it's about time

Technical Whitepaper

The Application of Foreign Keys and Linked Columns in kdb+

Author:

Kevin Smyth has worked as a consultant for some of the world's leading financial institutions. Based in London, Kevin has implemented data capture and high-frequency data analysis projects across a large number of mainstream and alternative asset classes.

Table of Contents

1 INTRODUCTION

Tables in a database define a relationship between different types of data, whether that relationship is static, dynamic (i.e. fluctuating as part of a time series) or a mixture of both. In general, it is regularly the case that database queries will require data from multiple tables for enrichment and aggregation purposes and so a key aspect of database design is developing ways in which data from several tables is mapped together quickly and efficiently. Although kdb+ contains a very rich set of functions for joining tables in real time, if permanent and well-defined relationships between different tables can be established in advance then data retrieval latency and related memory usage may be significantly reduced.

This whitepaper will discuss foreign keys and linked columns in a kdb+ context, two ways whereby table structure and organisation can be optimised to successfully retrieve and store data in large-scale time series databases.

Tests performed using kdb+ 3.0 (2013.04.05).

2 FOREIGN KEYS

2.1 Simple Foreign Keys

The concept behind a foreign key is analogous to that of an enumerated list. While enumerating a list involves separating it into its distinct elements and their associated indexes within the list, here we take an arbitrary table column and enumerate it across a keyed column, which may be either in the same table as itself or a different table in the same database. Behind the scenes, a pointer to the associated key column replaces the enumerated column's values essentially creating a parent-child relationship if they are in the same table or a data link if they are in different tables.

In the case of a single key enumeration, creating a foreign key is very straightforward and is specified either within the initial table definition or on the fly through an update statement:

```
//Keyed table that will be used for enumeration; values in the keyed
//column must completely encapsulate the values in the column being
//enumerated
q)financials:([sym:`A`B`C]earningsPerShare:1.2 2.3
1.5;bookValPerShare:2.1 2.5 3.2)
//Schema of trade table prior to enumeration
q)trade:([]time:`time$();sym:`$();price:`float$())
//Use the '$' operator to create an enumerated list within the trade
//table by casting to the keyed table 'financials'
q)update sym:`financials$sym from `trade
//Alternatively the foreign key may be defined at the outset
q)trade:([]time:`time$();sym:`financials$();price:`float$())
q)`trade insert (.z.T;`A;20.3)
,0
```

We can see that the sym column in the `trade` table is indeed an enumerated list with respect to the keyed table:

```
q)exec sym from trade
`financials$,`A
```

Technical note: the first enumeration defined in a database will have type 20h, with each additional enumeration iterating this value by 1 up to a maximum of 76h; in effect 57 is the maximum number of enumerations/foreign keys that may exist in a single database or the session will throw an `'elim` *error*.

When we inserted a row into the `trade` table above, kdb+ performed a lookup on the keyed table to see what row of the table this entry will map to (in this case row 0) and rather than placing the sym into the table directly, a pointer to this key value was inserted instead. Given that this entry is a reference and not a value any alterations to the referenced key column will directly influence the trade table and indeed any other tables referencing this key. It is therefore vitally important that great care be taken

when managing data that is being referenced elsewhere since modifying, rearranging or deleting this data will have unwanted knock-on effects:

```
q)delete from `financials where sym=`A
`financials
//trade table still has a reference to the sym entry in row 0 which is
//now 'B'
q)trade
time         sym price
----------------------
09:06:24.849 B   20.3
```

On the other hand - as is the case with a regular enumerated list - if an enumeration attempt is made where the referencing column value does not exist then the lookup fails and a cast error is returned:

```
q)`trade insert (.z.T;`D;12.1)
'cast
```

Inserting the relevant mapping data into the keyed table will fix this problem:

```
q)`financials insert (`D;1.3;4.0)
,2
q)`trade insert (.z.T;`D;12.1)
,1
```

