it's about time

Technical Whitepaper

Kdb+ Data Management: Sample Customization Techniques

Author:

Simon Mescal is based in New York, Simon is a Financial Engineer who has designed and developed kdb+ related data management solutions for top tier investment banks and trading firms, across multiple asset classes. Simon has extensive experience implementing a range of applications in both object oriented and vector programming languages.

Table of Contents

1 INTRODUCTION

This paper illustrates the flexibility of kdb+ and how instances can be customized to meet the practical needs of business applications across different financial asset classes. Given the vast extent to which kdb+ can be customized, this document focuses on some specific cases relating to the capture and storage of intra-day and historical time-series data. Many of the cases described should resonate with those charged with managing large scale kdb+ installations. These complex systems can involve hundreds of kdb+ processes and/or databases supporting risk, trading and compliance analytics powered by petabytes of data, including tick, transaction, pricing and reference data.

The goal of the paper is to share implementation options to those tasked with overseeing kdb+ setups, in the hope that the options will allow them to scale their deployments in a straight forward manner, maximize performance and enhance the end user experience. Seven scenarios are presented focused on tuning the technology using some common and uncommon techniques that allow for a significantly more efficient and performant system. The context where these techniques would be adopted is discussed, as well as the gains and potential drawbacks of adopting them.

Where appropriate, sample code snippets are provided to illustrate how the technique might be implemented. The code examples are there for illustrative purposes only and we recommend that you always check the use and effect of any of the samples on a test database before implementing changes to a large dataset or production environment. Sample on disk test databases can be created using scripts found at http://www.kx.com/q/taq/tq.q

Code examples are intended for readers with at least a beginner's level knowledge of kdb+ and the q language. However we encourage any interested reader to contact us at lectureseries@firstderivatives.com with questions regarding code syntax or any other aspect of the paper.

2 SAMPLE DATA MANAGEMENT TECHNIQUES

2.1 Multiple Enumeration Files

In splayed databases, the default behavior is that all columns of type symbol are enumerated using the same list, a file called sym. Enumerating in this context means that rather than containing the actual symbol values, the column, when saved to disk, will consist of a list of integers each of which is the index of the actual value in the list (file) called sym. For more on enumerations, see
http://code.kx.com/wiki/JB:QforMortals/casting_and_enumerations

This scenario is not always desirable in databases containing multiple tables and numerous columns of type symbol, which may represent for instance different types of security identifier, country codes, and exchange codes. If the sym file becomes corrupt then all the symbol columns will be lost. By decoupling the columns of type symbol - maintaining separate enumeration files for each symbol column – corruption or loss of data in one enumeration file will not impact on all columns of type symbol. The q function which enumerates a table is `.Q.en`. It takes a directory path and a table and enumerates the table using a file called sym in the directory (if there is no such file it creates it). The enumerate function below takes the same arguments but, rather than enumerating against the sym file, it enumerates against a different file for each column, with the file name matching the column name.

```
enumerate:{[d;t] /directory,table
/t is a mapping from column name to column (list)
      t:cols[t]!
      {[d;n;c] /directory,name,column

/if column is a symbol enumerate it, otherwise return it
            $[11=type c;
            [     @[load;f:` sv d,n;n set `symbol$()]; /load the file if it
exists
                  if[not all c in value n;
                        f set value n set distinct value[n],c /add new values
to the enum file
                  ];
                  n$c /enumerate the list (c) with enumeration file
            ];
                  c
            ]
      }[d]'[cols t;value flip t]; /iterate over each column-name/column pair
      flip t
 }
/continued overleaf
```

```
/the following line calls enumerate for each (non-keyed) table in the root
namespace, writing the table in the current date directory

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

A slight variation on the enumerate function detailed below specifies a mapping from column to enumeration file so that certain columns share enumeration files.

