it's about time

Technical Whitepaper

Kdb+ Query Scaling

Author:

Ian Lester is a Financial Engineer who has worked as a consultant for some of the world's largest financial institutions. Based in New York, Ian is currently working on a trading application at a US investment bank.

Table of Contents

INTRODUCTION

Trading volumes in financial market exchanges have increased significantly in recent years and as a result it has become increasingly important for financial institutions to invest in the best technologies and to ensure that they are used to their full potential.

This whitepaper will examine some of the key steps which can be taken to ensure the optimal efficiency of queries being run against large kdb+ databases. In particular it will focus on q-sql statements, joins, and other constructs which provide kdb+ with a powerful framework for extracting information from increasingly large amounts of timeseries data both-in memory and on-disk. It covers key ideas and optimization solutions while also highlighting potential pitfalls.

All tests were run using kdb+ 3.1 (2013.12.27)

1. OVERVIEW OF DATA

For the purpose of analysis we have created three pairs of trade and quote tables containing equity timeseries data where one pair is fully in memory, one has been splayed on-disk and one has been partitioned by date. Each pair of tables, quote and trade, have been loaded into a namespace (.mem, .splay or .par) so that it is clear which table we are looking at during our analysis. The code used to create this database is available in the appendix.

The data contains 4 primary symbols (AAPL, GOOG, IBM and MSFT) along with 96 randomly generated others. The in-memory quote table contains 2 million records, the splayed quote table contains 5 million records and the partitioned quote table contains 2 million records per partition over 5 days giving 10 million records in total. The trade data is 10% the size of each quote table and has been simplified by basing it off the previous quote rather than maintaining an order book.

```
q)1_count each .mem
quote| 2000000
trade| 200000


q)1_count each .splay
quote| 5000000
trade| 500000


q)1_count each .par
quote| 50000000
trade| 5000000


q)date
2014.02.19 2014.02.20 2014.02.21 2014.02.22 2014.02.23

q)select count i by date from .par.quote
date      | x
----------| --------
2014.02.19| 10000000
2014.02.20| 10000000
2014.02.21| 10000000
2014.02.22| 10000000
2014.02.23| 10000000
```

Each table has an attribute applied to the sym column - the in-memory tables being grouped (g#), and those on-disk being parted (p#). The table meta data below displays the structure of the in-memory quote and trade tables, the meta of the tables on-disk differ by attribute only.

```
q)meta .mem.quote
c  | t f a
---| -----
sym| s   g
dt | p
ap | f
as | j
bp | f
bs | j

q)meta .mem.trade
c    | t f a
----| -----
sym | s   g
dt  | p
tp  | f
ts  | j
side| s
```

2. SELECT STATEMENTS

2.1. Overview

In this section we will look at the most common and flexible method for querying kdb+ data, the q-sql select statement. This construct allows us to query both on-disk (memory mapped) and in-memory data in a similar fashion.

2.2. Retrieving records by symbol

In this section we will look at methods for retrieving the first and last records by symbol with and without further constraints added to the select statement. We will focus on the use of the efficient `select by sym from tab` construct as well as using the find `?` operator to perform a lookup for a table.

In the following example we look at how we can retrieve the last record by symbol from a table. The default behaviour of the kdb+ `by` clause is to retrieve the last value of each column grouped by each parameter in the `by` clause. If no aggregation function is specified there is no need to explicitly call the `last` function; comparing the two constructs we see a 2.5x speed improvement and similar memory usage.

```
q)select by sym from .mem.quote
sym | dt                            ap       as bp       bs
----| ----------------------------------------------------
AAPL| 2014.02.23D15:59:56.081766206 527.1077 93 527.0023 77
ACJ | 2014.02.23D15:59:58.392773419 487.661  66 487.5635 26
AGL | 2014.02.23D15:59:58.475614283 363.3669 43 363.2942 49
AHF | 2014.02.23D15:59:59.843160342 145.5018 52 145.4727 84
..

q)\ts select by sym from .mem.quote
20 16784064

q)\ts select last dt, last ap, last as, last bp, last bs by sym from
.mem.quote
51 16783328

q)(select by sym from .mem.quote) ~ select last dt, last ap, last as,
last bp, last bs by sym from .mem.quote
1b
```