The above example demonstrates a key benefit of using foreign keys, it ensures that trade data will always have relevant referential data available for queries and lookups, thus identifying missing or corrupt data and improving data integrity. Another benefit is that since we have created a link from the sym column in the `trade` table to various rows in the `financials` table, we can use this mapping to reference other rows as well using dot notation, just as if the referenced table columns were in the original:

```
//Display the Price-Earnings and Price-Book ratios by sym
q)select priceEarningsRatio:last
price%sym.earningsPerShare,priceBookRatio:last price%sym.
bookValPerShare by sym from trade

//Continued overleaf


sym| priceEarningsRatio priceBookRatio
```

```
---| --------------------------------
B  | 8.826087           8.12
D  | 9.307692           3.025
```

Similar to the case of kdb+ column attributes only a single foreign key can be referenced by a column at any one time; establishing a second foreign key will automatically delete the link to the first. Links to more than one table using a single column may be created by linking table 1 to table 2, table 2 to table 3 and so forth:

```
//Create a new table holding exchange information
q)exchange:([id:101 102 103 104];ex:`LSE`NDQ`NYSE`AMEX)
q)update exchangeID:`exchange$101 101 102 from `financials
`financials

//Compound dot notation
q)select time,sym,sym.exchangeID.ex from trade
time         sym ex
--------------------
09:06:24.849 B   LSE
09:07:44.282 D   NDQ
```

2.2 Compound Foreign Keys

A foreign key link across two or more columns is possible in kdb+. In this case to allow the usage of dot notation an extra column is appended to the referencing table storing the index link of the table being referenced:

```
q)t1:([sym:`A`B`C;ex:`NYSE`NYSE`NDQ];sharesInIssue:3?1000)
q)t2:([]time:2?.z.T;sym:`A`B;exchange:`NYSE`NYSE;price:2?10.)

//Append columns together using each-both
q)update t1fkey:`t1$(t2[`sym],'t2[`exchange]) from `t2

q)t2
time         sym exchange price    t1fkey
----------------------------------------
02:31:39.330 A   NYSE     7.043314 0
04:25:17.604 B   NYSE     9.441671 1

// Continued overleaf

q)select sym, marketCap:price*t1fkey.sharesInIssue from t2
sym marketCap
```

```
-------------
A   401.0
B   880.5
```

All future inserts into `t2` must enumerate across `t1` as below to avoid an error:

```
q)`t2 insert (.z.T;`C;`NDQ;4.05;`t1$`C`NDQ)
,2
q)t2
time         sym exchange price    t1fkey
-----------------------------------------
02:31:39.330 A   NYSE     7.043314 0
04:25:17.604 B   NYSE     9.441671 1
20:08:25.689 C   NDQ      4.05     2
```

Alternatively, a complex foreign key may be initialised along with the table itself; the following notation is required:

```
q) t2:([]time:`time$();sym:`$();exchange:`$();price:`float$();
t1fkey:`t1$())
q) `t2 insert (.z.T;`C;`NDQ;4.05;`t1$`C`NDQ)
,0
```

Note that since the enumerated column stores the row index lookup value and not the actual value the column type is converted to an integer and not a symbol list; once again all inserts must enumerate the foreign key values:

```
q)meta t2
c     | t f  a
------| ------
time  | t
t1fkey| i t1
price | f

q)t2
time         sym exchange price t1fkey
-------------------------------------
09:17:22.771 C   NDQ      4.05  2
```

2.3 Removing Foreign Keys

To remove a foreign key from a table for simple foreign keys the keyword `value` is used:

```
q)update sym:value sym from `trade
`trade
```

If a table has a large number of foreign keys then the following function may be used which looks up the index of each column containing a foreign key and applies the `value` function to each one:

```
q)removeKeys:{[x]
   v[i]:value each (v:value flip x)i:where not null(0!meta x)`f;
   flip (cols x)!v
 }

q)meta removeKeys t2
c     | t f a
------| -----
time  | t
t1fkey| i
price | f
```

Calling the `value` function on a complex foreign key column will remove the table mapping but will leave the previously enumerated column intact as a list of integers.

