it's about time

Developer Brief

A Natural Query Interface for Distributed Systems

Author:

Sean Keevey, who joined First Derivatives in 2011, is a kdb+ consultant and has developed data and analytic systems for some of the world's largest financial institutions. Sean is currently based in London developing a wide range of tailored analytic, reporting and data solutions in a major investment bank.

TABLE OF CONTENTS

1. INTRODUCTION

Technical constraints mean that large real-time database installations tend to be distributed over many processes and machines. In an environment where end-users need to be able to query the data directly, this poses a challenge - they must find the location of the data and connect before they can query it. Typically this inconvenience is mitigated by providing an API. This can take the form of a function which can be used to query tables in the data warehouse.

This paper briefly explores one alternative method of eliminating this burden, envisioning the use case of a user sitting at a desk, entering queries in real-time, for whom using a traditional API may be cumbersome and unnatural.

The idea is to intercept queries as they are entered, dissect them to identify references to remote data and to seamlessly return a result as if all the data were available in the immediate process.

Implementing this idea in full generality would rely on being able to solve a difficult problem - to identify a user's intent. We narrow the scope of the solution by only attempting to redirect q-sql queries which query datasets specified in configuration files.

The solution will use an undocumented feature of the q interpreter which allows a handler for a custom language to be defined. We will define a handler which will parse q-sql queries. It will determine which elements of a query refer to tables on other processes and execute these elements of the query remotely as appropriate. Certain internal kdb+ optimisations (in particular join optimisations) may be lost when using the implementation described in this paper.

All tests were run using kdb+ version 3.2 (2014.10.04).

2. IMPLEMENTATION

2.1. Background

Before describing the implementation, it is worth noting two important caveats:

1. The implementation uses a single-letter namespace. These are reserved for use by Kx and should not be used in a production system.

2. The implementation uses an undocumented feature of kdb+ which allows a handler for a custom language to be defined. This feature is not guaranteed to be present in future kdb+ versions.

A handler for standard SQL is distributed with q in a file called s.k. This allows queries to be entered in a q process in a more traditional SQL as follows:

```
s)SELECT field1, field2 FROM table WHERE date BETWEEN …
```

Important to note is the prefixed `s)` which tells q to evaluate the query using the handler defined in s.k. By examining this file, it is possible to gain an understanding of the interpreter feature to define custom language handlers. They must reside in single-letter namespaces. The key function in s.k is `.s.e` which is passed anything preceded by `s)` as a string. We will define our handler similarly, taking the namespace '`.H`'. Queries will be entered thus:

```
H)select from q where sym=`ABC, time within 12:00:00.000 13:00:00.000
```

This query is passed to `.H.e` by the interpreter. By breaking the query down from here, we can define custom behaviours where desired for certain aspects of the query.

The `parse` function takes a string, parses it as a q expression and returns the parse-tree. We will use this to break down the query entered by a user into a format where it is easy to identify the constituent elements. This will be useful for identifying where a user intends to query a remote dataset.

You can inspect how the parse statement breaks down a query as follows:

```
q)parse"(select from t where ex=`N) lj (select clo:last mid by sym
from q)"

k){.Q.ft[,\:[;y];x]} //this is the definition of lj
(?;`t;,,(=;`ex;,`N);0b;()) //first select statement
(?;`q;();(,`sym)!,`sym;(,`clo)!,(last;`mid)) //second select
```

2.2.H.q

We will use the tools discussed above to provide a simple implementation of the idea described at the outset. Our handler will scan q statements for queries to remote datasets and evaluate them as specified in the configuration.

For the purposes of this paper, configuration will be defined in a simple table on the local instance. It serves to provide a mapping from table names (or aliases) to handles of processes where the data is available. In a more formal production environment this would be slightly more involved with centrally controlled configuration.

```
.H.H:([alias:`trade`quote`traders]host:`:localhost:29001`:localhost:2
9002`:localhost:29001;name:`t`q`traders; handle:3#0N);
//open handle to each distinct process
update handle:.Q.fu[hopen each] host from `.H.H;
//utilities to lookup handle or table-name for a given alias
.H.h:{.H.H[x][`handle]};
.H.n:{.H.H[x][`name]};
```

`select`/`exec` operations can be 4, 5 or 6-adic `?[...]` functions, the fifth and sixth arguments being seldom used, while `update`/`delete` operations are 4-adic `![...]` functions.

