same is done for splayed tables – where `0b~.Q.qp` for a table.

```
/sym files from splayed tables
symFiles,:raze{
  ` sv/: hsym[x],/:exec c from meta x where t in > "s"
  }each tables[] where {0b~.Q.qp value x}each tables[]
```

This next snippet of code is quite memory-intensive as we obtain a distinct symbol list from all of the symbol column file paths that we've generated. As before, it's difficult to estimate the duration that this operation can take so patience is required depending on the number of files.

```
/symbol files we're dealing with - memory intensive
allsyms:distinct raze{[file] :distinct @[value get@;file;`symbol$()]} peach symFiles
```

Performing garbage collect at this stage will assist in freeing up any fragmented system memory.

```
.Q.gc[] /memory intensive so gc
```

None of the code discussed up until this point has made any changes to the database. At this stage, it's beneficial to compare the counts of the symbol lists we now have in memory – the current sym-file list and our new list of syms that are being used in the dataset.

```
count[allsyms]%count sym
```

Should the number justify continuing, the code below will make changes to the database and can be considered a point of no return. The code follows the same format as the compacting sym section discussed in detail above, with the main difference being that we are using `peach` to multithread across the sym files.

```
system"mv sym zym" /make backup of sym file
`:sym set `symbol$() /reset sym file
`sym set get`:sym /set sym variable in memory
.Q.en[`:.;([]allsyms)] /enumerate all syms at once
{[file]
  s:get file; /file contents
  //attributes - due to no`g# error in threads
  //this can be just a:attr s
  //if your version of kdb+ does support setting `g\# in threads
  a:first `p`s inter attr s;
  s:oldSym`int$s; /unenumerate against old sym file
  file set a#`sym$s; /enumerate against new sym file & add attrib &
  write to disk
  0N!"re-enumerated ", string file;
  } peach symFiles
```

Conclusion

While the q language provides tools to easily create and update HDBs through functions in the `.Q` namespace, understanding how the underlying code and concepts work is integral to creating and maintaining a functioning database.

In this paper, we demonstrated, through examples, enumeration in kdb+ and various methods of using symbols within an HDB; using multiple sym files and moving data between HDBs. In situations where you need new columns `dbmaint.q` provides utility functions to manage this with ease.

We discussed datatype considerations in order to utilize the performance benefits of enumerations while maintaining a sym universe containing only necessary symbols. However, if there is still a need to perform maintenance on the sym file because of bloat we walked through two methods to reduce the size of the sym file.

All tests performed using kdb+ version 3.6 (2018.05.17).

Appendix – for versions of kdb+ earlier than 3.6

Enumeration

Prior to V3.6 of kdb+, the type of an enumeration could range from `20h` to `76h`, with each new domain given an incremented type. Type `20h` is reserved for the sym domain (`` `sym$ ``).

```
q)p:`AAPL`AAPL`GE`IBM`GE`IBM`GE`GOOG`GE`IBM`AAPL
q)x:`u#distinct p
q)e:`x$p
q)e
`x$`AAPL`AAPL`GE`IBM`GE`IBM`GE`GOOG`GE`IBM`AAPL
q)"i"$e
0 0 1 2 1 2 1 3 1 2 0i
q)type e /21h as 20h is reserved for sym
21h
```

Adding a second domain to our process illustrates how domain types increment.

```
q)excg:`N`OQ`L`TO /this will be our domain
q)l:10?excg
q)l
`OQ`L`OQ`L`TO`TO`N`N`L`N
q)e2:`excg?l
q)e2
`excg$`OQ`L`OQ`L`TO`TO`N`N`L`N
q)type e2
22h
```

From this we can determine that we can only have 56 enums per q session.

Using multiple sym files

When saving data to disk using multiple sym files with a version of kdb+ released prior to V3.6 it's important to be aware that `.Q.en` enumerates to the sym domain by default so we will need to tweak q's out-of–the-box functionality to ensure that tables are enumerated to the correct file and data is saved successfully.

The code below takes three arguments: directory path, the table name reference, and the table itself obtained from the table name reference using `value`. Depending on the table name, it will enumerate against that table's sym file. If no sym file exists for that table, it will create one.

```
enumerate:{[d;t;tab]
  /res is a mapping from column name to column (list)
  res:cols[tab]!{[d;n;c]
    //only enumerate if the column is a symbol, else return it.
    $[11=type c;
      [n:`$"sym",string n; /sym file for table
      /load the file if it exists
      @[load;f:` sv d,n;n set `symbol$()];
      if[not all c in value n;
        f set value n set `u#distinct value[n],c];
      n$c];
      c]
    }[d;t;]'[value flip tab]; /for each column in the table
  flip res /return table}
```

This function can be used to save all tables in the current q session into a partition of your choice. Below we're using today's date as our partition.

```
{
  hsym[`$string[.z.D],"/",string[x],"/"] set enumerate[`:.; x; value x]
  }each tables[`.] where 98=type each get each tables`.
```

Note that this functionality only works on un-keyed tables. If a key is necessary, unkey the table, save to disk, reload and then key the table from within the q process.

Because a new enumeration is made for every table, the limit is 55 custom files using this method (types `21h-76h`, `20h` being reserved for sym). This holds true when upgrading 32-bit enums files from previous versions to V3.6 despite no practical limit to the number of 64-bit enum domains. All ops cast 32-bit enums to 64-bit in V3.6.

Compacting bloated sym files

Compacting a sym file with a version of kdb+ earlier than 3.6 is very similar to the function discussed in detail above. There are two main distinctions between the original function Charlie Skelton wrote and the one in this paper that works for V3.6.

- Anymap files are new to V3.6 so we only need to worry about compound files (#) in V3.5 and below.
- Type `20h` is reserved for the sym domain so if there are any files that are of a type within `21h-76h` it means we have one or more tables with multiple domain enumeration, which is out of the scope of the function.