We see an approximately 4x performance improvement when we apply to partitioned data:

```
q)\ts select by sym from .par.quote where date = last date
78 134226368
q)\ts select last date, last dt, last ap, last as, last bp, last bs by
sym from .par.quote where date = last date
345 201329840

q)(select by sym from .par.quote where date = last date) ~ select last
date, last dt, last ap, last as, last bp, last bs by sym from .par.quote
where date = last date
1b
```

We can use the find (`?`) operator to obtain a list of indices corresponding to the first occurrence of a symbol in our table. In our first example we obtain the position i of the first occurrence of each sym in our table, we then perform a lookup in the same table using the find operator and index into the original table with the result which will be a list of type long. Note that this example is actually redundant, we could use the simpler `select first sym, first dt, first ap, first as, first bp, first bs by sym from .mem.quote` to retrieve this data even more efficiently, but we will build on this in the rest of this section. We find this to be approximately 2x faster than using `fby` and it also uses slightly less memory:

```
q).mem.quote(select sym, i from .mem.quote)?0!select first i by sym from
.mem.quote
sym  dt                            ap       as bp       bs
-----------------------------------------------------------
AAPL 2014.02.23D09:00:00.117540266 544.0444 27 543.9356 56
ACJ  2014.02.23D09:00:01.156257558 487.5938 36 487.4963 60
AGL  2014.02.23D09:00:00.014462973 345.9409 80 345.8717 104
AHF  2014.02.23D09:00:00.466329697 138.5061 44 138.4784 63
AHK  2014.02.23D09:00:00.167160294 342.2259 97 342.1575 104
.

q)\ts .mem.quote(select sym, i from .mem.quote)?0!select first i by sym
from .mem.quote
24 33557504

q)\ts select from .mem.quote where i = (first; i) fby sym
63 42995872

q)(.mem.quote(select sym, i from .mem.quote)?0!select first i by sym
from .mem.quote) ~ select from .mem.quote where i = (first; i) fby sym
1b

q)\ts select first sym, first dt, first ap, first as, first bp, first bs
by sym from .mem.quote
12 15238560
```

In our second example we show how using the search operator does not outperform `select by sym from t` but is still an improvement over using `last`:

```
q)\ts .mem.quote(select sym, i from .mem.quote)?0!select last i by sym
from .mem.quote
32 33556528

q)\ts 0!select by sym from .mem.quote
20 16784144

q)(.mem.quote(select sym, i from .mem.quote)?0!select last i by sym from
.mem.quote) ~ 0!select by sym from .mem.quote
1b
```

Finally we demonstrate a key use case for this construct, how we can select the first occurrence of an event in one column of the table, in our example below we look at the maximum bid size by sym. Examples like this where there is no q primitive shortcut which can be applied consistently across all columns, for example the call to `first` used above, is where the performance improvements of this construct come to the fore. In the example below we will achieve 2x performance over the alternative methodology `fby`.

```
q).mem.quote(`sym`bs#.mem.quote)?0!select max bs by sym from .mem.quote
sym  dt                            ap       as  bp       bs
-------------------------------------------------------------
AAPL 2014.02.23D09:00:19.752919580 544.043  92  543.9342 109
ACJ  2014.02.23D09:00:12.435798905 487.5911 106 487.4936 109
AGL  2014.02.23D09:01:29.984035491 345.9841 79  345.915  109
AHF  2014.02.23D09:04:30.327046606 138.578  13  138.5503 109
AHK  2014.02.23D09:03:01.717110984 342.2449 12  342.1764 109
..

q)\ts  .mem.quote(`sym`bs#.mem.quote)?0!select  max  bs  by  sym  from
.mem.quote
28 16779264
```

2.3. Efficient select statements using attributes

One of the most effective ways to ensure fast lookup of data is the correct use of attributes, which has been covered previously in Edition 5 of the Technical Whitepaper series 'Columnar database and query optimization.' In this section we will look at how we can ensure that an attribute is used throughout an entire select statement.

