Dictionaries

! Dict	make a dictionary	key	key list
group	group list by values	value	value list

Lists and dictionaries

A list is a mapping from its indexes to its items: `v:1040 59 27` maps

```
0 -> 1040
1 -> 59
2 -> 27
```

A dictionary is a mapping from a list of keys to a list of values.

```
q)show d:`tom`dick`harry!1040 59 27
tom  | 1040
dick | 59
harry| 27
```

The indexes of `v` are `0 1 2`. The indexes of `d` are `` `tom`dick`harry ``.

The values of `v` and `d` are the same.

```
q)value d
1040 59 27
q)value `v
1040 59 27
```

Construction

Use Dict to make a dictionary from a list of keys and a list of values.

```
q)show d:`a`b`c!1 2 3
a| 1
b| 2
c| 3
```

The lists must be the same length. The keys should be unique (no duplicates) but no error is signalled if duplicates are present.

Avoid duplicating keys in a dictionary or (column names in a) table.

Q does not reject duplicate keys, but operations on dictionaries and tables with duplicate keys are **undefined**.

If you know the keys are unique you can set the `u` attribute on them.

```
(`u#`a`b`c)!100 200 300
```

The dictionary will then function as a hash table – and indexing will be faster.

Set Attribute

Items of the key and value lists can be of any datatype, including dictionaries or tables.

Keys and values

```
q)key d
`a`b`c
q)value d
1 2 3
```

Keywords `key` and `value` return the key and value lists respectively.

Indexing

A dictionary is a mapping from its key items to its value items.

A list is a mapping from its indexes to its items. If the indexes of a list are its keys, it is unsurprising to find a dictionary is indexed by its keys.

```
q)k:`a`b`c`d`e
q)v:10 20 30 40 50
q)show dic:k!v
a| 10
b| 20
c| 30
d| 40
e| 50

q)dic[`d`b]
40 20
q)v[3 1]
40 20
```

Nor that we can omit index brackets the same way.

```
q)dic `d`b
40 20
q)v 3 1
40 20
```

Indexing out of the domain works as for lists, returning a null of the same type as the first value item.

```
q)v 5
0N
q)dic `x
0N
```

But unlike a list, indexed assignment to a dictionary has upsert semantics.

```
q)v[5 1]:42 100
'length
  [0]  v[5 1]:42 100
             ^
q)dic[`x`b]:42 100
q)dic
a| 10
b| 100
c| 30
d| 40
e| 50
x| 42
```

Dictionary indexing uses Find to search the keys.

```
q)d:k!v
q)d[x] ~ v[k?x]
1b
```

`where` and Find

Find and `where` both return indexes from lists. Also from dictionaries.

```
q)d:`a`b`c`d!10 20 30 10

q)where d=10
`a`d

q)d?30
`c
```

Reverse dictionary lookup: use Find for the key of the first matching value, or `where` for all of them.

```
q)dns:`netbox`google`apple!`$("104.130.139.23";"216.58.212.206";"17.172.224.47")

q)dns `apple
```

```
`17.172.224.47

q)dns?`$"17.172.224.47"
`apple

q)where dns=`$"17.172.224.47"
,`apple
```

Order

Dictionaries are ordered.

```
q)first dic
10
q)last dic
42

q)k:`a`b`c
q)v:1 2 3
q)(k!v) ~ reverse[k]!reverse v
0b
```

Taking and dropping from a dictionary

Dictionaries are ordered, so you can take and drop items from either end of them.

```
q)d
a| 10
b| 20
c| 30
d| 10

q)-2#d
c| 30
d| 10

q)-1 _ d
a| 10
b| 20
c| 30
```

You can also take and drop selected items.

```
q)`b`d#d
b| 20
d| 10

q)`b`x _ d
a| 10
```

```
c| 30
d| 10
```

Joining dictionaries

Join on dictionaries has upsert semantics.

```
q)(`a`b`c!10 20 30),`c`d!400 500
a| 10
b| 20
c| 400
d| 500
```

Empty and singleton dictionaries

Just like a list, a dictionary may be empty or have a single item. But its key and value must still be lists.

```
q)()!()                     / general empty dictionary
q)(`symbol$())!`float$()    / typed empty dictionary

q)sd:(enlist `a)!enlist 1   / singleton dictionary
a| 1
q)key sd
,`a
q)value sd
,1
```

Column dictionaries

When a dictionary's value items are all same-length lists, it is a *column dictionary*.

```
q)show bd:`name`dob`sex!(`jack`jill`john;1982.09.15 1984.07.05 1990.11.16;`m`f`m)
name| jack       jill       john
dob | 1982.09.15 1984.07.05 1990.11.16
sex | m          f          m
```

Flip it and we see a table.

```
q)flip bd
name dob        sex
-------------------
jack 1982.09.15 m
jill 1984.07.05 f
john 1990.11.16 m
```

Step dictionaries

Tables

Q for Mortals §5. Dictionaries,

Tables

```
cols      column names              ungroup  normalize
meta      metadata                  xasc     sort ascending
xcol      rename cols               xdesc    sort descending
xcols     re-order cols             xgroup   group by values in selected cols
insert    insert records            xkey     sset cols as primary keys
upsert    add/insert records        xdesc    sort descending
! Enkey, Unkey  add/remove keys

qSQL query templates:   select   exec   update   delete
```

Tables are first-class objects in q.

A table is an ordered list of its rows.

Relations in SQL are sets. There are no duplicate rows, and rows are not ordered. It is possible to define a cursor on a result set and then manipulate the cursor rows in order. (Not in ANSI SQL.)

kdb+ tables are ordered and may contain duplicates. The order allows a class of very useful aggregates that are unavailable to the relational database programmer without the cumbersome and poorly-performing temporal extensions.

Because kdb+ tables are ordered, for a table `trade` in timestamp order, the functions `first` and `last` give open and close prices:

```
q)first trade
stock| `ibm
price| 121.3
amt  | 1000
time | 09:03:06.000

q)last trade
stock| `msft
price| 78.52
amt  | 200
time | 16:59:39.000
```

Construction

Flip a column dictionary

Here is a matrix: a list of four vectors.

```
q)show mat:(`Jack`Jill`Janet;`brown`black`fair;`blue`green`hazel;12 9 14)
Jack  Jill  Janet
brown black fair
blue  green hazel
12    9     14
```

The same information as a column dictionary: each value is a 3-vector.

```
q)`name`hair`eye`age!mat
name| Jack  Jill  Janet
hair| brown black fair
eye | blue  green hazel
age | 12    9     14
```

Flipped, it is a table.

```
q)show t:flip`name`hair`eye`age!mat
name  hair  eye   age
---------------------
Jack  brown blue  12
Jill  black green 9
Janet fair  hazel 14
```

Each row is a dictionary.

```
q)first t
name| `Jack
hair| `brown
eye | `blue
age | 12
```

A table is an ordered list of same-key dictionaries.

Like any list, its length – the number of its records – is given by `count` .

```
q)count t
3
```

Table notation

The same table can be defined with table notation:

```
q)t~([]name:`Jack`Jill`Janet;hair:`brown`black`fair;eye:`blue`green`hazel;age:12 9 14)
1b
```

By default, column names are taken from corresponding variable names.

```
q)stock:`ibm`bac`usb
q)price:121.3 5.76 8.19
q)amount:1000 500 800
q)time:09:03:06.000 09:03:23.000 09:04:01.000

q)show trade:([]stock;price;amt:amount;time)
stock price amt  time
-----------------------------
ibm   121.3 1000 09:03:06.000
bac   5.76  500  09:03:23.000
usb   8.19  800  09:04:01.000
```

Columns get the datatypes of the values assigned them.

```
q)meta trade
c    | t f a
-----| -----
stock| s
price| f
amt  | j
time | t
```

Unlike SQL, there is no need to first declare the types of the columns.

Dict Each Right

If your records are in a row-order matrix, Dict Each Right (`!/:`) will make each row a like dictionary – and a table is a list of like dictionaries.

```
q)mat
`ibm 121.3 1000 09:03:06.000
`bac 5.76  500  09:03:23.000
`usb 8.19  800  09:04:01.000

q)`stock`price`amt`time !/: mat
stock price amt  time
-----------------------------
ibm   121.3 1000 09:03:06.000
bac   5.76  500  09:03:23.000
usb   8.19  800  09:04:01.000
```

Read from disk

Use Load CSV to load a CSV file as a table; more generally, File Text for reading data from text files.

For reading tables persisted to the filesystem, see Database: persisting tables in the filesystem

Compound columns

Any list of the right length can become a column of a table, and its items can be any kdb+ objects – including tables.

Most table columns are vectors: simple lists of uniform type. This is generally most efficient for storage and query execution.

A table column of uniform type but with vector items is called a *compound list*. A book index is a familiar example.

```
q)term:`$("analytical engine";"Babbage, Charles";"difference engine")
q)pages:(5 17 324;1 5 17;17 359)
q)([]term;pages)
term              pages
-------------------------
analytical engine 5 17 324
Babbage, Charles  1 5 17
difference engine 17 359
```

Above, the `term` column is a symbol vector and the `pages` column is a compound list.

In the result of `meta` its compound nature is indicated by its type code in upper case.

```
q)meta([]term;pages)
c    | t f a
-----| -----
term | s
pages| J
```

`meta` does not read the entire table.

The `meta` keyword samples only the top of each column. You cannot rely on it to determine whether a column is simple, compound or mixed.

Table schema

An empty table can be created by initializing each column as the general empty list.

```
q)meta trade:([] stock:(); price:(); amount:(); time:())
c     | t f a
------| -----
stock |
price |
amount|
time  |
```

In this table, the datatype of each column is *mixed*. The first record then inserted into the table sets the datatypes of the columns. Subsequent inserts may only insert values of the same type; otherwise a `type` error is signalled, which can be trapped and handled.

This requires the table to start with the correct column types, so it is often better to initialize a table with empty columns of the correct type.

```
q)meta trade:([] stock:`$(); price:`float$(); `long$amount:(); `time$time:())
c     | t f a
------| -----
stock | s
price | f
amount| j
time  | t
```

Keyed tables

A kdb+ table is a list and it can contain duplicates.

A kdb+ keyed table is a dictionary.

- Its key is a table of the key column/s.
- Its value is a table of the non-key columns.

In table notation, write the key field/s inside the square brackets. For a simple table of financial markets and their addresses:

```
q)market:([name:`symbol$()] address:())
```

When constructing a table key, ensure its items are unique.

To protect performance, kdb+ does not ensure key items are unique.

But there is no use case for duplicate key items, which make operation results unpredictable.

If market names are not unique, the country name could be part of the primary key.

```
q)market:([name:`symbol$(); country:`symbol$()] address:())
```

An alternative to using multiple columns as a primary key is a column of unique integer IDs

`keys`, `xkey` get, set key columns of a table

Foreign keys

Foreign keys in SQL provide referential integrity: an attempt to insert a foreign key value that is not in the primary key will fail. This is also true in q.

Suppose we want to record in our `trades` table the market (NYSE, LSE, etc) where each trade has been executed.

We make the primary key of the `markets` table a foreign key in the `trades` table.

```
q)name:`$("Stock Exchange";"Boersen";"NYSE")
q)country:`$("United Kingdom";"Denmark";"United States")
q)city:`$("London";"Copenhagen";"New York")
q)id:1001 1002 1003
q)show market:([id]name;country;city)
id  | name           country        city
----| ----------------------------------------
1001| Stock Exchange United Kingdom London
1002| Boersen        Denmark        Copenhagen
1003| NYSE           United States  New York

q)trade:([]
    stock:`symbol$();
    market:`market$();
    price:`float$();
    amount:`int$();
    time:`time$() )
```

Only primary keys of keyed tables can be used as a foreign key. But there are other ways to link table columns.