Additionally, queries on remote tables must match a table alias in our configuration.

With these criteria, we can define functions to identify these operations in a parse tree.

```
//check if subject of select/exec is configured as a remote table
.H.is_configured_remote:{$[(1 = count x 1)and(11h = abs type x 1);not
null .H.h first x 1;0b]};
//check valence and first list element to determine function
.H.is_select:{(count[x] in 5 6 7) and (?)~first x};
.H.is_update:{(count[x]=5) and (!)~first x};
```

We combine these for convenience:

```
.H.is_remote_exec:{$[.H.is_select[x] or
.H.is_update[x];.H.is_configured_remote[x];0b]};
```

To evaluate a functional query remotely, we define a function which will take the parse tree, look up the correct handle and table name for the remote process and evaluate accordingly.

```
.H.remote_evaluate:{(.H.h x 1)@(eval;@[x;1;.H.n])};
```

We can define a pair of functions which, together, will take a parse tree and traverse it, inspecting each element and deciding whether to evaluate remotely.

```
.H.E:{$[.H.is_remote_exec x;.H.E_remote x;1=count x;x;.z.s'[x]]};

.H.E_remote:{
    //need to examine for subqueries
    r:.H.remote_evaluate{$[(0h~type x)and not .H.is_remote_exec
x;.z.s'[x];.H.is_remote_exec x;.H.E_remote x;x]}'[x];
    //need special handling for symbols so that they aren't
    //interpreted as references by name
    $[11h=abs type r;enlist r;r]};
```

`.H.E` recursively scans a parse tree and identifies any queries which need to be remotely evaluated. These are then passed to `.H.E_remote`.

In `.H.E_remote`, we again scan the parse tree looking for remote queries. This is to identify any sub-queries that need to be evaluated remotely, e.g. `select from X where sym in exec distinct sym from Y`, where `X` and `Y` are located on two separate remote processes. In this way, we iteratively create a new parse tree, where all the remote queries in the original parse tree have been replaced with their values, as evaluated via IPC.

Next, a function which will parse a query, apply the above function and finally evaluate what remains:

```
.H.evaluate:{eval .H.E parse x};
```

These functions are recursive which comes with the usual caveats about stack space. This should only be a concern for the most extreme one-liners.

All that remains is to define the key '`.e`' function, the point of entry for queries entered with a `H)` prefix in the q interpreter. For brevity, this will simply add a layer of error trapping around the evaluation function defined above.

```
.H.e:{@[.H.evaluate;x;{'"H-err - ",x}]};
```

In the next section we will use the above to demonstrate seamless querying of tables from remote processes within a q session.

3. EXAMPLE

It is easy to demonstrate where this idea may be useful. Consider a simple case where there is a quote table on one rdb process and a trade table on another along with a supplementary meta-data.

We use the following files to initialise q processes with a quote, trade and traders table, populated with some sample data:

3.1. quote.q

```
\p 29002
//set random seed
\S 1

rnorm:{$[x=2*n:x div 2;raze sqrt[-2*log n?1f]*/:(sin;cos)@\:(2*acos -
1)*n?1f;-1_.z.s 1+x]};
q:([]time:asc
1000?01:00:00.000000000;sym:`g#1000?`ABC`DEF`GHI;bsize:1000*1+1000?10
;bid:1000#0N;ask:1000#0N;asize:1000*1+1000?10);

//simulate bids as independent random walks
update bid:abs rand[100f]+sums rnorm[count i] by sym from `q;

//asks vary above bids
update ask:bid + count[i]?0.5 from `q;
```

3.2. trade.q

```
\p 29001
\S 2

traders:([tid:til 4];name:("M Minderbinder";"J Yossarian";"C
Cathcart";"M M M Major"));
t:([]time:asc
100?00:00:00.000000000;sym:`g#100?`ABC`DEF`GHI;side:100?"BS";qty:100?
1000;tid:100?til 4);
```

3.3. Gateway

Then, we start up our gateway process, using the code above in Section 2.2. This opens the necessary connections to the remote instances and initialises the H query-handler.