If we wish to filter a table for entries containing a particular list of syms the simplest way of doing this is to use a statement like `select from table where sym in symList`. However, when we apply the `in` operator to a column which contains an attribute we will only receive that attribute's performance benefit for the first symbol in the list we are searching. An alternative is to rewrite the query using a lambda and pass in each symbol in turn: `{select from table where sym = x} each symList`. When we use the lambda function we get the performance improvement for every symbol in the list so even with the overhead of appending the results using the `raze` function we will often see an improvement in execution time at the cost of the extra memory needed to store the intermediate results.

In the following examples, which are run on the partitioned quote table, we see a speed increase of slightly below 2x at the cost of a larger memory footprint however as the number of symbols under consideration and the size of the data increases we would expect to see greater benefits.

```
q)raze {select from .par.quote where date = last date, sym = x}each
`AAPL`GOOG`IBM
date       sym  dt                            ap       as  bp       bs
-------------------------------------------------------------------------
2014.02.23 AAPL 2014.02.23D09:00:00.248076673 527.0607 29  526.9553 86
2014.02.23 AAPL 2014.02.23D09:00:01.054893527 527.0606 18  526.9552 85
2014.02.23 AAPL 2014.02.23D09:00:01.189490128 527.0605 15  526.9551 72
2014.02.23 AAPL 2014.02.23D09:00:01.211703848 527.0605 41  526.9551 69
2014.02.23 AAPL 2014.02.23D09:00:01.286917179 527.0606 49  526.9552 59
2014.02.23 AAPL 2014.02.23D09:00:01.546945609 527.061  100 526.9556 88
..

q)\ts raze {select from .par.quote where date = last date, sym = x}each
`AAPL`GOOG`IBM
15 44042032

q)\ts select from .par.quote where date = last date, sym in
`AAPL`GOOG`IBM
25 27264720

q)(raze {select from .par.quote where date = last date, sym = x}each
`AAPL`GOOG`IBM) ~ select from .par.quote where date = last date, sym in
`AAPL`GOOG`IBM
1b
```

The example below uses a similar construct to find the maximum ask price for each symbol from our in-memory quote table; here we achieve a 20% speed increase and a huge reduction in memory usage:

```
q)raze{select max ap by sym from .mem.quote where sym = x} each
`AAPL`GOOG`IBM
sym | ap
----| --------
AAPL| 544.0496
GOOG| 1198.86
IBM | 183.5632

q)\ts select max ap by sym from .par.quote where date = last date, sym
in `AAPL`GOOG`IBM
20 12584960

q)\ts raze {select max ap by sym from .par.quote where date = last date,
sym = x}each `AAPL`GOOG`IBM
16 2100176

q)(raze{select max ap by sym from .mem.quote where sym = x} each
`AAPL`GOOG`IBM) ~ select max ap by sym from .mem.quote where sym in
`AAPL`GOOG`IBM
1b
```

2.4. Obtaining a Subset of Columns

In cases where our goal is to obtain a subset of columns from a table we can use the take (#) operator to do this more efficiently than in a standard select statement. This potential improvement comes from recognising that a table is a list of dictionaries with its indices swapped, (position (1;2) in a table is equivalent to position (2;1) in a dictionary) and is therefore subject to dictionary operations. This is a highly efficient operation as kdb+ only has to index into the keys defined in the list to the left of #:

```
q)`sym`ap`as#.mem.quote
sym  ap       as
---------------
AAPL 544.0444 27
AAPL 544.0431 57
AAPL 544.043  77
..

q)\ts do[1000000; `sym`ap`as#.mem.quote]
687 672

q)\ts do[1000000; select sym, ap, as from .mem.quote]
819 1200

q)(`sym`ap`as#.mem.quote) ~ select sym, ap, as from .mem.quote
1b
```