Linking columns
The application of foreign keys and linked columns in kdb+

Indexing a table

Column indexing

If a table is a flipped dictionary, it is unsurprising we can also index the table by column names.

```
q)trade `stock`amt
ibm  bac usb ibm  bac usb
1000 500 800 1000 500 800
```

Row indexing

Selecting the ith row in a table is complex in SQL, but easy in q.

The pseudo column `i` represents the row index. It can be used in queries.

The `select` expression returns a table with a single row.

```
q)select from trade where i=5
stock price amt time
----------------------------
usb   8.19  800 09:04:01.000
```

Indexing a table with an atom returns a dictionary.

```
q)trade[5]
stock| `usb
price| 8.19
amt  | 800
time | 09:04:01.000
```

Row indexing cannot be used on a keyed table, which is a dictionary.

It is also easy to access, say, the second-to-last row.

```
q)trade[(count trade) - 2]
stock| `bac
price| 5.76
amt  | 500
time | 09:03:23.000
```

We see at last the dual nature of a table. It is *both*

- a list of named same-length columns
- a list of like (same-key) dictionaries

And we can index it either way – or both, which is indexing at depth.

```
q)trade 1                    / index by row
stock| `bac
price| 5.76
amt  | 500
time | 09:03:23.000

q)trade `price               / index by column
121.3 5.76 8.19

q)trade[1 0;`stock`amt]      / index at depth
`bac 500
`ibm 1000
```

Its items are dictionaries and, as a table is a list of like dictionaries, any sublist of the table is – also a table.

```
q)trade 1 0
stock price amt  time
-----------------------------
bac   5.76  500  09:03:23.000
ibm   121.3 1000 09:03:06.000
```

Indexing at depth

Indexing at depth can be used to read a column within a specific row.

```
trade[5;`stock]
`usb
```

Also useful for updates.

```
trade[5;`amt]:15
```

Apply/Index at depth

Index out of range

If we use `select`, the result is a table with no rows.

```
q)select from trade where i = 300000
stock price amount time
---------------------------------------
```

If we use indexing, the result is a dictionary containing null values.

```
q)trade 30000
stock| `
price| 0n
amt  | 0N
time | 0Nt
```

Indexing a keyed table

There are two ways to index a keyed table.

First, with a single row from its key, returning a dictionary.

```
q)flip{x cols x}key kt
dick
jane
jack
jill
john
q)kt `john
dob| 1990.11.16
sex| `m

q)flip{x cols x}key ku
Tom NYC
Jo  LA
Tom Lagos
q)ku `Tom`Lagos
eye| brown
sex| m
```

Second, with a sublist from its key, returning a list of dictionaries, which is a table.

```
q)ku ([]city:`LA`Lagos; name:`Jo`Tom)
eye   sex
---------
blue  f
brown m
```

qSQL and Functional SQL

The foregoing describes dictionaries and tables in terms of lists and indexes. Functional SQL extends the concepts through the query operators `?` and `!`.

If you are familiar with SQL, you will find qSQL queries more readable.

Functional SQL, qSQL
Q for Mortals §8. Tables

List operations

A table is an ordered list. Many list operators work.

Take and Drop

For the head and tail of a table use the Take operator `#`.

```
q)2#trade                         / take first two records
stock price amt  time
-----------------------------
ibm   121.3 1000 09:03:06.000
bac   5.76  500  09:03:23.000

q)-2#trade                        / take last two records
stock price amt time
----------------------------
bac   5.76  500 09:03:23.000
usb   8.19  15  09:04:01.000

q)-2 _ trade                      / drop last two records
stock price amt  time
-----------------------------
ibm   121.3 1000 09:03:06.000
bac   5.76  500  09:03:23.000
usb   8.19  800  09:04:01.000
ibm   121.3 1000 09:03:06.000
```

Note that Take treats a list as circular if the number of items to take is longer than the list.

```
q)7#2 3 5
2 3 5 2 3 5 2
```

An alternative is to use `sublist`, which takes only as many rows as are available.

```
q)count trade
6
q)count 3 sublist trade
3
q)count 30 sublist trade
6
```

Limit expressions in `select`

You can also take selected columns.

```
q)`price`amt#trade
price amt
----------
121.3 1000
5.76  500
8.19  800
121.3 1000
5.76  500
8.19  15
```

Join and Join Each

The Join operator `,` catenates two lists – and tables are lists. The table columns need not be in the same order.

```
q)trade,([]time:10:32:17.000 10:35:45.000;stock:`msft`aapl;amt:1500 750;price:17.5 103.2)
stock price amt  time
-----------------------------
ibm   121.3 1000 09:03:06.000
bac   5.76  500  09:03:23.000
usb   8.19  800  09:04:01.000
ibm   121.3 1000 09:03:06.000
bac   5.76  500  09:03:23.000
usb   8.19  15   09:04:01.000
msft  17.5  1500 10:32:17.000
aapl  103.2 750  10:35:45.000
```

Join Each joins pairs of dictionaries and so has upsert semantics.

```
q)trade,'([]year:2019+til 6;exch:6?`NYSE`LSE;amt:999)
stock price amt time         year exch
--------------------------------------
ibm   121.3 999 09:03:06.000 2019 NYSE
bac   5.76  999 09:03:23.000 2020 LSE
usb   8.19  999 09:04:01.000 2021 NYSE
ibm   121.3 999 09:03:06.000 2022 LSE
```

```
bac   5.76  999 09:03:23.000 2023 NYSE
usb   8.19  999 09:04:01.000 2024 LSE
```

There are many other join keywords.

List alternatives to queries

Many qSQL queries are equivalent to simple list operations.

```
select from trade where i=5           / trade[5]
select stock,amt from trade           / `stock`amt#trade
select from trade where stock=`ibm    / trade where `ibm=trade`stock
```

Amending a table

Inserting records

```
`trade insert (`ibm; 1001; 122.5; 500; 09:04:59:000)
```

Alternative syntax:

```
insert [`trade] (`ibm; 1001; 122.5; 500; 09:04:59:000)
insert [`trade; (`ibm; 1001; 122.5; 500; 09:04:59:000)]

q)table:([stock:()] price:())
q)insert[`table; (`intel; enlist (123.2; 120.4; 131.0))]
q)table
stock| price
-----| ---------------
intel| 123.2 120.4 131
```

Bulk insert

The right argument to `insert` above is a list. It can also be a table having the same column names as the first argument.

```
q)`trade insert trade
```

Upsert

```
q)`trade upsert (`ibm; 122.5; 50; 09:04:59:000)
```

For a simple table (not keyed) the above is equivalent to an `insert`. For a keyed table, it is an `update` if the key exists in the table and an `insert` otherwise.

An alternative syntax for `upsert` is to use the operator `,:`

```
q)trade ,: (`ibm; 122.5; 50; 09:04:59:000)
```

`upsert` can also take a table as an argument.

```
trade ,: trade
```

Delete rows

```
trade: delete from trade where stock=`ibm
```

`delete` returns a table, but does not modify the `trade` table in place. The assignment accomplishes that.

An alternative that updates the table in place:

```
delete from `trade where stock=`ibm
```

Update values

In SQL:

```
UPDATE trade SET amount=42+amount WHERE stock='ibm'
```

In q:

```
trade: update amount:42+amount from trade where stock=`ibm
```

`update` returns a table, but does not modify the underlying table. The assignment accomplishes that. Alternatively:

```
update amount+42 from `trade where stock=`ibm
```

`update` modifies the table in place much like `delete` deletes in place if a symbol is given as the table name. Also note the default column names.

Replace null values

Use the Fill operator `^`. For example, the following replaces all nulls in column `amount` by zeroes.

```
trade.amount: 0^trade.amount
```

Table to set

Use `distinct` to remove duplicate records, and `count` to count them.

```
distinct trade
count distinct trade
```

---

qSQL query templates

Functional qSQL

Database: persisting tables to the filesystem

Q for Mortals §8. Tables

QSQL query templates

```
delete  delete rows or columns from a table
exec    return columns from a table, possibly with new columns
select  return part of a table, possibly with new columns
update  add rows or columns to a table
```

The query templates of qSQL share a query syntax that varies from the syntax of q and closely resembles conventional SQL. For many use cases involving ordered data it is significantly more expressive.

Template syntax

Below, square brackets mark optional elements; a slash begins a trailing comment.

```
select [$L_{exp}$]     [$p_s$] [by $p_b$] from $t_{exp}$ [where $p_w$]
exec   [distinct] [$p_s$] [by $p_b$] from $t_{exp}$ [where $p_w$]
update             $p_s$  [by $p_b$] from $t_{exp}$ [where $p_w$]
delete                          from $t_{exp}$ [where $p_w$]        / rows
delete             $p_s$           from $t_{exp}$                    / columns
```

A template is evaluated in the following order.

From phrase	t_{exp}
Where phrase	p_w
By phrase	p_b
Select phrase	p_s
Limit expression	L_{exp}

From phrase

The From phrase `from` t_{exp} is required in all query templates.

The table expression t_{exp} is

- a table or dictionary (call-by-value)
- the name of a table or dictionary, in memory or on disk, as a symbol atom (call-by-name)

Examples:

```
update c:b*2 from ([]a:1 2;b:3 4)    / call by value
select a,b from t                    / call by value
select a,b from `t                   / call by name
update c:b*2 from `:path/to/db       / call by name
```

Limit expressions

Limit expressions restrict the results returned by `select` or `exec`. (For `exec` there is only one: `distinct`). They are described in the articles for `select` and `exec`.

Result and side effects

In a `select` query, the result is a table or dictionary.

In an `exec` query the result is a list of column values, or dictionary.

In an `update` or `delete` query, where the table expression is a call

- by value, the query returns the modified table or a dictionary
- by name, the table or dictionary is amended in place (in memory or on disk) as a side effect, and its name returned as the result

```
q)t1:t2:([]a:1 2;b:3 4)

q)update a:neg a from t1
a  b
----
-1 3
-2 4
q)t1~t2   / t1 unchanged
1b

q)update a:neg a from `t1
`t1
q)t1~t2   / t1 changed
0b
```

Phrases and subphrases

p_s, p_b, and p_w are respectively the Select, By, and Where *phrases*. Each phrase is a comma-separated list of subphrases.

A *subphrase* is a q expression in which names are resolved with respect to t_{exp} and any table/s linked by foreign keys. Subphrases are evaluated in order from the left, but each subphrase expression is evaluated right-to-left in normal q syntax.

To use the Join operator within a subphrase, parenthesize the subphrase.

```
q)select (id,'4),val from tbl
x   val
-------
1 4 100
1 4 200
2 4 300
2 4 400
2 4 500
```

Names in subphrases

A name in a subphrase is resolved (in order) as the name of

1. column or key name
2. local name in (or argument of) the encapsulating function
3. global name in the current working namespace – not necessarily the space in which the function was defined

Dot notation allows you to refer to foreign keys.