2.4 Contrasting Foreign Keys and Joins

We start with a fresh `trade` table and another table, `exInfo`, which maps each symbol to its traded exchange:

```
q)trade:([]time:`time$();sym:`$();price:`float$();size:`int$());
q)exInfo:([sym:`$()]exID:`int$();exSym:`$();location:`$());

//Once million row entries
q)n:10000000;

//Continued overleaf



//Start and end time
```

```
q)st:08:00:00.000;
q)et:17:00:00.000;

//100 random syms of length 3
q)syms:-100?`3;

//Exchange information
q)exdata:(syms;count[syms]#101 102 103
104;count[syms]#`LSE`NDQ`HKSE`TSE;count[syms]#`GB`US`HK`JP)
q)insert[`exInfo;exdata];

//Trade data
q)tdata:(asc st+n?et-st;n?syms;n?100f;n?1000);
q)insert[`trade;tdata];
```

Simple select statements from the database take much longer using a left join and require more memory since the table mappings must be built up from scratch and the entire lookup table must be expanded to match the length of the source table prior to the output columns being specified:

```
q)\ts select time,sym,exSym from trade lj exInfo
68 469762928

q)update sym:`exInfo$sym from `trade

q)\ts select time,sym,sym.exSym from trade
35 268436016
```

The difference above is even more evident in higher dimensions:

```
//Remove the existing foreign key from the trade table and add the
//exchange ID for joining across two columns
q)update sym:value sym from `trade
q) update exID:exInfo[;`exID] each sym from `trade

//Re-key exInfo to key on exchange ID as well as sym
q)exInfo:`sym`exID xkey 0!exInfo

// Continued overleaf
//Left join on two columns, takes almost twenty times longer
q)\ts select time,sym,exSym from trade lj exInfo
```

```
1324 402654032

//Now create a complex foreign key
q) update exfKey:`exInfo$(trade[`sym],'trade[`exID]) from `trade

//Same results as above in simple case
q)\ts select time,sym,exfKey.exSym from trade
36 268436016
```

3 LINKED COLUMNS

In the previous section we saw how foreign keys are established by enumerating across a key column; in kdb+ it is also possible to avoid the necessity of using key columns/enumerations and link two or more columns together, allowing all tables involved to be readily splayed to disk if desired. In general links may be applied to two or more tables whether the tables are in memory, splayed on disk or even in different kdb+ databases; we will consider all three scenarios here.

3.1 Simple Linked Columns

Taking the table of financial data as before and a table of equity position data, with the `financials` table remaining unkeyed we can create a mapping similar to that in the case of complex foreign keys, that is by creating an index of integers that are used as a lookup. We use the bang symbol operator (`!`) to establish the connection once we have mapped each row in the referencing table (`equityPositions`) to the corresponding row number in the referenced table (`financials`):

```
q)equityPositions:([]sym:5#`A`B`C`D`E;size:5?10000;mtm:5?2.)
//Look up where the entries in the symbol column correspond to the
//rows in the now unkeyed 'financials' table, then store the
//references in the column 'finLink'
q) financials:0!financials
q)update finLink:`financials!financials.sym?sym from `equityPositions
```