The above gives a small performance increase and corresponding decrease in memory usage and will also work on splayed tables. It can be applied to keyed tables when used in conjunction with the each-right operator /:; this will return the columns passed as a list of symbol in the first argument along with the key column of the table. To illustrate this we create a table .mem.ktrade, which is the in memory trade table defined above keyed on unique GUID values.

```
// add a unique GUID tradeID primary key to in-memory trade table
q).mem.ktrade:(flip enlist[`tradeID]!enlist `u#(neg count
.mem.trade)?0Ng)!.mem.trade
q)`sym`side#/:.mem.ktrade
tradeID                              | sym  side
-------------------------------------| ---------
deaf3e2d-f9ba-6a8b-1db4-ffe76a6a3be6| AAPL Sell
986e1121-ba3c-d9a1-80f4-a884ad783444| AAPL Sell
37472984-556c-4a6b-1f2c-e7eb716144ee| AAPL Buy
6e65f67e-23e1-4ef9-cb08-9c68467f9e34| AAPL Sell
..

q)\ts do[10000; `sym`side#/:.mem.ktrade]
19 736

q)\ts do[10000; select tradeID, sym, side from .mem.ktrade]
21 1376

q)\ts do[10000; `tradeID xkey select tradeID, sym, side from
.mem.ktrade]
63 1424

q)(`tradeID xkey select tradeID, sym, side from .mem.ktrade) ~
`sym`side#/:.mem.ktrade
1b
```

As before we see a modest improvement in runtime if our goal is to return an unkeyed table, however if we want to return the data keyed, as it was in the original table, the performance improvement is approximately 3x; in both cases we cut memory usage in half.

2.5. Efficient querying of partitioned on-disk data

A basic but incredibly important consideration when operating on a large dataset is the ordering of the where clause which will be used to extract the data required from a table. An inefficiently ordered where clause that does not take advantage of the partitioned structure of a database or its attributes can cause orders of magnitude slowdown in a query.

The following code illustrates how efficient construction of a where clause in a partitioned database by filtering for the partition column first can lead to vastly superior results:

```
q)select from .par.trade where date = last date, ts > 50
date       sym  dt                              tp       ts side
-----------------------------------------------------------------
2014.02.23 AAPL 2014.02.23D09:00:15.613542722 527.0524 52 Buy
2014.02.23 AAPL 2014.02.23D09:00:33.319332489 526.9366 53 Sell
2014.02.23 AAPL 2014.02.23D09:00:42.741552220 527.0334 70 Buy
2014.02.23 AAPL 2014.02.23D09:00:45.434857198 526.9292 70 Buy
2014.02.23 AAPL 2014.02.23D09:00:50.608371516 527.0314 63 Sell

q)\ts select from .par.trade where date = last date, ts > 50
47 20973216

q)\ts select from .par.trade where ts > 50, date = last date
1822 169872240
```

The date parameter in the above example can also be modified within the select statement while still providing a performance improvement. For example we could use the `mod` function to obtain data for one particular day of the week, or a cast to get the data for an entire month. The key to applying these functions properly is ensuring that the virtual date column is the first parameter in the first element of the where clause; this can be confirmed by looking at the parse tree created by the select statement as shown below:

```
// select all the data for February 2014
q)select from .par.trade where date.month = 2014.02m
date       sym  dt                              tp       ts side
-----------------------------------------------------------------
2014.02.19 AAPL 2014.02.19D09:00:00.627678019 509.4149 18 Buy
2014.02.19 AAPL 2014.02.19D09:00:02.227734720 509.4179 42 Sell
2014.02.19 AAPL 2014.02.19D09:00:03.376288471 509.4192 19 Sell
2014.02.19 AAPL 2014.02.19D09:00:03.714012114 509.42   12 Sell
2014.02.19 AAPL 2014.02.19D09:00:13.403248267 509.4331 57 Sell
..

q)\ts select from .par.trade where date.month = 2014.02m
93 322964016

q)parse"select from .par.trade where date.month = 2014.02m "
?
`.par.trade
,,(=;`date.month;2014.02m)
0b
()
```

The parse tree above shows how a query is broken down to be executed by the q interpreter. The first element, `?`, tells us that we have a `select` or `exec` statement, the second element is the table we are selecting from, third is the `where` clause, fourth is any aggregations to be applied (`by` clause) and fifth is a list of the columns which we are selecting or an empty list if we want to return all columns. This pattern is the same as that used in the functional form of `select` or `exec`. More information on functional form can be found here. http://code.kx.com/wiki/JB:QforMortals2/queries_q_sql#Functional_Forms

The key to this query working efficiently is ensuring that `date.month` is the first element in the where clause of the parsed query.