Suppliers and parts database `sp.q`

```
q)\l sp.q
+`p`city!(`p$`p1`p2`p3`p4`p5`p6`p1`p2;`london`london`london`london`london`lon..
(`s#+(,`color)!,`s#`blue`green`red)!+(,`qty)!,900 1000 1200
+`s`p`qty!(`s$`s1`s1`s1`s2`s3`s4;`p$`p1`p4`p6`p2`p2`p4;300 200 100 400 200 300)

q)select sname:s.name, qty from sp
sname qty
---------
smith 300
smith 200
smith 400
smith 200
clark 100
smith 100
jones 300
jones 400
blake 200
clark 200
clark 300
smith 400
```

Implicit joins

You can refer explicitly to namespaces.

```
select (\`. \`toplevel) x from t
```

Duplicate names for columns or groups

`select` auto-aliases colliding duplicate column names for either `select az,a from t`, or `select a by c,c from t`, but not for `select a,a by a from t`.

Such a collision throws a `'dup names for cols/groups a` error during parse, indicating the first column name which collides. (Since V4.0 2020.03.17.)

```
q)parse"select b by b from t"
'dup names for cols/groups b
  [2]  select b by b from t
       ^
```

The easiest way to resolve this conflict is to explicitly rename columns. e.g. `select a,b by c:a from t`.

When compiling functions, the implicit args `x`, `y`, `z` are visible to the compiler only when they are not inside the Select, By, and Where phrases. The table expression is not masked. This can be observed by taking the `value` of the function and observing the second item: the args.

```
q)args:{(value x)1}
q)args{} / no explicit args, so x is a default implicit arg of identity (::)
,`x

q)/from phrase is not masked, y is detected as an implicit arg here
q)args{select from y where a=x,b=z}
`x`y
q)args{[x;y;z]select from y where a=x,b=z} / x,y,z are now explicit args
`x`y`z

q)/call with wrong number of args results in rank error
q){select from ([]a:0 1;b:2 3) where a=x,b=y}[0;2]
'rank
  [0]  {select from ([]a:0 1;b:2 3) where a=x,b=y}[0;2]
       ^

q)/works with explicit args
q){[x;y]select from ([]a:0 1;b:2 3) where a=x,b=y}[0;2]
a b
---
0 2
```

Computed columns

In a subphrase, a q expression computes a new column or key, and a colon names it.

```
q)t:([] c1:`a`b`c; c2:10 20 30; c3:1.1 2.2 3.3)

q)select c1, c3*2 from t
c1 c3
------
a  2.2
b  4.4
```

```
c  6.6

q)select c1, dbl:c3*2 from t
c1 dbl
------
a  2.2
b  4.4
c  6.6
```

In the context of a query, the colon names a result column or key. It does not assign a variable in the workspace.

If a computed column or key is not named, q names it if possible as the leftmost term in the column expression, else as `x`. If a computed name is already in use, q suffixes it with `1`, `2`, and so on as needed to make it unique.

```
q)select c1, c1, 2*c2, c2+c3, string c3 from t
c1 c11 x  c2   c3
--------------------
a  a   20 11.1 "1.1"
b  b   40 22.2 "2.2"
c  c   60 33.3 "3.3"
```

Virtual column `i`

A virtual column `i` represents the index of each record, i.e., the row number.

Partitioned tables

In a partitioned table `i` is the index (row number) relative to the partition, not the whole table.

Because it is implicit in every table, it never appears as a column or key name in the result.

```
q)select i, c1 from t
x c1
----
0 a
1 b
2 c

q)select from t where i in 0 2
c1 c2 c3
---------
a  10 1.1
c  30 3.3
```

Where phrase

The Where phrase with a boolean list selects records.

```
q)select from t where 101b
c1 c2 c3
---------
a  10 1.1
c  30 3.3
```

Subphrases specify *successive* filters.

```
q)select from t where c2>15,c3<3.0
c1 c2 c3
---------
b  20 2.2

q)select from t where (c2>15) and c3<3.0
c1 c2 c3
---------
b  20 2.2
```

The examples above return the same result but have different performance characteristics.

In the second example, all `c2` values are compared to 15, and all `c3` values are compared to 3.0. The two result vectors are ANDed together.

In the first example, only `c3` values corresponding to `c2` values greater than 15 are tested.

Efficient Where phrases start with their most stringent tests.

Querying a partitioned table

When querying a partitioned table, the first Where subphrase should select from the value/s used to partition the table.

Otherwise, kdb+ will (attempt to) load into memory all partitions for the column/s in the first subphrase.

Use `fby` to filter on groups.

Aggregates

In SQL:

```
SELECT stock, SUM(amount) AS total FROM trade GROUP BY stock
```

In q:

```
q)select total:sum amt by stock from trade
stock| total
-----| -----
bac  | 1000
```

```
ibm  | 2000
usb  | 815
```

The column `stock` is a key in the result table.

Mathematics for more aggregate functions

Sorting

Unlike SQL, the query templates make no provision for sorting. Instead use `xasc` and `xdesc` to sort the query results.

As the sorts are stable, they can be combined for mixed sorts.

```
q)sp
s  p  qty
---------
s1 p1 300
s1 p2 200
s1 p3 400
s1 p4 200
s4 p5 100
s1 p6 100
s2 p1 300
s2 p2 400
s3 p2 200
s4 p2 200
s4 p4 300
s1 p5 400

q)`p xasc `qty xdesc select from sp where p in `p2`p4`p5
s  p  qty
---------
s2 p2 400
s1 p2 200
s3 p2 200
s4 p2 200
s4 p4 300
s1 p4 200
s1 p5 400
s4 p5 100
```

Performance

- Select only the columns you will use.
- Use the most restrictive constraint first.
- Ensure you have a suitable attribute on the first non-virtual constraint (e.g. `` `p `` or `` `g `` on sym).
- Constraints should have the unmodified column name on the left of the constraint operator (e.g. where sym in syms,...)

- When aggregating, use the virtual field first in the By phrase. (E.g. `select .. by date,sym from …`)

Tip

`…where `g=,`s  within …`
Maybe rare to get much speedup, but if the `` `g `` goes to 100,000 and then `` `s `` is 1 hour of 24 you might see some overall improvement (with overall table of 30 million).

Q for Mortals §14.3.6 Query Execution on Partitioned Tables

Multithreading

The following pattern will make use of secondary threads via `peach`

```
select … by sym, … from t where sym in …, …
```

when `sym` has a `` `g `` or `` `p `` attribute. (Since V3.2 2014.05.02)

It uses `peach` for both in-memory and on-disk tables. For single-threaded, this is approx 6× faster in memory, 2× faster on disk, and uses less memory than previous releases – but mileage will vary. This is also applicable for partitioned DBs as

```
select … by sym, … from t where date …, sym in …, …
```

Table counts in a partitioned database

Special functions

The following functions (essentially `.Q.a0` in `q.k`) receive special treatment within `select`:

```
avg     first   prd
cor     last    sum
count   max     var
cov     med     wavg
dev     min     wsum
```

When used explicitly, such that it can recognize the usage, q will perform additional steps, such as enlisting results or aggregating across partitions. However, when wrapped inside another function, q does not know that it needs to perform these additional steps, and it is then left to the programmer to insert them.

```
q)select sum a from ([]a:1 2 3)
a
-
6
q)select {(),sum x}a from ([]a:1 2 3)
a
```

```
-
6
```

Cond

Cond is not supported inside qSQL expressions.

```
q)u:([]a:raze ("ref/";"kb/"),\:/:"abc"; b:til 6)
q)select from u where a like $[1b;"ref/*";"kb/*"]
'rank
  [0]  select from u where a like $[1b;"ref/*";"kb/*"]
                                  ^
```

Enclose in a lambda

```
q)select from u where a like {$[x;"ref/*";"kb/*"]}1b
a       b
---------
"ref/a" 0
"ref/b" 2
"ref/c" 4
```

or use the Vector Conditional instead.

Functional SQL

The interpreter translates the query templates into functional SQL for evaluation. The functional forms are more general, and some complex queries require their use. But the query templates are powerful, readable, and there is no performance penalty for using them.

Wherever possible, prefer the query templates to functional forms.

Stored procedures

Any suitable lambda can be used in a query.

```
q)f:{[x] x+42}
q)select stock, f amount from trade
stock amount
------------
ibm   542
...
```

Parameterized queries

Query template expressions can be evaluated in lambdas.

```
q)myquery:{[tbl; amt] select stock, time from tbl where amount > amt}
q)myquery[trade; 100]
stock time
------------------
ibm   09:04:59.000
...
```

Column names cannot be parameters of a qSQL query. Use functional qSQL in such cases.

Queries using SQL syntax

Q implements a translation layer from SQL. The syntax is to prepend `s)` to the SQL query.

```
q)s)select * from trade
```

⚠ **Only a subset of SQL is supported.**

`fby`, `insert`, `upsert`,
Functional SQL
Views
Q for Mortals §9.0 Queries: q-sql
Q for Mortals §9.9.10 Parameterized Queries

Functional qSQL

The functional forms of `delete`, `exec`, `select` and `update` are particularly useful for programmatically-generated queries, such as when column names are dynamically produced.

Functional form is an alternative to using a qSQL template to construct a query. For example, the following are equivalent:

```
q)select n from t
q)?[t;();0b;(enlist `n)!enlist `n]
```

> **Performance**
>
> The q interpreter parses `delete`, `exec`, `select`, and `update` into their equivalent functional forms, so there is no performance difference.

The functional forms are

```
![t;c;b;a]              /update and delete

?[t;i;p]                /simple exec

?[t;c;b;a]              /select or exec
?[t;c;b;a;n]            /select up to n records
?[t;c;b;a;n;(g;cn)]     /select up to n records sorted by g on cn
```

where:

- `t` is a table, or the name of a table as a symbol atom.
- `c` is the Where phrase, a list of constraints.
  Every constraint in `c` is a parse tree representing an expression to be evaluated; the result of each being a boolean vector. The parse tree consists of a function followed by a list of its arguments, each an expression containing column names and other variables. Represented by symbols, it distinguishes actual symbol constants by enlisting them. The function is applied to the arguments, producing a boolean vector that selects the rows. The selection is performed in the order of the items in `c`, from left to right: only rows selected by one constraint are evaluated by the next.
- `b` is the By phrase.
  The domain of dictionary `b` is a list of symbols that are the key names for the grouping. Its range is a list of column expressions (parse trees) whose results are used to construct the groups. The grouping is ordered by the domain items, from major to minor. `b` is one of:
  - the general empty list `()`

    - boolean atom: `0b` for no grouping; `1b` for distinct
    - a symbol atom or list naming table column/s
    - a dictionary of group-by specifications

- `a` is the Select phrase. The domain of dictionary `a` is a list of symbols containing the names of the produced columns. QSQL query templates assign default column names in the result, but here each result column must be named explicitly.
  Each item of its range is an evaluation list consisting of a function and its argument(s), each of which is a column name or another such result list. For each evaluation list, the function is applied to the specified value(s) for each row and the result is returned. The evaluation lists are resolved recursively when operations are nested.
  `a` is one of
    - the general empty list `()`
    - a symbol atom: the name of a table column
    - a parse tree
    - a dictionary of select specifications (aggregations)
- `i` is a list of indexes
- `p` is a parse tree
- `n` is a non-negative integer or infinity, indicating the maximum number of records to be returned
- `g` is a unary grade function

Call by name

Columns in `a`, `b` and `c` appear as symbols.