```
/mapping from columns names to enumeration names
columnFileMapping:![`cusip`isin`symbol`market`currency`sector`trader`traderI
D;
      `products`products`products`markets`markets`markets`people`people]
enumerate:{[d;t] /directory,table
      t:cols[t]!
      {[d;n;c] /directory,name,column
/if there is no mapping for column n, use sym
            enum:`sym^columnFileMapping n;
            $[11=type c;
            [     @[load;f:` sv d,enum;enum set `symbol$()];
                  if[not all c in value enum;
                        f set value enum set distinct value[enum],c
                  ];
                  enum$c
            ];
                  c
            ]
      }[d]'[cols t;value flip t];
      flip t
 }
```

2.2 Automating Schema Changes

In some kdb+ installations, table schemas change frequently. This can be difficult to manage if the installation consists of a splayed partitioned database since each time the schema changes you have to manually change all the historical data to comply with the new schema. This is time-consuming and prone to error (dbmaint.q in the contributed code section of the KX website is a useful tool for performing these manual changes). To reduce the administrative overhead, the historical database schema can be updated programmatically as intra-day data is persisted. This can be achieved by invoking function `updateHistoricalSchema` (overleaf), which connects to the historical database and runs locally defined functions to add and remove tables, add and remove columns, reorder columns and change their types. It does this by comparing the table schemas in the last partition with those of the other partitions, and making the necessary adjustments to the other partitions. Note that this code is only compatible with kdb+ versions 2.6 and upwards (in previous versions the meta data is read from the first, rather than the last, partition therefore minor changes are necessary to get this to work on earlier versions). Note also that column type conversion (function `changeColumnTypes`) will not work when changing to or from nested types or symbols - this functionality is beyond the scope of this paper.

The updateHistoricalSchema function should be invoked once the in-memory data is fully persisted. If kdb+ tick is being used, the function should be called as the last line of the .u.end function in r.q - the real-time database. updateHistoricalSchema takes the port of the historical database as an argument, opens a connection to the historical database and calls functions that perform changes to the historical data. Note that the functions are defined in the calling process. This avoids having to load functionality into the historical process, which usually has no functionality defined in it.

```
updateHistoricalSchema:{[con]
      h:hopen con;
      h(addTables;());
      h(removeTables;());
      h(addColumns;());
      h(removeColumns;());
      h(reorderColumns;());
      h(changeColumnTypes;());
 }
```

The following function simply adds empty tables to any older partitions based on the contents of the latest partition.

```
addTables:{.Q.chk`:.}
```

The following function finds all tables that are not in the latest partition but are in other partitions and removes those tables.

```
removeTables:{
      t:distinct[raze key each hsym each `$string -1_date]except tables`.;
      {@[system;x;::]}each "rm -r ",/:string[-1_date] cross "/",/:string t;
 }
```

The function overleaf iterates over each table-column pair in all partitions except for the latest one. It makes sure all columns are present and if not, creates the column with the default value of the column type in the latest partition.

```
addColumns:{
      {[t]
            {[t;c]
                  {[t;c;d]
                        defaults:" Cefihjsdtz"!("";""),first each
"efihjsdtz"$\:();
                        if[0=type key
f:hsym`$string[d],"/",string[t],"/",string c;
                              f set count[get
hsym`$string[d],"/",string[t],"/sym"]#defaults meta[t][c]`t;
                              @[hsym`$string[d],"/",string[t];`.d;,;c]
                        ]
                  }[t;c]each -1_date
            }[t] each cols[t]except `date
      }each tables`.
 }
```

The following function deletes columns in earlier partitions that are not in the last partition. It does this by iterating over each of the tables in all partitions, except for the last one, getting the columns that are in that partition but not in the last partition and deletes them.

```
removeColumns:{
      {[t]  {[t;d]
                  delcols:key[t:hsym`$string[d],"/",string t]except cols t;
                  {[t;c]
                        hdel`$string[t],"/",string c;
                  }[t]each delcols;
                  if[count delcols;
                        @[t;`.d;:;cols[t] except `date]
                  ]
            }[t] each -1_date
      }each tables`.
 }
```

The function overleaf re-orders the columns by iterating over each of the partitions except for the last partition. It checks that the column order matches that of the latest partition by looking at the .d file. If there is a mismatch, it modifies the .d file to the column list in the last partition.

```
reorderColumns:{
/.d file specifies the column names and order
      {[d]
            {[d;t]
                  if[not except[cols t;`date]~get
f:hsym`$string[d],"/",string[t],"/.d";
                        f set cols[t] except `date
                  ]
            }[d]each tables`.
      }each -1_date
 }
```

The following function iterates over every table-column pair in all partitions except for the last. It checks that the type of the column matches that of the last partition and if not, it casts it to the correct type.