3. JOINS

3.1. Overview

One of the most powerful aspects of kdb+ is that it comes equipped with a large variety of join operators for enriching datasets. These operators may be sub-divided into two distinct types, timeseries joins and non-timeseries joins. In this section we will focus on optimal use of three of the most commonly utilised join operators, the `aj` and `wj` (both timeseries joins) and the non-timeseries join `lj`; basic information on these and others can be found here on the KX wiki: (http://code.kx.com/wiki/Reference/joins).

The majority of joins in a kdb+ database should be performed implicitly by foreign keys and linked columns, permanent relationships which can be defined between tables in a database. These provide a performance advantage over the standard joins and should be used where appropriate, more information on implementing these database structures can be found in Edition 8 of the Technical Whitepaper series 'The application of foreign keys and linked columns in kdb+.'

3.2. Left joins

The left join (`lj`) is one of the most common, important and widely used joins in kdb+ which is used to join two tables based on the key columns of the table in the second argument. If the key columns in the second table are unique it will perform an operation similar to a left outer join in standard SQL, however if the keys in the second table are not unique it will look up the first value only. If there is a row in the first table which has no corresponding row in the second table a null record will be added in the resulting table.

The example below illustrates a simple left join which is used to add reference data to our quote table. First we define a table .mem.ref keyed on symbol with some market information as the value, the `lj` joins this market information to the .mem.trade table returning the result. In our second example we see that when there are duplicate keys in the second table only the first value corresponding to that key is used in the join.

```
q).mem.ref:([sym:exec distinct sym from .mem.trade]; mkt:100?`n`a`l)
q).mem.quote lj .mem.ref
sym  dt                            ap       as bp       bs mkt
-------------------------------------------------------------
AAPL 2014.02.23D09:00:00.117540266 544.0444 27 543.9356 56 n
AAPL 2014.02.23D09:00:00.770298577 544.0431 57 543.9343 68 n
AAPL 2014.02.23D09:00:01.014678832 544.043  77 543.9342 96 n
AAPL 2014.02.23D09:00:03.650976810 544.0455 12 543.9367 48 n
..
```

```
q).mem.ref2:update sym:`AAPL from .mem.ref1 where 0 = i mod 2
q).mem.ref2
sym | mkt
----| ---
AAPL| n
ACJ | n
AAPL| l
AHF | a
AAPL| l
..

q)select firstMkt:first mkt, lastMkt:last mkt by sym from .mem.ref2
where sym = `AAPL
sym | firstMkt lastMkt
----| ----------------
AAPL| n        l

q).mem.quote lj .mem.ref2
sym  dt                             ap       as bp       bs mkt
-------------------------------------------------------------------
AAPL 2014.02.23D09:00:00.117540266 544.0444 27 543.9356 56 n
AAPL 2014.02.23D09:00:00.770298577 544.0431 57 543.9343 68 n
AAPL 2014.02.23D09:00:01.014678832 544.043  77 543.9342 96 n
AAPL 2014.02.23D09:00:03.650976810 544.0455 12 543.9367 48 n
AAPL 2014.02.23D09:00:06.727747153 544.0471 81 543.9383 50 n
```

Since kdb+ 2.7 the left join has been built into the `dict,keyedTab` and `tab,\:keyedTab` built-ins. This has provided a significant performance improvement over the previous implementation of left join and since kdb+ 3.0 `lj` has been updated to use this form inside a `.Q.ft` wrapper to allow the table being joined to be keyed. `.Q.ft` will unkey the table, run the left join, and then rekey the result.

Between versions 2.7 and 3.0 there is an approximately 2x performance improvement when joining a keyed table to a table, and approximately 5x improvement when joining a keyed table to a dictionary using the `,\:` construct over the old implementation of `lj`.