To distinguish symbol atoms and vectors from columns, enlist them.

```
q)t:([] c1:`a`b`a`c`a`b`c; c2:10*1+til 7; c3:1.1*1+til 7)

q)select from t where c2>35,c1 in `b`c
c1 c2 c3
---------
c  40 4.4
b  60 6.6
c  70 7.7

q)?[t; ((>;`c2;35);(in;`c1;enlist[`b`c])); 0b; ()]
c1 c2 c3
---------
c  40 4.4
b  60 6.6
c  70 7.7
```

Note above that

- the columns `c1` and `c2` appear as symbol atoms

- the symbol vector `` `b`c `` appears as ``enlist[`b`c]``

Use `enlist` to create singletons to ensure appropriate entities are lists.

Different types of `a` and `b` return different types of result for Select and Exec.

```
           | b
a          | bool    ()         sym/s   dict
-----------|----------------------------------------
()         | table    dict       -       keyed table
sym        | -        vector     dict    dict
parse tree | -        vector     dict    dict
dict       | table    vector/s   table   table
```

? Select

```
?[t;c;b;a]
```

Where `t`, `c`, `b`, and `a` are as above, returns a table.

```
q)show t:([]n:`x`y`x`z`z`y;p:0 15 12 20 25 14)
n p
----
x 0
y 15
x 12
z 20
z 25
y 14

q)select m:max p,s:sum p by name:n from t where p>0,n in `x`y
name| m  s
----| -----
x   | 12 12
y   | 15 29
```

select

Following is the equivalent functional form. Note the use of `enlist` to create singletons, ensuring that appropriate entities are lists.

```
q)c: ((>;`p;0);(in;`n;enlist `x`y))
q)b: (enlist `name)!enlist `n
q)a: `m`s!((max;`p);(sum;`p))
q)?[t;c;b;a]
name| m  s
----| -----
x   | 12 12
y   | 15 29
```

Degenerate cases

- For no constraints, make `c` the empty list
- For no grouping make `b` a boolean `0b`
- For distinct rows make `b` a boolean `1b`
- To produce all columns of `t` in the result, make `a` the empty list `()`

`select from t` is equivalent to functional form `?[t;();0b;()]`.

Select distinct

For special case `select distinct` specify `b` as `1b`.

```
q)t:([] c1:`a`b`a`c`b`c; c2:1 1 1 2 2 2; c3:10 20 30 40 50 60)

q)?[t;(); 1b; `c1`c2!`c1`c2]        / select distinct c1,c2 from t
c1 c2
-----
a  1
b  1
c  2
b  2
```

Rank 5

Limit result rows

```
?[t;c;b;a;n]
```

Returns as for rank 4, but where `n` is

- an integer or infinity, only the first `n` rows, or the last if `n` is negative
- a pair of non-negative integers, up to `n[1]` rows starting with row `n[0]`

```
q)show t:([] c1:`a`b`c`a; c2:10 20 30 40)
c1 c2
-----
a  10
b  20
c  30
a  40

q)?[t;();0b;();-2]                   / select[-2] from t
c1 c2
-----
c  30
a  40
```

```
q)?[t;();0b;();1 2]                 / select[1 2] from t
c1 c2
-----
b  20
c  30
```

Rank 6

Limit result rows and sort by a column

```
?[t;c;b;a;n;(g;cn)]
```

Returns as for rank 5, but where

- `g` is a unary grading function
- `cn` is a column name as a symbol atom

sorted by `g` on column `cn`.

```
q)?[t; (); 0b; `c1`c2!`c1`c2; 0W; (idesc;`c1)]
c1 c2
-----
c  30
b  20
a  10
a  40
```

Q for Mortals §9.12.1 Functional select

? Exec

A simplified form of Select that returns a list or dictionary rather than a table.

```
?[t;c;b;a]
```

The constraint specification `c` (Where) is as for Select.

```
q)show t:([] c1:`a`b`c`c`a`a; c2:10 20 30 30 40 40;
    c3: 1.1 2.2 3.3 3.3 4.4 3.14159; c4:`cow`sheep`cat`dog`cow`dog)
c1 c2 c3      c4
-------------------
a  10 1.1     cow
b  20 2.2     sheep
c  30 3.3     cat
c  30 3.3     dog
a  40 4.4     cow
a  40 3.14159 dog
```

exec

No grouping

`b` is the general empty list.

```
b   a      result
--------------------------------------------------------------
()  ()     the last row of t as a dictionary
()  sym    the value of that column
()  dict   a dictionary with keys and values as specified by a
```

```
q)?[t; (); (); ()]                          / exec last c1,last c2,last c3 from t
c1| `a
c2| 40
c3| 3.14159
c4| `dog

q)?[t; (); (); `c1]                         / exec c1 from t
`a`b`c`c`a`a

q)?[t; (); (); `one`two!`c1`c2]             / exec one:c1,two:c2 from t
one| a  b  c  c  a  a
two| 10 20 30 30 40 40

q)?[t; (); (); `one`two!(`c1;(sum;`c2))]    / exec one:c1,two:sum c2 from t
one| `a`b`c`c`a`a
two| 170
```

Group by column

`b` is a column name. The result is a dictionary.

Where `a` is a **column name**, in the result

- the keys are distinct values of the column named in `b`
- the values are lists of corresponding values from the column named in `a`

```
q)?[t; (); `c1; `c2]      / exec c2 by c1 from t
a| 10 40 40
b| ,20
c| 30 30
```

Where `a` is a **dictionary**, in the result

- the key is a table with a single anonymous column containing distinct values of the column named in `b`
- the value is a table with columns as defined in `a`

```
q)?[t; (); `c1; enlist[`c2]!enlist`c2]      / exec c2:c2 by c1 from t
 | c2
-| --------
a| 10 40 40
b| ,20
```

```
c| 30 30

q)?[t; (); `c1; `two`three!`c2`c3]          / exec two:c2,three:c3 by c1 from t
 | two      three
-| ------------------------
a| 10 40 40 1.1 4.4 3.14159
b| ,20      ,2.2
c| 30 30    3.3 3.3

q)?[t;();`c1;`m2`s3!((max;`c2);(sum;`c3))]  / exec m2:max c2,s3:sum c3 by c1 from t
 | m2  s3
-| -----------
a| 40  8.64159
b| 20  2.2
c| 30  6.6
```

Group by columns

b is a list of column names.

Where a is a **column name**, returns a dictionary in which

- the key is the empty symbol
- the value is the value of the column/s specified in a

```
q)?[t; (); `c1`c2; `c3]
| 1.1 2.2 3.3 3.3 4.4 3.14159

q)?[t; (); `c1`c2; `c3`c4!((max;`c3);(last;`c4))]
| c3  c4
| -------
| 4.4 dog
```

Group by a dictionary

b is a dictionary. Result is a dictionary in which the key is a table with columns as specified by b and

```
b     a     result value
-----------------------------------------------------
dict  ()    last records of table that match each key
dict  sym   corresponding values from the column in a
dict  dict  values as defined in a
```

```
q)?[t; (); `one`two!`c1`c2; ()]
one two| c1 c2 c3      c4
-------| -------------------
a   10 | a  10 1.1     cow
a   40 | a  40 3.14159 dog
b   20 | b  20 2.2     sheep
c   30 | c  30 3.3     dog
q)/ exec last c1,last c2,last c3,last c4 by one:c1,two:c2 from t
```

```
q)?[t; (); enlist[`one]!enlist(string;`c1); ()]
one | c1 c2 c3      c4
----| -------------------
,"a"| a  40 3.14159 dog
,"b"| b  20 2.2     sheep
,"c"| c  30 3.3     dog
q)/ exec last c1,last c2,last c3,last c4 by one:string c1 from t

q)?[t; (); enlist[`one]!enlist `c1; `c2]     / exec c2 by one:c1 from t
one|
---| --------
a  | 10 40 40
b  | ,20
c  | 30 30

q)?[t; (); `one`four!`c1`c4; `m2`s3!((max;`c2);(sum;`c3))]
one four | m2 s3
---------| ----------
a   cow  | 40 5.5
a   dog  | 40 3.14159
b   sheep| 20 2.2
c   cat  | 30 3.3
c   dog  | 30 3.3
```

Q for Mortals §9.12.2 Functional exec

? Simple Exec

```
?[t;i;p]
```

Where `t` is not partitioned, another form of Exec.

```
q)show t:([]a:1 2 3;b:4 5 6;c:7 9 0)
a b c
-----
1 4 7
2 5 9
3 6 0

q)?[t;0 1 2;`a]
1 2 3
q)?[t;0 1 2;`b]
4 5 6
q)?[t;0 1 2;(last;`a)]
3
q)?[t;0 1;(last;`a)]
2
q)?[t;0 1 2;(*;(min;`a);(avg;`c))]
5.333333
```

! Update

```
![t;c;b;a]
```

update

Arguments `t`, `c`, `b`, and `a` are as for Select.

```
q)show t:([]n:`x`y`x`z`z`y;p:0 15 12 20 25 14)
n p
----
x 0
y 15
x 12
z 20
z 25
y 14

q)select m:max p,s:sum p by name:n from t where p>0,n in `x`y
name| m  s
----| -----
x   | 12 12
y   | 15 29

q)update p:max p by n from t where p>0
n p
----
x 0
y 15
x 12
z 25
z 25
y 15

q)c: enlist (>;`p;0)
q)b: (enlist `n)!enlist `n
q)a: (enlist `p)!enlist (max;`p)

q)![t;c;b;a]
n p
----
x 0
y 15
x 12
z 25
z 25
y 15
```

The degenerate cases are the same as in Select.

Q for Mortals §9.12.3 Functional update

! Delete

A simplified form of Update

```
![t;c;0b;a]
```

`delete`

One of `c` or `a` must be empty, the other not. `c` selects which rows will be removed. `a` is a symbol vector with the names of columns to be removed.

```
q)t:([]c1:`a`b`c;c2:`x`y`z)

q)/following is: delete c2 from t
q)![t;();0b;enlist `c2]
c1
--
a
b
c

q)/following is: delete from t where c2 = `y
q)![t;enlist (=;`c2; enlist `y);0b;`symbol$()]
c1 c2
-----
a  x
c  z
```

Q for Mortals §9.12.4 Functional delete

Conversion using parse

Applying parse to a qSQL statement written as a string will return the internal representation of the functional form. With some manipulation this can then be used to piece together the functional form in q. This generally becomes more difficult as the query becomes more complex and requires a deep understanding of what kdb+ is doing when it parses qSQL form.

An example of using parse to convert qSQL to its corresponding functional form is as follows:

```
q)t:([]c1:`a`b`c; c2:10 20 30)
q)parse "select c2:2*c2 from t where c1=`c"
?
`t
,,(=;`c1;,`c)
0b
(,`c2)!,(*;2;`c2)

q)?[`t; enlist (=;`c1;enlist `c); 0b; (enlist `c2)!enlist (*;2;`c2)]
c2
--
60
```

Issues converting to functional form

To convert a `select` query to a functional form one may attempt to apply the `parse` function to the query string:

```
q)parse "select sym,price,size from trade where price>50"
?
`trade
,,(>;`price;50)
0b
`sym`price`size!`sym`price`size
```

As we know, `parse` produces a parse tree and since some of the elements may themselves be parse trees we can't immediately take the output of parse and plug it into the form `?[t;c;b;a]`. After a little playing around with the result of `parse` you might eventually figure out that the correct functional form is as follows.

```
q)funcQry:?[`trade;enlist(>;`price;50);0b;`sym`price`size! `sym`price`size]

q)strQry:select sym,price,size from trade where price>50 q)
q)funcQry~strQry
1b
```

This, however, becomes more difficult as the query statements become more complex:

```
q)parse "select count i from trade where 140>(count;i) fby sym"
?
`trade
,,(>;140;(k){@[(#y)#x[0]0#x
1;g;:;x[0]'x[1]g:.=y]};(enlist;#:;`i);`sym))
0b
(,`x)!,(#:;`i)
```

In this case, it is not obvious what the functional form of the above query should be, even after applying `parse`.