Similarly to before the `finLink` column in the `equityPositions` table is identified as a foreign key to the `financials` table within the table metadata (even though it is not *strictly* a foreign key as before) and `select/exec/update/delete` statements incorporating dot notation may again be utilised. Appending additional rows to the `equityPositions` table must maintain the link to the `financials` table by providing the `finLink` column with the row index in the `financials` table that will be mapped to in each case. In contrast to a foreign key mapping no enumeration is present and there are therefore no restrictions on what row numbers are inserted:

```
q)`equityPositions insert (`A;200;2.;`financials!0)
,5
q)equityPositions
sym    size mtm        finLink
------------------------------
A      6927 1.266082   0
B      3700 1.150539   1
C      5588 0.01802349 2
D      5607 0.2896114  3
E      1666 1.541226   3
A      200  2          0

//Insert a sym not in the financials table, link to column 0
q)`equityPositions insert (`S;200;2.;`financials!0)
,6

//Insert the same sym again, link to an index not in the financials
//table
q)`equityPositions insert (`S;200;2.;`financials!6)
,7

q)equityPositions
sym    size mtm        finLink
------------------------------
A      6927 1.266082   0
B      3700 1.150539   1
C      5588 0.01802349 2
D      5607 0.2896114  3
E      1666 1.541226   3
A      200  2          0
S      200  2          0
S      200  2          6

//Unmapped data results in missing entries
q)select sym,finLink.earningsPerShare from equityPositions
sym earningsPerShare
--------------------
A
B   2.3
C   1.5

//Continued overleaf
D   1.3
```

```
E
A    2.3
S    2.3
S
```

3.2 Simple linked columns on disk

It is possible to create linked columns on tables that have already been splayed to disk:

```
//Create two tables and splay to disk

q)companyInfo:([] sym:`a`b`c`d;exchange:`NYSE`NDQ`NYSE`TSE;
sector:4?("Banking";"Retail";"Food Producers");
MarketCap:30000000+4?1000000);

`:db/companyInfo/ set .Q.en[`:db] companyInfo

`:db/companyInfo/

q) t:([]sym:`a`b`c`a`b;ex:`NYSE`NDQ`LSE`NYSE`NDQ;price:5?100.);

q) `:db/t/ set .Q.en[`:db] t

`:db/t/

//Create a new column in 't' linking to 'companyInfo' via the sym
//column

q)`:db/t/cLink set `companyInfo!(companyInfo`sym)?(t`sym)
`:db/t/cLink

//Update the .d file on disk so that it picks up the new column

q).[`:db/t/.d;();,;`cLink]

`:db/t/.d

q)get `:db/t/.d

`sym`ex`price`cLink



// Continued overleaf


//Load the table to update the changes in memory
```

```
q)\l db/t

//Sample query

q)select sym,cLink.sector,cLink.MarketCap from t

sym sector    MarketCap

----------------------

a   "Retail"  30886470

b   "Banking" 30230906

c   "Retail"  30352036

a   "Retail"  30886470
```

3.3 Compound linked columns on disk

Only a small adjustment to the single column case is required to link tables together based on multiple columns. We demonstrate this by continuing with the above tables:

```
//We need to reload companyInfo otherwise the link example below will
//not execute properly since the enumerated sym columns from the
//splayed table 't' have type 20h rather than 11h

\l db/companyInfo

//Initiate the mapping by flipping the columns to lists and searching
//on each sym/exchange combination

q) `:db/t/cLink2 set `companyInfo!(flip
companyInfo`sym`exchange)?(flip t`sym`ex)

`:db/t/cLink2

//Update the .d file and reload the table once again

q).[`:db/t/.d;();,;`cLink2]

q)\l db/t

// Continued overleaf
```

```
//Sample query, double link means sym 'c' does not map this time

q)select sym,cLink2.sector,cLink2.MarketCap from t

sym sector    MarketCap

----------------------

a   "Banking" 30450974

b   "Retail"  30909716

c   ""

a   "Banking" 30450974

b   "Retail"  30909716
```

3.4 Linking across multiple kdb+ databases

For practical purposes kdb+ only allows a single on-disk database to be memory-mapped to each process at any one time. Occasionally however it may be necessary to perform analytics on data in several databases simultaneously and although it is possible to aggregate data from many locations to one centralised location via IPC if the datasets in question are very large and span multiple days and weeks then this is rendered impractical. In UNIX based operating systems such as Linux and Mac OSX an alternative is to use symbolic links in conjunction with kdb+ linked columns to allow us to retrieve and analyse vast amounts of data while keeping the level of RAM usage to an acceptable level.