3.3. Window joins

`wj` and `wj1` are the most general forms of timeseries joins; they provide an aggregation over all the values in specified columns for a given time interval. `wj1` differs from `wj` in that it only considers values from within the given time period whereas `wj` will also consider the current prevailing values.

In the example below `wj` calculates the minimum ask price and maximum bid price in the second table, .mem.quote based on the time windows w, in this case the values just before and after a trade is executed in .mem.trade:

```
// Define the time windows to aggregate over
q)w:-2 1+\:exec dt from .mem.trade where sym = `AAPL

q)w
2014.02.23D09:00:00.118540264 2014.02.23D09:00:22.930035590
2014.02.23D09:00:00.118540267 2014.02.23D09:00:22.930035593

// calculate the min ask price and max bp for each time window and join
to the trade table
q)wj[w; `sym`dt; select from .mem.trade where sym = `AAPL; (.mem.quote;
(min; `ap); (max; `bp))]
sym  dt                             tp       ts side ap       bp
-------------------------------------------------------------------
AAPL 2014.02.23D09:00:00.118540266 544.0444 27 Sell 544.0444 543.9356
AAPL 2014.02.23D09:00:22.930035592 544.0437 55 Sell 544.0437 543.9349
AAPL 2014.02.23D09:00:25.852858872 544.0401 88 Buy  544.0401 543.9313
AAPL 2014.02.23D09:00:35.654202142 543.9216 12 Sell 544.0304 543.9216
AAPL 2014.02.23D09:00:42.274095995 543.9134 40 Buy  544.0222 543.9134
..

q)\ts wj[w; `sym`dt; select from .mem.trade where sym = `AAPL;
(.mem.quote; (min; `ap); (max; `bp))]
9 207696

q)w:-2 1+\:.mem.trade.dt

q)\ts wj[w; `sym`dt; .mem.trade; (.mem.quote; (min; `ap); (max; `bp))]
485 14898368
```

`wj` can also be used to efficiently run aggregations on splayed or partitioned data as we can see in the example below:

```
// define an aggregation window from this trade back to the previous
one
q)w:value flip select dt^prev dt, dt from .par.trade where date = max
date, sym = `AAPL

// calculate max and min values between two trades
q)wj[w; `sym`dt; select from .par.trade where date = max date, sym =
`AAPL; (select from .par.quote where date = max date; (min; `ap); (max;
`bp))]
date       sym  dt                             tp       ts side ap
bp
-------------------------------------------------------------------------
---
2014.02.23 AAPL 2014.02.23D09:00:01.606196483 527.0609 43 Sell 527.0609
526.9555
2014.02.23 AAPL 2014.02.23D09:00:02.905704958 527.0586 12 Buy  527.0609
526.9555
2014.02.23 AAPL 2014.02.23D09:00:14.610539881 526.9486 33 Buy  527.0597
526.9543
..

q)\ts wj[w; `sym`dt; select from .par.trade where date = max date, sym
= `AAPL; (select from .par.quote where date = max date; (min; `ap);
(max; `bp))]
```

`wj` can be useful for finding `max` and `min` values in a given timeframe as shown above, however in most situations it is preferable to use an asof join or a combination of asof joins.

3.4. aj joins

`aj` and `aj0` are simpler versions of `wj`; they return the last value from the given time interval rather than the results from an arbitrary aggregation function. `aj` displays the time column from the first table in its result whereas `aj0` uses the column from the second table.

If a table has either a grouped or parted attribute on its sym column, as is the case for all of the tables in our sample database, it will likely be a good candidate for an asof join which we would expect to give constant time performance. However it is important to realise that only the attributes on the first column in an asof join will be used, therefore it is rarely a good idea to use an `aj` on more than two columns. If there is no attribute on the data being joined, or there is a need to apply extra constraints we will expect a linear runtime and a select statement will in most cases be more appropriate.