There are three issues with this parse-and-"by eye" method to convert to the equivalent functional form. We will cover these in the next three subsections.

Parse trees and eval

The first issue with passing a `select` query to `parse` is that each returned item is in unevaluated form. As discussed here, simply applying `value` to a parse tree does not work. However, if we evaluate each one of the arguments fully, then there would be no nested parse trees. We could then apply `value` to the result:

```
q)eval each parse "select count i from trade where 140>(count;i) fby sym"
?
+`sym`time`price`size!(`VOD`IBM`BP`VOD`IBM`IBM`HSBC`VOD`MS..
,(>;140;(k){@[(#y)#x[0]0#x
1;g;:;x[0]'x[1]g:.=y]};(enlist;#:;`i);`sym))
0b
(,`x)!,(#:;`i)
```

The equivalence below holds for a general qSQL query provided as a string:

```
q)value[str]~value eval each parse str
1b
```

In particular:

```
q)str:"select count i from trade where 140>(count;i) fby sym"

q)value[str]~value eval each parse str
1b
```

In fact, since within the functional form we can refer to the table by name we can make this even clearer. Also, the first item in the result of `parse` applied to a `select` query will always be `?` (or `!` for a `delete` or `update` query) which cannot be evaluated any further. So we don't need to apply `eval` to it.

```
q)pTree:parse str:"select count i from trade where 140>(count;i) fby sym"
q)@[pTree;2 3 4;eval]
?
`trade
,(>;140;(k){@[(#y)#x[0]0#x
1;g;::;x[0]'x[1]g:.=y]};(enlist;#:;`i);`sym))
0b
(,`x)!,(#:;`i)
q)value[str] ~ value @[pTree;2 3 4;eval]
1b
```

Variable representation in parse trees

Recall that in a parse tree a variable is represented by a symbol containing its name. So to represent a symbol or a list of symbols, you must use `enlist` on that expression. In k, `enlist` is the unary form of the comma operator in k:

```
q)parse"3#`a`b`c`d`e`f"
#
3
,`a`b`c`d`e`f
q)(#;3;enlist `a`b`c`d`e`f)~parse"3#`a`b`c`d`e`f"
1b
```

This causes a difficulty. As discussed above, q has no unary syntax for operators.

Which means the following isn't a valid q expression and so returns an error.

```
q)(#;3;,`a`b`c`d`e`f)
',
```

In the parse tree we receive we need to somehow distinguish between k's unary `,` (which we want to replace with `enlist`) and the binary Join operator, which we want to leave as it is.

Explicit definitions in `.q` are shown in full

The `fby` in the `select` query above is represented by its full k definition.

```
q)parse "fby"
k){@[(#y)#x[0]0#x 1;g;::;x[0]'x[1]g:.=y]}
```

While using the k form isn't generally a problem from a functionality perspective, it does however make the resulting functional statement difficult to read.

The solution

We will write a function to automate the process of converting a `select` query into its equivalent functional form.

This function, `buildQuery`, will return the functional form as a string.

```
q)buildQuery "select count i from trade where 140>(count;i) fby sym"
"?[trade;enlist(>;140;(fby;(enlist;count;`i);`sym));0b;
  (enlist`x)! enlist (count;`i)]"
```

When executed it will always return the same result as the `select` query from which it is derived:

```
q)str:"select count i from trade where 140>(count;i) fby sym"
q)value[str]~value buildQuery str
1b
```

And since the same logic applies to `exec`, `update` and `delete` it will be able to convert to their corresponding functional forms also.

To write this function we will solve the three issues outlined above:

1. parse-tree items may be parse trees
2. parse trees use k's unary syntax for operators
3. q keywords from `.q.` are replaced by their k definitions

The first issue, where some items returned by `parse` may themselves be parse trees is easily resolved by applying `eval` to the individual items.

The second issue is with k's unary syntax for `,` . We want to replace it with the q keyword `enlist`. To do this we define a function that traverses the parse tree and detects if any element is an enlisted list of symbols or an enlisted single symbol. If it finds one we replace it with a string representation of `enlist` instead of `,` .

```
ereptest:{ //returns a boolean
  (1=count x) and ((0=type x) and 11=type first x) or 11=type x}
ereplace:{"enlist",.Q.s1 first x}
funcEn:{$[ereptest x;ereplace x;0=type x;.z.s each x;x]}
```

Before we replace the item we first need to check it has the correct form. We need to test if it is one of:

- An enlisted list of syms. It will have type `0h`, count 1 and the type of its first item will be `11h` if and only if it is an enlisted list of syms.

- An enlisted single sym. It will have type `11h` and count 1 if and only if it is an enlisted single symbol.

The `ereptest` function above performs this check, with `ereplace` performing the replacement.

Console size

`.Q.s1` is dependent on the size of the console so make it larger if necessary.

Since we are going to be checking a parse tree which may contain parse trees nested to arbitrary depth, we need a way to check all the elements down to the base level.

We observe that a parse tree is a general list, and therefore of type `0h`. This knowledge combined with the use of `.z.s` allows us to scan a parse tree recursively. The logic goes: if what you have passed into `funcEn` is a parse tree then reapply the function to each element.

To illustrate we examine the following `select` query.

```
q)show pTree:parse "select from trade where sym like \"F*\",not sym=`FD"
?
`trade
,((like;`sym;"F*");(~:;(=;`sym;,`FD))) 0b
()

q)x:eval pTree 2          //apply eval to Where clause
```

Consider the Where clause in isolation.

```
q)x //a 2-list of Where clauses
(like;`sym;"F*")
(~:;(=;`sym;,`FD))

q)funcEn x
(like;`sym;"F*")
(~:;(=;`sym;"enlist`FD"))
```

Similarly we create a function which will replace k functions with their q equivalents in string form, thus addressing the third issue above.

```
q)kreplace:{[x] $[`=qval:.q?x;x;string qval]}
q)funcK:{$[0=t:type x;.z.s each x;t<100h;x;kreplace x]}
```

Running these functions against our Where clause, we see the k representations being converted to q.

```
q)x
(like;`sym;"F*")
(~:;(=;`sym;,`FD))

q)funcK x //replaces ~: with “not”
```

```
(like;`sym;"F*")
("not";(=;`sym;,`FD))
```

Next, we make a slight change to `kreplace` and `ereplace` and combine them.

```
kreplace:{[x] $[`=qval:.q?x;x;"~~",string[qval],"~~"]}
ereplace:{"~~enlist",(.Q.s1 first x),"~~"}
q)funcEn funcK x
(like;`sym;"F*") ("~~not~~";(=;`sym;"~~enlist`FD~~"))
```

The double tilde here is going to act as a tag to allow us to differentiate from actual string elements in the parse tree. This allows us to drop the embedded quotation marks at a later stage inside the `buildQuery` function:

```
q)ssr/[;("\"~~";"~~\"");("";"")] .Q.s1 funcEn funcK x
"((like;`sym;\"F*\");(not;(=;`sym;enlist`FD)))"
```

thus giving us the correct format for the Where clause in a functional select. By applying the same logic to the rest of the parse tree we can write the `buildQuery` function.

```
q)buildQuery "select from trade where sym like \"F*\",not sym=`FD"
"?[trade;((like;`sym;\"F*\");(not;(=;`sym;enlist`FD)));0b;()]"
```

One thing to take note of is that since we use reverse lookup on the `.q` namespace and only want one result we occasionally get the wrong keyword back.

```
q)buildQuery "update tstamp:ltime tstamp from z"
"![z;();0b;(enlist`tstamp)!enlist (reciprocal;`tstamp)]"

q).q`ltime
%:
q).q`reciprocal
%:
```

These instances are rare and a developer should be able to spot when they occur. Of course, the functional form will still work as expected but could confuse readers of the code.

Fifth and sixth arguments

Functional select also has ranks 5 and 6; i.e. fifth and sixth arguments.

☞ *Q for Mortals*: §9.12.1 Functional queries

We also cover these with the `buildQuery` function.

```
q)buildQuery "select[10 20] from trade"
"?[trade;();0b;();10 20]"
q)//5th parameter included
```

The 6th argument is a column and a direction to order the results by. Use `<` for ascending and `>` for descending.

```
q)parse"select[10;<price] from trade"
?
`trade
()
0b
()
10
,(<:;`price)

q).q?(<:;>:)
`hopen`hclose

q)qfind each ("<:";">:")   //qfind defined above
hopen
hclose
```

We see that the k function for the 6th argument of the functional form is `<:` (ascending) or `>:` (descending). At first glance this appears to be `hopen` or `hclose`. In fact in earlier versions of q, `iasc` and `hopen` were equivalent (as were `idesc` and `hclose`). The definitions of `iasc` and `idesc` were later altered to signal a rank error if not applied to a list.

```
q)iasc
k){$[0h>@x;'`rank;<x]}

q)idesc
k){$[0h>@x;'`rank;>x]}

q)iasc 7
'rank
```

Since the columns of a table are lists, it is irrelevant whether the functional form uses the old or new version of `iasc` or `idesc`.

The `buildQuery` function handles the 6th argument as a special case so will produce `iasc` or `idesc` as appropriate.

```
q)buildQuery "select[10 20;>price] from trade"
"?[trade;();0b;();10 20;(idesc;`price)]"
```

The full `buildQuery` function code is as follows:

```
\c 30 200
tidy:{ssr/[;("\"~~";"~~\"");("";"")] $[","=first x;1_x;x]}
strBrk:{y,(";" sv x),z}

//replace k representation with equivalent q keyword
kreplace:{[x] $[`=qval:.q?x;x;"~~",string[qval],"~~"]}
funcK:{$[0=t:type x;.z.s each x;t<100h;x;kreplace x]}

//replace eg ,`FD`ABC`DEF with "enlist`FD`ABC`DEF"
ereplace:{"~~enlist",(.Q.s1 first x),"~~"}
ereptest:{(1=count x) and ((0=type x) and 11=type first x) or 11=type x}
```

```
funcEn:{$[ereptest x;ereplace x;0=type x;.z.s each x;x]}

basic:{tidy .Q.s1 funcK funcEn x}

addbraks:{"(",x,")"}

//Where clause needs to be a list of Where clauses,
//so if only one Where clause, need to enlist.
stringify:{$[(0=type x) and 1=count x;"enlist ";""],basic x}

//if a dictionary, apply to both keys and values
ab:{
  $[(0=count x) or -1=type x; .Q.s1 x;
    99=type x; (addbraks stringify key x ),"!",stringify value x;
    stringify x] }

inner:{[x]
  idxs:2 3 4 5 6 inter ainds:til count x;
  x:@[x;idxs;'[ab;eval]];
  if[6 in idxs;x[6]:ssr/[;("hopen";"hclose");("iasc";"idesc")] x[6]];
  //for select statements within select statements
  x[1]:$[-11=type x 1;x 1;[idxs,:1;.z.s x 1]];
  x:@[x;ainds except idxs;string];
  x[0],strBrk[1_x;"[";"]"] }

buildQuery:{inner parse x}
```

qSQL

Q for Mortals §9.12 Functional Forms

Functional Query Functions

`select`

Select all or part of a table, possibly with new columns

`select` **is a qSQL query template and varies from regular q syntax.**

For the Select operator `?`, see Functional SQL

Syntax

Below, square brackets mark optional elements.

```
select [$L_{exp}$] [$p_s$] [by $p_b$] from $t_{exp}$ [where $p_w$]

where

$L_{exp}$  Limit expression
$p_s$      Select phrase
$p_b$      By phrase
$t_{exp}$  Table expression
$p_w$      Where phrase
```

qSQL syntax

The `select` query returns a table for both call-by-name and call-by-value.