```
changeColumnTypes:{
      {[t]
            {[t;c]
                  typ:meta[t][c]`t;
                  frst:type get hsym`$string[first
date],cpath:"/",string[t],"/",string c;
                  lst:type get hsym`$string[last date],cpath;
/if type of column in first and last partition are different
/and type in last partition is not symbol, character or list
/and type in first partition is not generic list, char vector or symbol
/convert all previous partitions to the type in last partition
                  if[not[frst=lst]&not[typ in "sc ",.Q.A]&not frst in 0 10
11h;
                        {[c;t;d]
                              hsym[`$string[d],c] set t$get hsym`$string[d],c
                        }[cpath;typ]each -1_date
                  ]
            }[t]each cols[t]except `date
      }each tables`.
 }
```

2.3 Attributes on Splayed Partitioned Tables

Attributes are used to speed up table joins and searches on lists. There are 4 different attributes:

grouped (`g#)
parted (`p#)
unique (`u#)
sorted (`s#)

To apply the parted attribute to a list, all occurrences in the list of each element n must be adjacent to one another. For example, the following list is not of the required structure and attempting to apply the parted attribute to it will fail:

3 3 5 5 5 3

However, the following list is of the required structure and the parted attribute can be applied:

5 5 5 3 3 4 4

Sorting a list in ascending or descending order guarantees that the parted attribute can be applied.

Likewise, to apply the unique attribute, the elements of the list must be unique and to apply the sorted attribute, the elements of a list must be in ascending order. The grouped attribute does not dictate any constraints on the list and the attribute can be applied to any list of atoms.

When the unique, parted or grouped attribute is set on a list, q creates a hashtable alongside the list. For a list with the grouped attribute, the hashtable maps the elements to the indices where they occur, for a parted list it maps to the index at which the element starts and for a list with the unique attribute applied, it maps to the index of the element. The sorted attribute merely marks the list as sorted, so that q knows to use binary search on certain function (such as `=`, `in` , `?`, `within`).

Now we explore the use of attributes on splayed partitioned tables. The most common approach with regard to attributes in splayed partitioned tables is to set the parted attribute on the security identifier column of the table, as this is usually the most commonly queried column. However, we are not limited to just one attribute on splayed partitioned tables. Given the constraint required for the parted and sorted attributes, as explained above, it is rarely possible to apply more than one parted, more than one sorted or a combination of the two on a given table. There would have to be a specific functional relationship between the two data sets for this to be feasible. That leaves grouped as a viable additional attribute to exist alongside parted or sorted.

To illustrate the performance gain of using attributes, the following query was executed with the various attributes applied, where the number of records in t in partition x was 10 million, the number of distinct values of c in partition x is 6700 and c was of type enumerated symbol. Note that the query itself is not very useful but does provide a useful benchmark.

```
select c from t where date=x,c=y
```

The query was 18 times faster when column c had the grouped attribute compared with no attribute. With c sorted and parted attribute applied the query was 29 times faster than with no attribute. However, there is a trade off with disk space. The column with the grouped attribute applied consumed 3 times as much space as the column with no attribute. The parted attribute applied consumed only 1% more than no attribute. Parted and sorted attributes are significantly more space efficient than grouped and unique.

Suppose the data contains IDs which for a given date are unique, then the unique attribute can be applied regardless of any other attributes on the table. Taking the same dataset as in the example above with c (of type long instead of enumerated symbol) unique for each date and the unique attribute applied, the same query is 42 times faster with the attribute compared with no attribute. However, the column consumes 5 times the amount of disk space.

Sorting the table by that same column and setting the sorted attribute results in the same query performance as with the unique attribute, and the same disk space consumed as with no attribute. However, the drawback of using sorted, as with parted for reasons explained above, is not being able to use the sorted or parted attributes on any other columns.

The following is an approximation of the additional disk space (or memory) overhead, in bytes, of each of the attributes, where n is the size of the list and u is the number of unique elements.

Table 1 –Disk space/memory overhead of each attribute

Attribute	Space overhead
sorted	0
unique	16*n
parted	24*u
grouped	(24*u)+4*n

2.4 ID Fields – The GUID Data Type

It is often the case that a kdb+ application receives an ID field which does not repeat and contains characters that are not numbers. The question is which data type should be used for such a field. The problem with using enumerated symbols in this case is that the enumeration file becomes too large, slowing down searches and resulting in too much memory being consumed by the enumeration list.

(see http://code.kx.com/wiki/Cookbook/SplayedTables for more on splayed tables and enumerations)

The problem with using char vector is that it is not an atom, so attributes cannot be applied, and searches on char vectors are very slow. In this case, it is worth considering GUID which can be used for storing 16-byte values (kdb+ version 3.0 onwards, see http://code.kx.com/wiki/Reference/Datatypes).

As the data is persisted to disk, or written to an intra-day in-memory instance, the ID field can be converted to GUID and persisted along with the original ID. The following code shows how this would be done in memory. It assumes that the original ID is a char vector of no more than 16 characters in length.