As we can see in the results below our second asof join without the attribute is four orders of magnitude slower than the first join and used an order of magnitude more memory.

```
q)meta .mem.quote
c  | t f a
---| -----
sym| s   g
dt | p
ap | f
as | j
bp | f
bs | j

q)\ts aj[`sym`dt; select from .mem.trade where sym = `AAPL; .mem.quote]
2 150848

q)update sym:reverse reverse sym from `.mem.quote
`.mem.quote

q)meta .mem.quote
c  | t f a
---| -----
sym| s
dt | p
ap | f
as | j
bp | f
bs | j

q)\ts aj[`sym`dt; select from .mem.trade where sym = `AAPL; .mem.quote]
11393 2442416
```

An asof join can also be performed directly on memory mapped data without having to read the entire files. It is important to take advantage of this by only reading the data needed into

memory rather than performing further restrictions in the where clause as this will result in a subset of the data being copied into memory and will greatly increase the runtime despite working with a smaller dataset.

```
q)`sym`dt`ap`bp`tp#aj[`sym`dt; select from .par.trade where date = max
date, sym = `AAPL; select from .par.quote where date = .z.d]
sym  dt                            ap       bp       tp
-----------------------------------------------------------------
AAPL 2014.02.23D09:00:01.606196483 527.0609 526.9555 527.0609
AAPL 2014.02.23D09:00:02.905704958 527.0586 526.9532 527.0586
AAPL 2014.02.23D09:00:14.610539881 527.054  526.9486 526.9486
AAPL 2014.02.23D09:00:15.613542722 527.0524 526.947  527.0524
AAPL 2014.02.23D09:00:25.251668544 527.0447 526.9393 526.9393
AAPL 2014.02.23D09:00:28.020791189 527.0443 526.9389 526.9389
AAPL 2014.02.23D09:00:32.284428963 527.0435 526.9381 527.0435
AAPL 2014.02.23D09:00:33.319332489 527.042  526.9366 526.9366
AAPL 2014.02.23D09:00:33.863122707 527.0424 526.937  527.0424
AAPL 2014.02.23D09:00:34.584276510 527.0417 526.9363 527.0417
..

q)\ts aj[`sym`dt; select from .par.trade where date = .z.D, sym = `AAPL;
select from .par.quote where date = max date]
24 68438016

q)\ts aj[`sym`dt; select from .par.trade where date = .z.D, sym = `AAPL;
select from .par.quote where date = max date, sym = `AAPL]
4696 8193984
```

CONCLUSION

This paper looked at how to take advantage of multiple kdb+ query structures in order to achieve optimal query performance for large volumes of data. It focused on making efficient changes to standard select statements for both on-disk and in-memory databases illustrating how they can provide both flexibility and performance within a database. It also considered joins, in particular the left join, window join and asof join which allow us to perform large scale analysis on in-memory and on-disk tables. The performance and flexibility of user queries and timeseries joins are some of the main reasons why kdb+ is such an effective tool for the analysis of large scale timeseries data. The efficient application of these tools is vital for any kdb+ enterprise system.

All tests were run using kdb+ 3.1 (2013.12.27)