Since 4.1t 2021.03.30, select from partitioned tables maps relevant columns within each partition in parallel when running with secondary threads.

Minimal form

The minimal form of the query returns the evaluated table expression.

```
q)tbl:([] id:1 1 2 2 2;val:100 200 300 400 500)
q)select from tbl
id val
------
1  100
1  200
2  300
2  400
2  500
```

Select phrase

The Select phrase specifies the columns of the result table, one per subphrase.

Absent a Select phrase, all the columns of the table expression are returned. (Unlike SQL, no `*` wildcard is required.)

```
q)t:([] c1:`a`b`c; c2:10 20 30; c3:1.1 2.2 3.3)

q)select c3, c1 from t
c3  c1
------
1.1 a
2.2 b
3.3 c

q)select from t
c1 c2 c3
---------
a  10 1.1
b  20 2.2
c  30 3.3
```

A computed column in the Select phrase cannot be referred to in another subphrase.

Limit expression

To limit the returned results you can include a limit expression L_{exp}

```
select[n]
select[m n]
select[order]
select[n;order]
select distinct
```

where

- `n` limits the result to the first `n` rows of the selection if positive, or the last `n` rows if negative
- `m` is the number of the first row to be returned: useful for stepping through query results one block of `n` at a time
- `order` is a column (or table) and sort order: use `<` for ascending, `>` for descending

```
select[3;>price] from bids where sym=s,size>0
```

This would return the three best prices for symbol `s` with a size greater than 0.

This construct works on in-memory tables but not on memory-mapped tables loaded from splayed or partitioned files.

Performance

`select[n]` applies the Where phrase on all rows of the table, and takes the first `n` rows, before applying the Select phrase.

So if you are paging it is better to store the result of the query somewhere and `select[n,m]` from there, rather than run the filter again.

`select distinct` returns only unique records in the result.

By phrase

A `select` query that includes a By phrase returns a keyed table. The key columns are those in the By phrase; values from other columns are grouped, i.e. nested.

```
q)k:`a`b`a`b`c
q)v:10 20 30 40 50

q)select c2 by c1 from ([]c1:k;c2:v)
c1| c2
--| -----
a | 10 30
b | 20 40
c | ,50

q)v group k   / compare the group keyword
a| 10 30
b| 20 40
c| ,50
```

Unlike in SQL, columns in the By phrase

- are included in the result and need not be specified in the Select phrase
- can include computed columns

The SQL `GROUP BY` statement

The `ungroup` keyword reverses the grouping, though the original order is lost.

```
q)ungroup select c2 by c1 from ([]c1:k;c2:v)
c1 c2
-----
a  10
a  30
b  20
b  40
c  50
```

```
q)t:([] name:`tom`dick`harry`jack`jill;sex:`m`m`m`m`f;eye:`blue`green`blue`blue`gray)
q)t
```

```
name  sex eye
---------------
tom   m   blue
dick  m   green
harry m   blue
jack  m   blue
jill  f   gray

q)select name,eye by sex from t
sex| name                 eye
---| ------------------------------------------
f  | ,`jill               ,`gray
m  | `tom`dick`harry`jack `blue`green`blue`blue

q)select name by sex,eye from t
sex eye  | name
---------| ---------------
f   gray | ,`jill
m   blue | `tom`harry`jack
m   green| ,`dick
```

A By phrase with no Select phrase returns the last row in each group.

```
q)select by sex from t
sex| name eye
---| ---------
f  | jill gray
m  | jack blue
```

Where there is a By phrase, and no sort order is specified, the result is sorted ascending by its key.

Cond

Cond is not supported inside query templates: see qSQL.

`delete`, `exec`, `update`
qSQL, Functional SQL
Q for Mortals §9.3 The `select` Template

exec

Return selected rows and columns from a table

`exec` **is a qSQL query template and varies from regular q syntax.**

For the Exec operator `?`, see Functional SQL

Syntax

Below, square brackets mark optional elements.

```
exec [distinct] $p_s$ [by $p_b$] from $t_{exp}$ [where $p_w$]
```

qSQL syntax

From phrase

The table expression t_{exp} may be a table in memory, or on disk, where it may be splayed but not partitioned.

The workaround is to use the result of a `select` query as the table expression:

```
exec … from select … from …
```

Select phrase

Where the Select phrase

- is omitted, returns the last record
- contains a single column, returns the value of that column
- contains multiple columns or assigns a column name, returns a dictionary with column names as keys

```
q)\l sp.q

q)exec from sp  / last record
s  | `s!0
p  | `p$`p5
qty| 400

q)exec qty from sp  / list
300 200 400 200 100 100 300 400 200 200 300 400
```

```
q)exec amount:qty from sp  / assigns column name
amount| 300 200 400 200 100 100 300 400 200 200 300 400

q)exec (qty;s) from sp  / list per column
300 200 400 200 100 100 300 400 200 200 300 400
s1  s1  s1  s1  s4  s1  s2  s2  s3  s4  s4  s1

q)exec qty, s from sp  / dict by column name
qty| 300 200 400 200 100 100 300 400 200 200 300 400
s  | s1  s1  s1  s1  s4  s1  s2  s2  s3  s4  s4  s1

q)exec sum qty by s from sp  / dict by key
s1| 1600
s2| 700
s3| 200
s4| 600

q)exec q:sum qty by s from sp  / xtab:list!table
  | q
--| ----
s1| 1600
s2| 700
s3| 200
s4| 600

q)exec sum qty by s:s from sp  / table!list
s |
--| ----
s1| 1600
s2| 700
s3| 200
s4| 600

q)exec qty, s by 0b from sp  / table
qty s
------
300 s1
200 s1
400 s1
200 s1
100 s4
100 s1
300 s2
400 s2
200 s3
200 s4
300 s4
400 s1

q)exec q:sum qty by s:s from sp
s | q
--| ----
s1| 1600
s2| 700
s3| 200
s4| 600
```

Compare the results of `select` and `exec` queries with multiple columns:

- a `select` query result is a table, and all columns are necessarily the same length
- an `exec` query result is a dictionary, and column lengths can vary

```
q)t
name  sex eye
---------------
tom   m   blue
dick  m   green
harry m   blue
jack  m   blue
jill  f   gray
q)select name, distinct eye from t
'length
  [0]  select name, distinct eye from t
       ^
q)exec name, distinct eye from t
name| `tom`dick`harry`jack`jill
eye | `blue`green`gray
```

Limit expression

`exec distinct` returns only unique items in the first item of the result.

```
q)exec distinct s,p,s from sp
s | `s$`s1`s4`s2`s3
p | `p$`p1`p2`p3`p4`p5`p6`p1`p2`p2`p2`p4`p5
s1| `s$`s1`s1`s1`s1`s4`s1`s2`s2`s3`s4`s4`s1
```

Cond

Cond is not supported inside query templates: see qSQL.

`delete`, `select`, `update`
qSQL, Functional SQL
Q for Mortals §9.4 The `exec` Template

`update`

Add or amend rows or columns of a table or entries in a dictionary

`update` is a qSQL query template and varies from regular q syntax.

For the Update operator `!`, see Functional SQL

Since 4.1t 2021.06.04 updates from splayed table and path@tablename now leverage peach to load columns (when running with secondary threads).

```
q)update x:0 from get`:mysplay
```

Syntax

```
update $p_s$ [by $p_b$] from $t_{exp}$ [where $p_w$]
```

qSQL query templates

From phrase

`update` will not modify a splayed table on disk.

Select phrase

Names in the Select phrase refer to new or modified columns in the table expression.

```
q)t:([] name:`tom`dick`harry; age:28 29 35)
q)update eye:`blue`brown`green from t
name  age eye
---------------
tom   28  blue
dick  29  brown
harry 35  green
```

Where phrase

The Where phrase restricts the scope of updates.

```
q)t:([] name:`tom`dick`harry; hair:`fair`dark`fair; eye:`green`brown`gray)
q)t
name  hair eye
----------------
tom   fair green
dick  dark brown
harry fair gray

q)update eye:`blue from t where hair=`fair
name  hair eye
----------------
tom   fair blue
dick  dark brown
harry fair blue
```

New values must have the type of the column being amended.

If the query adds a new column it will have values only as determined by the Where phrase. At other positions, it will have nulls of the column's type.

By phrase

The By phrase applies the update along groups. This is most useful with aggregate and uniform functions.

With an aggregate function, the entire group gets the value of the aggregation on the group.

```
q)update avg weight by city from p
p | name  color weight city
--| -------------------------
p1| nut   red   15     london
p2| bolt  green 14.5   paris
p3| screw blue  17     rome
p4| screw red   15     london
p5| cam   blue  14.5   paris
p6| cog   red   15     london
```

A uniform function is applied along the group in place. This can be used, for example, to compute cumulative volume of orders.

```
q)update cumqty:sums qty by s from sp
s p  qty cumqty
---------------
0 p1 300 300
0 p2 200 500
0 p3 400 900
0 p4 200 1100
3 p5 100 100
0 p6 100 1200
1 p1 300 300
1 p2 400 700
2 p2 200 200
3 p2 200 300
```

```
3 p4 300 600
0 p5 400 1600
```

Since 4.1 2024.04.29 throws `type` error if dictionary update contains by clause (previously ignored).

Cond

Cond is not supported inside query templates: see qSQL.

`delete`, `exec`, `select`

qSQL, Functional SQL

Q for Mortals §9.5 The `update` template

`delete`

Delete rows or columns from a table, entries from a dictionary, or objects from a namespace

```
delete    from x
delete    from x where pw
delete ps from x
```

`delete` is a qSQL query template and varies from regular q syntax

For the Delete operator `!` , see 📖 Functional SQL

Table rows

```
delete    from x
delete    from x where pw
```

Where

- `x` is a table
- `pw` is a condition

deletes from `x` rows matching `pw` , or all rows if `where pw` not specified.

```
q)show table: ([] a: `a`b`c; n: 1 2 3)
a n
---
a 1
b 2
c 3
q)show delete from table where a = `c
a n
---
a 1
b 2
```

Attributes may or may not be dropped: reapply or remove as needed

Table columns

```
delete    from x
delete ps from x
```

Where

- `x` is a table
- `ps` a list of column names

deletes from `x` columns `ps` or all columns if `ps` not specified.

```
q)show delete n from table
a
-
a
b
c
```

Dictionary entries

```
delete    from x
delete ps from x
```

Where

- `x` is a dictionary
- `ps` a list of keys to it

deletes from `x` entries for `ps` .

```
q)show d:`a`b`c!til 3
a| 0
b| 1
c| 2
q)delete b from `d
`d
q)d
a| 0
c| 2
```

⚠ **Cond is not supported inside q-SQL expressions**

Enclose in a lambda or use Vector Conditional instead.

☞ qSQL

Namespace objects

```
delete    from x
delete ps from x
```

Where

- `x` is a namespace
- `ps` a symbol atom or vector of name/s defined in it

deletes the named objects from the namespace.

```
q)a:1
q)\v
,`a
q)delete a from `.
`.
q)\v
`symbol$()
```

☛ qSQL

upsert

Overwrite or append records to a table

```
x upsert y    upsert[x;y]
```

Where

- `x` is a table, or the name of a table as a symbol atom, or the name of a splayed table as a directory handle
- `y` is zero or more records

the records are upserted into the table.