The following section will outline how to construct a link from a `trade` table in one partitioned database to a `quote` table in a separate database existing on the same file network; the method may easily be generalised to link between an arbitrary number of tables across an arbitrary number of databases. There are three main steps:

1) Initially we will create a partitioned database containing the tables we will be working with across multiple dates along with some utility functions for creating the database links.
2) We then map the entries in the `trade` table to those in the `quote` table for each date using a standard `asof` join across time and sym.
3) Lastly, we append a column linking the `trade` table to the `quote` table based on this mapping and save this to disk.

We utilise slightly modified versions of the standard `.Q.en` and `.Q.dpft` functions to ensure that no sym file clashes occur across the two databases. We will place these functions, and the other utility functions we will define, into the linked column namespace .lc.

The following code defines the `trade` and `quote` tables and writes them to disk in databases `db1` and `db2` respectively:

```
//Table schemas, being with fresh trade and quote schemas
q)trade:([]time:`time$();sym:`$();price:`float$();size:`int$());
q)quote:([]time:`time$();sym:`$();bid:`float$();bsize:`int$();ask:`flo
at$();asize:`int$());

//Number of entries in trade table
q)n:10000;

//Start and end of day
q)st:08:00:00.000;
q)et:17:00:00.000;
q)syms:`A`B`C`D;

q)tdata:(asc st+n?et-st;n?syms;n?100f;n?1000);
q)insert[`trade;tdata];

//Generate 10x number of quotes
q)n*:10;
q)qdata:(asc st+n?et-st;n?syms;n?100f;n?1000;n?100f;n?1000);
q)insert[`quote;qdata];

//Historical database builder function
q)buildHDB:{[dir;dt;t] .Q.dpft[hsym `$dir;dt;`sym;t];};

//Partition the tables to disk
q)buildHDB["/root/db1";;`trade] each .z.D-til 3;
q)buildHDB["/root/db2";;`quote] each .z.D-til 3;
```

The first utility function creates a symlink in a directory `basePath` to a table `rTab` that exists in a remote directory `remotePath`. The symlink name will be the same name as `rTab`. For our purposes `basePath` is the path to each `trade` table, `remotePath` the path to each `quote` table and `rTab` is the `quote` table name. All parameters are passed as strings:

```
//Check if the symlink exists and use Unix command 'ln -s' to create
//the symlink if not
q).lc.createSymLink:{[basePath;remotePath;rTab]
  remoteTablePath:remotePath,"/",rTab;
  baseTablePath:basePath,"/",rTab;

  if[not(`$rTab) in key hsym `$basePath;system "ln -s
  ",remoteTablePath," ",baseTablePath];
  };
```

In order to avoid sym file clashes when loading tables from different kdb+ databases into memory we must save the trade table down using an alternative sym file name. The following are custom versions of `.Q.en` and `.Q.dpft` that take an additional parameter for saving splayed tables to disk using a bespoke sym file name instead of the default 'sym' name:

```
//d is the database directory the table will be saved down to
//a is the name of the alternative sym file name used to avoid clashes
//p is the database partition slice
//f is the table partition field
//t is the table name

q).lc.dpft:{[d;a;p;f;t]
  if[not all .Q.qm each r:flip .lc.en[d;a]`. t;'`unmappable];
  {[d;t;i;x]
      @[d;x;:;t[x]i]
  }[d:.Q.par[d;p;t];r;iasc r f] each key r;
  @[;f;`p#]@[d;`.d;:;f,(r:key r) except f];  };

//The following function is called above when splaying the table
q).lc.en:{[d;a;x]
   if[not -11h=type a;'`$"expected symbol parameter type for a"];
   @[x; cs@where 11h=type each x cs:key flip x; (` sv (hsym d),a)?]
   };
```