```
/guidFromRawID function pads the char vector with spaces to the left,
converts to byte array, converts byte array to GUID
guidFromRawID:{0x0 sv `byte$#[16-count x;" "],x}
update guid:guidFromRawID each id from `t
```

Once the GUID equivalent of the ID field is generated, attributes can be applied to the column and queries such as the following can be performed.

```
select from t where date=d,guid=guidFromRawID["raw-id"]
```

If the ID is unique for each date, the unique attribute could be considered for GUID, but at the expense of more disk space as illustrated above.

If the number of characters in the ID is not known in advance, one should consider converting the right most 16 characters to GUID (or left most depending on which section of the ID changes the most), and keeping the remaining 16 character sections of the ID in a column consisting of GUID lists. An alternative is to just convert the right most 16 characters to GUID and use the original ID in the query as follows:

```
select from t where guid=guidFromRawID["raw-id"],id like "raw-id"
```

In this case, the right most 16 characters will not necessarily be unique therefore parted or grouped might be wise choices of attribute on the GUID column in the splayed partitioned table.

2.5 Combining Real-time and Historical Data

Typically, intra-day and historical data reside in separate kdb+ processes. Although there are instances when this is not desirable:

- The data set may be too small to merit multiple processes
- Users might want to view all their data in one location
- It may be desirable to minimise the number of components to manage
- Performance of intra-day queries may not be a priority (by combining intra-day and historical data into a single process intraday queries may have to wait for expensive I/O bound historical queries to complete)

Moreover, by having separate processes for intra-day and historical data, it is often necessary to introduce a gateway as a single point of contact for querying intra-day and historical data, thereby running three processes for what is essentially a single data set.

However, one of the main arguments against combining intraday and historical data is that large I/O bound queries on the historical data prevent updates being sent from the feeding process. This results in the TCP/IP buffer filling up, or blocking the feeding process entirely if it is making synchronous calls.

The code snippet below provides a function which simply writes the contents of the in-memory tables to disk in splayed partitioned form, with different names to their in-memory equivalent. The function then memory maps the splayed partitioned tables.

```
eod:{
/get the list of all in-memory, non-keyed tables
     t:tables[`.] where {not[.Q.qp x]&98=type x} each get each tables`.;
/iterate through each of the tables, x is a date
     {[d;t]
/enumerate the table, ascend sort by sym and apply the parted attribute
/save to a splayed table called <table>_hist
          hsym[`$string[d],"/",string[t],"_hist/"] set update
`p#optimisedColumn from `optimisedColumn xasc .Q.en[`:.;get t];
/delete the contents of the in-memory table and re-apply the grouped
attribute
          update `g#optimisedColumn from delete from t
     }[x]each t;
/re-memory map the splayed partitioned tables
     system"l ."
 }
```

The above will result in an additional (splayed partitioned) table being created for each non-keyed in-memory table. When combining intra-day and historical data into one process, it is best to provide functions to users and external applications which handle the intricacies of executing the query across the two tables (as a gateway process would), rather than allow them to run raw queries.

2.6 Persisting Intra-day Data to Disk Intra-day

Let us now assume there are memory limitations and you want the intra-day data to be persisted to disk multiple times throughout the day. The following example assumes the intraday and historical data reside in the same process, as outlined in the previous section. However in most cases large volumes of data dictate that separate historical and intraday processes are required. Accordingly, minor modifications can be made to the below code to handle the case of separate intra-day and historical processes. One way of achieving intraday persistence to disk is to create a function such as the one below, which is called periodically and appends the in-memory tables to their splayed/partitioned counterparts. The function takes the date as an argument and makes use of splayed `upsert` which appends to the splayed table (as compared to set which overwrites).