The record/s `y` may be either

- lists with types that match `type each x cols x`
- a table with columns that are members of `cols x` and have corresponding types

If `x` is the name of a table, it is updated in place. Otherwise the updated table is returned.

If `x` is the name of a table as a symbol atom (or the name of a splayed table as a directory handle) that does not exist in the file system, it is written to file.

Simple table

If the table is simple, new records are appended. If the records are in a table, it must be simple.

```
q)t:([]name:`tom`dick`harry;age:28 29 30;sex:`M)

q)t upsert (`dick;49;`M)
name  age sex
-------------
tom   28  M
dick  29  M
harry 30  M
dick  49  M

q)t upsert((`dick;49;`M);(`jane;23;`F))
name  age sex
-------------
tom   28  M
dick  29  M
harry 30  M
dick  49  M
jane  23  F

q)`t upsert ([]age:49 23;name:`dick`jane)
```

```
`t
q)t
name  age sex
-------------
tom   28  M
dick  29  M
harry 30  M
dick  49
jane  23
```

Keyed table

If the table is keyed, any new records that match on key are updated. Otherwise, new records are inserted.

If the right argument is a table it may be keyed or unkeyed.

```
q)a upsert (`e;30;70)                         / single record
s| r  u
-| -----
q| 1  5
w| 2  6
e| 30 70

q)a upsert ((`e;30;70);(`r;40;80))            / multiple records
s| r  u
-| -----
q| 1  5
w| 2  6
e| 30 70
r| 40 80

q)show a:([]s:`q`w`e;r:1 2 3;u:5 6 7)          / simple table
s| r u
-| ---
q| 1 5
w| 2 6
e| 3 7

q)/update `q and `e, insert new `r; return new table
q)a upsert ([s:`e`r`q]r:30 4 10;u:70 8 50)    / keyed table
s| r  u
-| -----
q| 10 50
w| 2  6
e| 30 70
r| 4  8

q)`a upsert ([s:`e`r`q]r:30 4 10;u:70 8 50)   / same but update table in place
`a
```

Serialized table

```
q)`:data/tser set ([] c1:`a`b; c2:1.1 2.2)
`:data/tser
q)`:data/tser upsert (`c; 3.3)
`:data/tser

q)get `:data/tser
c1 c2
------
a  1.1
b  2.2
c  3.3
```

Upserting to a serialized table reads the table into memory, updates it, and writes it back to file.

Splayed table

```
q)`:data/tsplay/ set ([] c1:`sym?`a`b; c2:1.1 2.2)
`:data/tsplay/
q)`:data/tsplay upsert (`sym?`c; 3.3)
`:data/tsplay
q)select from `:data/tsplay
c1 c2
------
a  1.1
b  2.2
c  3.3
```

Upserting to a splayed table appends new values to the column files.

> **Upserting to a serialized or splayed table removes any attributes set.**

> **Cond is not supported inside q-SQL expressions**
>
> Enclose in a lambda or use Vector Conditional instead.

`insert`, Join
Joins, qSQL, Tables
Q for Mortals §9.2 Upsert

`view` , `views`

`view`

Expression defining a view

```
view x     view[x]
```

Where `x` is a view (by reference), returns the expression defining `x`.

```
q)v::2+a*3                        / define dependency v
q)a:5
q)v
17
q)view `v                         / view the dependency expression
"2+a*3"
```

`views`

List views defined in the default namespace

```
views[]
```

Returns a sorted list of the views currently defined in the default namespace.

```
q)w::b*10
q)v::2+a*3
q)views[]
`s#`v`w
```

Metadata
Views
Q for Mortals §4.11 Alias

fby

Apply an aggregate to groups

```
(aggr;d) fby g
```

Where

- `aggr` is an aggregate function
- `d` and `g` are conforming vectors

collects the items of `d` into sublists according to the corresponding items of `g`, applies `aggr` to each sublist, and returns the results as a vector with the same count as `d`.

When to use `fby`

`fby` is designed to collapse cascaded

`select … from select … by … from t`

expressions into a single

`select … by … from … where … fby …`

Think of `fby` when you find yourself trying to apply a filter to the aggregated column of a table produced by `select … by …`.

```
q)show dat:10?10
4 9 2 7 0 1 9 2 1 8
q)grp:`a`b`a`b`c`d`c`d`d`a
q)(sum;dat) fby grp
14 16 14 16 9 4 9 4 4 14
```

Collect the items of `dat` into sublists according to the items of `grp`.

```
q)group grp
a| 0 2 9
b| 1 3
c| 4 6
d| 5 7 8

q)dat group grp
a| 4 2 8
b| 9 7
c| 0 9
d| 1 2 1
```

Apply `aggr` to each sublist.

```
q)sum each dat group grp
a| 14
b| 16
c| 9
d| 4
```

The result is created by replacing each item of `grp` with the result of applying `aggr` to its corresponding sublist.

```
q)(sum;dat) fby grp
14 16 14 16 9 4 9 4 4 14
q)(sum each dat group grp)grp / w/o fby
14 16 14 16 9 4 9 4 4 14
```

Vectors

```
q)dat:2 5 4 1 7             / data
q)grp:"abbac"               / group by
q)(sum;dat) fby grp         / apply sum to the groups
3 9 9 3 7
q)(first;dat) fby grp       / apply first to the groups
2 5 5 2 7
```

Tables

When used in a `select`, usually a comparison function is applied to the results of `fby`, e.g.

```
select from t where 10 < (sum;d) fby a
```

```
q)\l sp.q
q)show sp                                       / for reference
s  p  qty
---------
s1 p1 300
s1 p2 200
s1 p3 400
s1 p4 200
s4 p5 100
s1 p6 100
s2 p1 300
s2 p2 400
s3 p2 200
s4 p2 200
s4 p4 300
s1 p5 400
```

Sales where quantity > average quantity by part:

```
q)select from sp where qty > (avg;qty) fby p
s  p  qty
---------
s2 p2 400
s4 p4 300
s1 p5 400
```

Sales where quantity = maximum quantity by part:

```
q)select from sp where qty = (max;qty) fby p
s  p  qty
---------
s1 p1 300
s1 p3 400
s1 p6 100
s2 p1 300
s2 p2 400
s4 p4 300
s1 p5 400
```

To group on multiple columns, tabulate them in `g`.

```
q)update x:12?3 from `sp
`sp
q)sp
s  p  qty x
-----------
s1 p1 300 0
s1 p2 200 2
s1 p3 400 0
s1 p4 200 1
s4 p5 100 0
s1 p6 100 0
s2 p1 300 0
s2 p2 400 2
s3 p2 200 2
s4 p2 200 2
s4 p4 300 1
s1 p5 400 1

q)select from sp where qty = (max;qty) fby ([]s;x)
s  p  qty x
-----------
s1 p2 200 2
s1 p3 400 0
s4 p5 100 0
s2 p1 300 0
s2 p2 400 2
s3 p2 200 2
s4 p2 200 2
s4 p4 300 1
s1 p5 400 1
```

`fby` before V2.7

In V2.6 and below, `fby`'s behavior is undefined if the aggregation function returns a list; it usually signals an error from the k definition of `fby`. However, if the concatenation of all list results from the aggregation function results `raze` has the same length as the original vectors, a list of some form is returned, but the order of its items is not clearly defined.

---

q-SQL

Joins

```
Keyed:                As of:
 ej        equi        aj aj0      as-of
 ij ijf    inner       ajf ajf0
 lj ljf    left        asof        simple as-of
 pj        plus        wj wj1      window
 uj ujf    union
 upsert
 ,         join
 ^         coalesce
```

A *join* combines data from two tables, or from a table and a dictionary.

Some joins are *keyed*, in that columns in the first argument are matched with the key columns of the second argument.

Some joins are *as-of*, where a time column in the first argument specifies corresponding intervals in a time column of the second argument. Such joins are not keyed.

In each case, the result has the merge of columns from both arguments. Where necessary, rows are filled with nulls or zeroes.

Keyed joins

`^` Coalesce

The Coalesce operator merges keyed tables ignoring nulls

`ej` Equi join

Similar to `ij`, where the columns to be matched are given as a parameter.

`ij` `ijf` Inner join

Joins on the key columns of the second table. The result has one row for each row of the first table that matches the key columns of the second table.

`,` Join

The Join operator `,` joins tables and dictionaries as well as lists. For tables `x` and `y`:

- `x,y` is `x upsert y`
- `x,'y` joins records to records
- `x,\:y` is `x lj y`

`lj` `ljf` Left join

> Outer join on the key columns of the second table. The result has one row for each row of the first table. Null values are used where a row of the first table has no match in the second table. This is now built-in to `,\:` . (Reverse the arguments to make a right outer join.)

`pj` Plus join

> A variation on left join. For each matching row, values from the second table are added to the first table, instead of replacing values from the first table.

`uj` `ujf` Union join

> Uses all rows from both tables. If the second table is not keyed, the result is the catenation of the two tables. Otherwise, the result is the left join of the tables, catenated with the unmatched rows of the second table.

`upsert`

> Can be used to join two tables with matching columns (as well as add new records to a table). If the first table is keyed, any records that match on key are updated. The remaining records are appended.

As-of joins

In each case, the time column in the first argument specifies [) intervals in the second argument.

`wj, wj1` Window join

> The most general forms of as-of join. Function parameters aggregate values in the time intervals of the second table. In `wj` , prevailing values on entry to each interval are considered. In `wj1` , only values occurring within each interval are considered.

`aj,aj0,ajf,ajf0` As-of join

> Simpler window joins where only the last value in each interval is used. In the `aj` result, the time column is from the first table, while in the `aj0` result, the time column is from the second table.

`asof`

> A simpler `aj` where all columns (or dictionary keys) of the second argument are used in the join.

Implicit joins

A foreign key is made by enumerating over the column/s of a keyed table.

Where a primary key table `m` has a key column `k` and a table `d` has a column `c` and foreign key linking to `k` , a left join is implicit in the query

```
select m.k, c from d
```

This generalizes to multiple foreign keys in `d`.