APPENDIX

A) Test Data

The following script (two code boxes) can be used to create a copy of the database used for testing.

```
// create trade and quote tables (trade is 10% the size of quote)
buildTab:{[numRows; endDate; numDays]
  quote:([] sym:numRows?key priceMap; dt:asc raze (endDate - til
numDays) + 09:00:00 + (`long$numDays # numRows %
numDays)?'07:00:00.000000000; ap:raze 1?'flip (neg\) numRows?0.05; as:10
+ numRows?100);
  quote:update ap:priceMap[sym; `price] * 1 + sums (1e-7 *
`long$`time$0D00 | dt - (prev; dt) fby sym) * ?[`boolean$mod[i;
1|priceMap[sym; `vol]]; moveMap[priceMap[sym; `movement]] @' ap; {x}ap]
by sym from quote;
  quote:update ap:ap + 0.0001 * ap, bp:ap - 0.0001 * ap, bs:10 +
numRows?100 from quote; / make this more dynamic and only increment ap
or bp

  numTrades:floor numRows % 10;
  tradeSide:numTrades?`Buy`Sell;
  trade:select sym, dt:dt + min(0D00:00.001; 0D00:00.001|abs 1 + dt -
(next; dt) fby sym), tp:?[tradeSide[i] = `Buy; ap; bp], ts:min each
flip(?[tradeSide[i] = `Buy; as; bs]; 10 + numTrades?100), side:tradeSide
from quote where i in asc neg[numTrades]?i;

  // update priceMap for next run
  priceMap[; `movement]:(n+4)?`u`d`s;
  priceMap[; `price]:(select by sym from trade)'[key priceMap; `tp];

  `sym`dt xasc/:(quote; trade)
  }

// save data splayed or partitioned by date
partitionData:{[numRows; numDays; partitionType; saveDir; tblNames]
  $[partitionType = `splay;
    (` sv' saveDir,/:(` sv' `.splay,/:tblNames),\:`) set'
.Q.en'[saveDir] @'[buildTab[numRows; .z.D; numDays]; `sym; `p#];
  partitionType = `date;
  {[dt; numRows; saveDir; tblNames] numDays:(-) . `date$(1+;::) @\: dt;
(` sv' saveDir,/:(`$string dt),/:(` sv' `.par,/:tblNames),\:`) set'
.Q.en'[saveDir] @'[buildTab[numRows; dt; numDays]; `sym; `p#]}[;
numRows; saveDir; tblNames] each (partitionType$.z.D) - til numDays;
// not generalised to deal with yearly partitions
    0N!"Partition type must be either splay, date"];
  }

// load a trade and quote table either in memory or on-disk
buildDB:{[parMap; saveDir; tblNames]
  $[parMap[0] = `inMem;
    (` sv' `.mem,'tblNames) set' @'[buildTab[parMap[1]; .z.D; 1]; `sym;
`g#];
  parMap[0] in `splay`date;
    partitionData[parMap[1]; parMap[2]; parMap[0]; saveDir; tblNames];
    0N!"Currently this script only supports the creation of inMem,
splay, and date partitioned data"];
  }
```

```
n:96; // 4 minus the total number of syms

// current prices, direction of movement of equity and volatility
definitions used in data creation
priceMap:(`GOOG`MSFT`IBM`AAPL,upper neg[n]?`3)!([] price:1198.74 37.63
183.544 543.99,10+n?500f; movement:(n+4)?`u`d`s; vol:1+(n+4)?7);
moveMap:`u`d`s!({abs x}; {neg abs x}; {x});


// create the database with 2 million rows in memory, 5 million splayed
and 10 million partitioned (split over 5 days)
buildDB[; saveDir:`:/home/ian/data; `quote`trade] @' flip
(`inMem`splay`date; 2000000 5000000 10000000; 1 1 5)
```

B) q Versions of timeseries joins

```
// for every record in t1 left join t2 updating any new columns with the
prevailing value at time t1.dt by sym
// @param x list of columns to join on (should be sym and time)
// @param y table to be joined to
// @param z table to join to y
// @return one record for each row in y and the last item by row order
from z matching the non-time columns in x or null if there are no
matching records
.ian.aj:{.ian.ft[{d:x _z; $[all j:-1 < i:(x#z) bin x#y; y,'d i; flip
.[flip .ian.ff[y]d; (key flip d; j); :; value flip d i j:where
j]]}[x,(); ; 0!z]]y}

// if y is a keyed table return function x on unkeyed version of y then
reapply keys
// otherwise run function x on y
// @param x function
// @param y table (keyed otherwise function does nothing)
// @return function x run on y
.ian.ft:{$[$[99h = type t:.ian.v y; 98h = type get t; 0]; [n:count flip
key y; n!x 0!y]; x y]}

// if input is an hsym load/map from diak otherwise return input
// @param x table or tablename
// @return x table or memory mapped table if stored on disk
.ian.v:{$[-11h = type x; get $[":" = first t:string x; `$t,"/"; x]; x]}

// add columns y to table x inititalising values to null
// @param x table
// @param y columns to add
// @return table with new null columns
.ian.ff:{$[(&/) (key flip y) in f:key flip x; x; x,'(f_y) (count x)#0N]}
```