From there we can explore the data:

```
q)trade
'trade
q)quote
'quote
```

Tables ‘trade’ and ‘quote’ do not exist in the immediate process. However, with the right prefix, we can query them:

```
q)H)select from trade
time                 sym side qty tid
-------------------------------------
0D00:45:55.800542235 GHI B    292 0
0D00:57:46.315256059 ABC S    24  1
0D01:03:35.359731763 DEF S    795 2
0D01:05:44.183354079 ABC S    660 2
0D01:09:37.164588868 DEF S    434 3
..
```

And on to something slightly more complicated:

```
q)H)select sum qty by sym, side from trade where sym in exec -2#sym
from quote
sym side| qty
--------| -----
DEF B   | 4552
DEF S   | 5425
GHI B   | 6361
GHI S   | 17095
```

Joins also work:

```
q)H)aj[`sym`time;select time, sym, ?[side="B";qty;neg qty] from
trade;select time, sym, bsize, bid, ask, asize from quote]
time                 sym qty  bsize bid      ask      asize
-----------------------------------------------------------
0D00:45:55.800542235 GHI 292  7000  11.50185 11.82094 5000
0D00:57:46.315256059 ABC -24  10000 81.01584 81.19805 1000
0D01:03:35.359731763 DEF -795 7000  45.43002 45.57759 7000
0D01:05:44.183354079 ABC -660 3000  81.71235 81.75569 4000
0D01:09:37.164588868 DEF -434 7000  45.43002 45.57759 7000
..
```

```
q)H)(select from trade)lj(select from traders)
time                 sym side qty tid name
-----------------------------------------------------
0D00:45:55.800542235 GHI B    292 0   "M Minderbinder"
0D00:57:46.315256059 ABC S    24  1   "J Yossarian"
0D01:03:35.359731763 DEF S    795 2   "C Cathcart"
0D01:05:44.183354079 ABC S    660 2   "C Cathcart"
0D01:09:37.164588868 DEF S    434 3   "M M M Major"
..
```

And updates:

```
q)H)update name:(exec tid!name from traders)tid from trade
time                 sym side qty tid name
-----------------------------------------------------
0D00:45:55.800542235 GHI B    292 0   "M Minderbinder"
0D00:57:46.315256059 ABC S    24  1   "J Yossarian"
0D01:03:35.359731763 DEF S    795 2   "C Cathcart"
0D01:05:44.183354079 ABC S    660 2   "C Cathcart"
0D01:09:37.164588868 DEF S    434 3   "M M M Major"
..
```

4. DRAWBACKS

We lose a lot of the internal optimizations of kdb+ if accessing tables on disparate remote processes using the method described above. The interpreter is no longer able to query data intelligently.

An example of this is when performing an asof-join on an on-disk dataset. Typically, one uses a minimal where-clause for the second table parameter of an asof-join - only a virtual column clause if querying off disk to map the data into memory:

```
q)aj[`sym`time;
select from t where date = .z.d, sym=`ABC …;
select from q where date = .z.d]
```

However, since we are accessing tables on remote processes, we are unable to take advantage of the optimizations of the `aj` built-in on mapped data. Therefore, we should behave as we would when typically querying a remote table and only select what is necessary, in order to minimise IPC overhead.

```
H)aj[`sym`time;
select from t where date = .z.d, sym=`ABC, …
select from q where date=.z.d, sym=`ABC, …]
```

Even if all the tables referenced in a query are on the same process, the interface isn't smart enough (in its current incarnation) to pass them as one logical unit to the remote process.

The scope for complexity in queries is theoretically unbounded. It is difficult to test the many variants of nested queries one may expect to see. Testing for the above has been limited but it serves as an illustration of an idea.

Some examples of where it will fail or may give unexpected results:

- Intended references to 'locals' within queries sent to remote processes - these will be evaluated on the remote process
- Foreign keys

However, with awareness of these drawbacks, there is no reason an interface such as the above cannot be used to facilitate analysis and compilation of data.

5. CONCLUSION

We have explored the idea set out above to build a simple proof-of-concept of a method of mitigating the overhead for end-users of having to locate data which they wish to access. Functionally, it behaves the same as a traditional API. Where it differs is providing a more natural, seamless experience for querying data directly.

This solution is clearly not appropriate for high-performance, large-scale querying of big data sets. However it may suit data-analysts as a form of gateway to a large distributed database.

A similar layer could easily sit on top of an existing API to give developers an easier means of interacting with raw data and to facilitate rapid prototyping of algorithms and procedures for business processes.

It could be integrated with a load balancer, a permissions arbiter, a logging framework or a managed cache as part of an enterprise infrastructure for providing access to data stored in kdb+.

All tests were run using kdb+ version 3.2 (2014.10.04).