```
flush:{[d]
     t:tables[`.] where {not[.Q.qp x]&98=type x} each get each tables`.;
     {[d;t]
/f is the path to the table on disk
          f:hsym`$string[d],"/",string[t],"_hist/";
/if the directory exists, upsert (append) the enumerated table, otherwise
set (creates the splayed partitioned table)
          $[0=type key f;
               set;
               upsert
          ][f;.Q.en[`:.;get t]];
          delete from t
     }[d]each t;
/re-memory map the data and garbage collect
     system"l .";
     .Q.gc[]
 }
```

The above flush function does not sort or set the parted attribute on the data as this would take too much time blocking incoming queries - it would have to re-sort and save the entire partition each time. You could write functionality to be called at end-of-day to sort, set the parted attribute and re-write the data. Alternatively, to simplify things, do not make use of the parted attribute if query performance is not deemed a priority or use the grouped attribute that does not require sorting the data (see section 2.3 on Attributes on Splayed Partitioned Tables).

If the component which feeds the process is a tickerplant (see kdb+ tick), it is necessary to truncate the tickerplant log file (file containing all records received) each time flush is called, otherwise the historical data could end up with duplicates. This can happen if the process saves to disk and then restarts, retrieving those same records from the log file which will subsequently be saved again. The call to the flush function can be initiated from the tickerplant in the same way it calls the end-of-day function. If the source of the data is something other than a component which maintains a file of records received, such as a tickerplant, then calling the flush function from within the process is usually the best approach.

2.7 Eliminate the Intra-day Process

In some cases, only prior day data is needed and there is no requirement to run queries on intra-day data. However, you still need to generate the historical data daily. If daily flat files are provided then simply use a combination of the following functions to load the data and write it to the historical database in splayed partitioned form.

Table 2 – Functions for creating splayed partitioned tables

Function	Brief Description
.Q.dsftg	Load and save file in chunks
.Q.hdpf	Save in-memory table in splayed partitioned format
.Q.dpft	Same as above, but no splayed partitioned process specified
.Q.en	Returns a copy of an in-memory table with symbol columns enumerated
0:	Load text file into an in-memory table or dictionary
set	Writes a table in splayed format
upsert	Appends to a splayed table

If kdb+ tick is being used, simply turn off the real-time database and write the contents of the tickerplant log file to the historical database as follows:

```
\l schema.q                       /load schema
\cd /path/to/hdb                  /change dir to HDB dir
upd:insert
-11!`:/path/to/tickerplantlog     /replay log file
.Q.hdpf[HDB_PORT;`:.;.z.D;`partedColumn]
```

If there are memory limitations on the box, then `upd` can be defined to periodically flush the in-memory data to disk as follows:

```

\l schema.q
\cd /path/to/hdb
flush:{
/for each table of count greater than 0
/create or append to the splayed table
/continued overleaf
```

```
      {[t]  f:hsym`$string[.z.D],"/",string[t],"/";
            $[0=type key f;
                  set;
                  upsert
            ][f;.Q.en[`:.;get t]];
            delete from t
      }each tables[`.] where 0<count each get each tables`.;
 }

upd:{insert[x;y];
/call flush if any tables have more than 1 million records
      if[any 1000000<count each get each tables`.;
            flush[]
      ]
 }
/replay the log file
-11!`:/path/to/tickerplantlog
/write the remaining table contents to disk
flush[]
```

If kdb+ tick is not being used, the equivalent of the tickerplant log file can be created simply by feeding the data into a q process via a callback function, which writes it to a file and persists it to disk as follows:

```
f:`:/path/to/logfile
f set ()

/initialise the file to empty list
h:hopen f
callback:{h enlist(`upd;x;y)} /x is table name, y is data
```

3 CONCLUSION

When designing kdb+ applications to tackle the management of diverse and large volumes of content, you must take many business and technical factors into account. Considerations around the performance requirements of the application, network and hardware, personnel constraints, ease of maintenance and scalability all play a role in running a production system. This document has outlined practical options available to system managers for addressing some of the common scenarios they may encounter when architecting and growing their kdb+ deployment. In summary, the questions that frequently arise when one is in the process of designing robust, high performance kdb+ applications include:

- What are the appropriate datatypes for the columns?
- How often is it expected that the schema will change and is the need for manual intervention in ensuring data consistency a concern?
- What are the fields that will be queried most often and what (if any) attributes should be applied to them?
- What is the trade-off between performance and disk/memory usage when the attributes are applied and is it worth the performance gain?
- Is it a concern that symbol columns would share enumeration files, and if so, should the enumerations be decoupled?
- Is it necessary to make intra-day data available? If not, consider conserving in-memory requirements by, for example, persisting to disk periodically, or writing the contents of a tickerplant log file to splayed format at end-of-day.
- Consider the overhead of maintaining separate intra-day and historical processes. Is it feasible or worth while to combine them into one?

As mentioned in the introduction, a key feature of kdb+ is that it is a flexible offering that allows users to extend the functionality of their applications through the versatility of the q language. There are vanilla intra-day and historical data processing architectures presented in existing documentation that cover many standard use cases. However, it is also common for a kdb+ system to quickly establish itself as a success within the firm, resulting in the need to process more types of data and requests from downstream users and applications. It is at this point that a systems manager is often faced with balancing how to ensure the goals of performance and scalability are realised, while at the same time dealing with resource constraints and the need for maintainability. The seven cases and customisation techniques covered in this paper provided examples of what you can do to help achieve those goals.