The next utility function maps the entries in the `trade` table to those in the quote table using an `asof` join, constructs a link between the two tables and saves to disk using the modified `.lc.dpft` function defined above. The `asof` join columns (usually `time` and `sym`) are passed in as a symbol list whereas the file path and table names are passed in as strings:

```
q).lc.joinSaveTables:{[ajCols;basePath;dt;baseTable;remoteTable]
 //Cast table names to syms for convenience
 remoteTable:`$remoteTable;
 baseTable:`$baseTable;
 //Force load the asof join columns from remote table into memory
 remoteFileHandle:` sv (hsym `$basePath),(`$string dt),remoteTable;
 remoteTable set select sym,time from (get remoteFileHandle);

 //Re-apply attributes of original table to the in-memory copy
 ![remoteTable;();0b;a[`c]!{(#;enlist x;y)} .' flip value a:exec a,c
 from meta get remoteFileHandle where c in ajCols];

 //Join tables and set the link column to be the point at which the
 //tables map together
 baseTable set aj[ajCols; value baseTable; ?[value
 remoteTable;();0b;(ajCols!ajCols),(enlist `id)!enlist `i]];
 update link:remoteTable!(exec i from select i from value
 remoteTable)?id from baseTable;
 // Splay base table to disk but use different, independent sym file
 //`tsym
 .lc.dpft[hsym `$basePath;`tsym;dt;`sym;baseTable]; };
```

Now that we have all the prerequisite utility functions defined, the following master function creates a symlink from the `trade` table directory to the `quote` table for each partition slice in question:

```
q).lc.createPart:{[basePath;baseTable;remotePath;
   remoteTable;ajCols;dt]
   .lc.createSymLink[raze basePath,"/",string dt;
   raze remotePath,"/",string dt;remoteTable];
   .lc.joinSaveTables[ajCols;basePath;dt;baseTable;remoteTable];
   };
```

If we create the partitioned databases and run something like the following a link will be created across all database partitions. For simplicity, the absolute directory paths are passed to the function as parameters and hence used when defining the symlink; the above example may accommodate relative directory paths with a little manipulation.

```
q).lc.createPart["/root/db1";"trade";"/root/db2";"quote";`sym`time;]
   each .z.D-til 3
```

Loading the database from memory, we achieve a `trade` table with an embedded link to the remote `quote` table:

```
q)\l db1
q)tables[]
`quote`trade

q)meta trade
c    | t f     a
-----| ---------
date | d
sym  | s       p
time | t
price| f
size | i
id   | j
link | i quote
```

Aggregations may be carried out using a single table; queries will be very efficient especially if repeated due to caching:

```
// First run
q)\ts res:select size wavg price,bsize wavg bid,asize wavg ask by
sym,10 xbar time.minute from select
time,sym,size,price,link.ask,link.asize,link.bid,link.bsize from trade
where date=max date
3030 1117552

// Second run
q)\ts res:select size wavg price,bsize wavg bid,asize wavg ask by
sym,10 xbar time.minute from select
time,sym,size,price,link.ask,link.asize,link.bid,link.bsize from trade
where date=max date
9 1116656

q)res
sym minute| price    bid      ask
----------| -------------------------
A   08:00 | 50.87719 49.76909 48.63532
A   08:10 | 48.9346  49.99889 52.01281
A   08:20 | 47.48985 49.68657 53.01129
A   08:30 | 50.03407 51.82779 47.96814
…
```

4 CONCLUSION

This whitepaper introduced how foreign keys and linked columns may be established and applied in kdb+ databases. As we observed, the benefits of using foreign keys are numerous; firstly, establishing a permanent link between tables is considerably more efficient than building up a relationship in real time through the use of a join, particularly if queries will be repeated regularly. Furthermore, enumerating columns ensures data integrity and helps identify missing or corrupt referential data. Database normalisation is much easier to achieve allowing greater data consistency, reduction of redundant data and more flexible database design.

A drawback to using foreign keys is that keyed tables cannot be splayed to disk; this is circumvented using linked columns that can establish permanent mappings between tables whether they are both in memory or on disk. Although not featuring the benefits of enumeration, linked columns are useful for establishing mappings between tables in large-scale historical databases, allowing users to either map data within partition slices in a single database or map each table in a particular partition slice in one database to the corresponding partition slice in another.