Suppliers and parts database `sp.q`

```
q)\l sp.q
+`p`city!(`p$`p1`p2`p3`p4`p5`p6`p1`p2;`london`london`london`london`london`lon..
(`s#+(,`color)!,`s#`blue`green`red)!+(,`qty)!,900 1000 1200
+`s`p`qty!(`s$`s1`s1`s1`s2`s3`s4;`p$`p1`p4`p6`p2`p2`p4;300 200 100 400 200 300)

q)select sname:s.name, qty from sp
sname qty
---------
smith 300
smith 200
smith 400
smith 200
clark 100
smith 100
jones 300
jones 400
blake 200
clark 200
clark 300
smith 400
```

Implicit joins extend to the situation in which the targeted keyed table itself has a foreign key to another keyed table.

```
q)emaster:([eid:1001 1002 1003 1004 1005] currency:`gbp`eur`eur`gbp`eur)
q)update eid:`emaster$1001 1002 1005 1004 1003 from `s
`s

q)select s.name, qty, s.eid.currency from sp
name  qty currency
------------------
smith 300 gbp
smith 200 gbp
smith 400 gbp
smith 200 gbp
clark 100 gbp
smith 100 gbp
jones 300 eur
jones 400 eur
blake 200 eur
clark 200 gbp
clark 300 gbp
smith 400 gbp
```

Q for Mortals §9.9.1 Implicit Joins

Q for Mortals §9.9 Joins

lj , ljf

Left join

```
x lj  y     lj [x;y]
x ljf y     ljf[x;y]
```

Where

- x is a table. Since 4.1t 2023.08.04 if x is the name of a table, it is updated in place.
- y is
  - a keyed table whose key column/s are columns of x , returns x and y joined on the key columns of y
  - or the general empty list () , returns x

For each record in x , the result has one record with the columns of y joined to columns of y :

- if there is a matching record in y , it is joined to the x record; common columns are replaced from y .
- if there is no matching record in y , common columns are left unchanged, and new columns are null

```
q)show x:([]a:1 2 3;b:`I`J`K;c:10 20 30)
a b c
------
1 I 10
2 J 20
3 K 30

q)show y:([a:1 3;b:`I`K]c:1 2;d:10 20)
a b| c d
---| ----
1 I| 1 10
3 K| 2 20

q)x lj y
a b c  d
---------
1 I 1  10
2 J 20
3 K 2  20
```

The y columns joined to x are given by:

```
q)y[select a,b from x]
c d
----
1 10
2 20
```

`lj` is a multithreaded primitive.

Changes in V4.0

`lj` checks that `y` is a keyed table. (Since V4.0 2020.03.17.)

```
q)show x:([]a:1 2 3;b:10 20 30)
a b
----
1 10
2 20
3 30
q)show y:([]a:1 3;b:100 300)
a b
-----
1 100
3 300
q)show r:([]a:1 2 3;b:100 20 300)
a b
-----
1 100
2 20
3 300

q)(1!r)~(1!x)lj 1!y
1b
q)r~x lj 1!y
1b

q)x lj y
'type
  [0]  x lj y
          ^
```

Changes in V3.0

Since V3.0, the `lj` operator is a cover for `,\:` (Join Each Left) that allows the left argument to be a keyed table. `,\:` was introduced in V2.7 2011.01.24.

Prior to V3.0, `lj` had similar behavior, with one difference - when there are nulls in the right argument, `lj` in V3.0 uses the right-argument null, while the earlier version left the corresponding value in the left argument unchanged:

```
q)show x:([]a:1 2;b:`x`y;c:10 20)
a b c
------
1 x 10
2 y 20
q)show y:([a:1 2]b:``z;c:1 0N)
a| b c
-| ---
1|   1
2| z
q)x lj y        / kdb+ 3.0
a b c
-----
1   1
2 z
q)x lj y        / kdb+ 2.8
a b c
------
1 x 1
2 z 20
```

Since 2014.05.03, the earlier version is available in all V3.x versions as `ljf`.

Joins
Q for Mortals §9.9.2 Ad Hoc Left Join

`ij`, `ijf`

Inner join

```
x ij  y     ij [x;y]
x ijf y     ijf[x;y]
```

Where

- `x` and `y` are tables
- `y` is keyed, and its key columns are columns of `x`

returns two tables joined on the key columns of the second table. The result has one combined record for each row in `x` that matches a row in `y`.

```
q)t
sym  price
---------------
IBM  0.7029677
FDP  0.08378167
FDP  0.06046216
FDP  0.658985
IBM  0.2608152
MSFT 0.5433888

q)s
sym | ex  MC
----| --------
IBM | N   1000
MSFT| CME 250

q)t ij s
sym  price     ex  MC
-----------------------
IBM  0.7029677 N   1000
IBM  0.2608152 N   1000
MSFT 0.5433888 CME 250
```

Common columns are replaced from `y`.

```
q)([] k:1 2 3 4; v:10 20 30 40) ij ([k:2 3 4 5]; v:200 300 400 500;s:`a`b`c`d)
k v   s
-------
2 200 a
3 300 b
4 400 c
```

`ij` is a multithreaded primitive.

Changes in V3.0

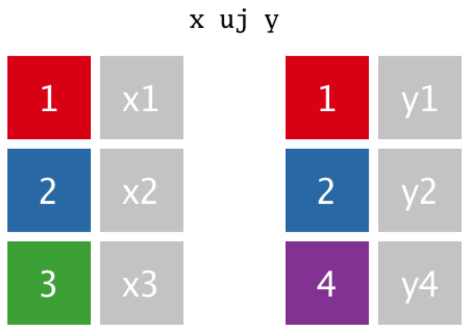
Since V3.0, `ij` has changed behavior (similarly to `lj`): when there are nulls in `y`, `ij` uses the `y` null, where the earlier version left the corresponding value in `x` unchanged:

```
q)show x:([]a:1 2;b:`x`y;c:10 20)
a b c
------
1 x 10
2 y 20
q)show y:([a:1 2]b:``z;c:1 0N)
a| b c
-| ---
1|   1
2| z
q)x ij y        /V3.0
a b c
-----
1   1
2 z
q)x ij y        /V2.8
a b c
------
1 x 1
2 z 20
```

Since 2016.02.17, the earlier version is available in all V3.4 and later versions as `ijf`.

Joins
Q for Mortals §9.9.4 Ad Hoc Inner Join

uj , ujf



Union join

```
x uj  y     uj [x;y]
x ujf y     ujf[x;y]
```

Where `x` and `y` are both keyed or both unkeyed tables, returns the union of the columns, filled with nulls where necessary:

- if `x` and `y` have matching key column/s, then records in `y` update matching records in `x`
- otherwise, `y` records are inserted.

```
q)show s:([]a:1 2;b:2 3;c:5 7)
a b c
-----
1 2 5
2 3 7

q)show t:([]a:1 2 3;b:2 3 7;c:10 20 30;d:"ABC")
a b c  d
--------
1 2 10 A
2 3 20 B
3 7 30 C

q)s,t                            / tables do not conform for ,
'mismatch

q)s uj t                         / simple, so second table is inserted
a b c  d
--------
1 2 5
2 3 7
1 2 10 A
2 3 20 B
3 7 30 C

q)(2!s) uj 2!t                   / keyed, so matching records are updated
a b| c  d
---| ----
1 2| 10 A
2 3| 20 B
3 7| 30 C
```

`uj` is a multithreaded primitive.

`uj` **generalizes the** **`,` Join** **operator.**

Changes in V3.0

The union join of two keyed tables is equivalent to a left join of the two tables with the catenation of unmatched rows from the second table.

As a result a change in the behavior of `lj` causes a change in the behavior of `uj`:

```
q)show x:([a:1 2]b:`x`y;c:10 20)
a| b c
-| ----
1| x 10
2| y 20
q)show y:([a:1 2]b:``z;c:1 0N)
a| b c
-| ---
1|   1
2| z
q)x uj y        / kdb+ 3.0
a| b c
-| ---
1|   1
2| z
q)x uj y        / kdb+ 2.8
a| b c
-| ----
1| x 1
2| z 20
```

Since 2017.04.10, the earlier version is available in all V3.5 and later versions as `ujf`.

Joins
Q for Mortals §9.9.7 Union Join

aj , aj0 , ajf , ajf0

As-of join

```
aj  [c; t1; t2]
aj0 [c; t1; t2]
ajf [c; t1; t2]
ajf0[c; t1; t2]
```

Where

- `t1` is a table. Since 4.1t 2023.08.04 if `t1` is the name of a table, it is updated in place.
- `t2` is a simple table
- `c` is a symbol list of `n` column names, common to `t1` and `t2` , and of matching type
- column `c[n]` is of a sortable type (typically time)

returns a table with records from the left-join of `t1` and `t2` . In the join, columns `c[til n-1]` are matched for equality, and the last value of `c[n]` (most recent time) is taken. For each record in `t1` , the result has one record with the items in `t1` , and

- if there are matching records in `t2` , the items of the last (in row order) matching record are appended to those of `t1` ;
- otherwise the remaining columns are null.

```
q)t:([]time:10:01:01 10:01:03 10:01:04;sym:`msft`ibm`ge;qty:100 200 150)
q)t
time       sym  qty
-----------------
10:01:01 msft 100
10:01:03 ibm  200
10:01:04 ge   150

q)q:([]time:10:01:00 10:01:00 10:01:00 10:01:02;sym:`ibm`msft`msft`ibm;px:100 99 101 98)
q)q
time     sym  px
-----------------
10:01:00 ibm  100
10:01:00 msft 99
10:01:00 msft 101
10:01:02 ibm  98

q)aj[`sym`time;t;q]
time       sym  qty px
---------------------
10:01:01 msft 100 101
```

```
10:01:03 ibm  200 98
10:01:04 ge   150
```

`aj` is a multithreaded primitive.

There is no requirement for any of the join columns to be keys but the join is faster on keys.

`aj`, `aj0`

`aj` and `aj0` return different times in their results:

```
aj     boundary time from t1
aj0    actual time from t2
```

`ajf`, `ajf0`

Since V3.6 2018.05.18 `ajf` and `ajf0` behave as V2.8 `aj` and `aj0`, i.e. they fill from LHS if RHS is null. e.g.

```
q)t0:([]time:2#00:00:01;sym:`a`b;p:1 1;n:`r`s)
q)t1:([]time:2#00:00:01;sym:`a`b;p:0 1)
q)t2:([]time:2#00:00:00;sym:`a`b;p:1 0N;n:`r`s)
q)t0~ajf[`sym`time;t1;t2]
1b
```

Performance

Order of search columns

Ensure the first argument to `aj`, the columns to search on, is in the correct order, e.g. `` `sym`time ``. Otherwise you'll suffer a severe performance hit.

If the resulting time value is to be from the quote (actual time) instead of the (boundary time) from trade, use `aj0` instead of `aj`.

`aj` should run at a million or two trade records per second; whether the tables are mapped or not is irrelevant. However, for speed:

medium	t2[c_1]	t2[c_2...]	example
memory	g#	sorted within c_1	`quote` has `` `g#sym `` and `time` sorted within `sym`
disk	p#	sorted within c_1	`quote` has `` `p#sym `` and `time` sorted within `sym`

Departure from this incurs a severe performance penalty.

Note that, on disk, the `g#` attribute does not help.

> ⚠ **Select the virtual partition column only if you need it. It is fabricated on demand, which can be slow for large partitions.**

`select` from `t2`

In memory, there is no need to select from `t2`. Irrespective of the number of records, use, e.g.:

```
aj[`sym`time;select … from trade where …;quote]
```

instead of

```
aj[`sym`time;select … from trade where …;
             select … from quote where …]
```

In contrast, on disk you must map in your splay or day-at-a-time partitioned database:

Splay:

```
aj[`sym`time;select … from trade where …;select … from quote]
```

Partitioned:

```
aj[`sym`time;select … from trade where …;
             select … from quote where date = …]
```

> ⚠ **If further `where` constraints are used, the columns will be *copied* instead of mapped into memory, slowing down the join.**

If you are using a database where an individual day's data is spread over multiple partitions the on-disk `p#` will be lost when retrieving data with a constraint such as `…date=2011.08.05`. In this case you will have to reduce the number of quotes retrieved by applying further constraints – or by re-applying the attribute.

---

asof
Joins
Q for Mortals §9.9.8 As-of Joins

`wj`, `wj1`

Window join

```
wj [w; c; t; (q; (f0;c0); (f1;c1))]
wj1[w; c; t; (q; (f0;c0); (f1;c1))]
```

Where

- `t` and `q` are simple tables to be joined (`q` should be sorted `` `sym`time `` with `` `p# `` on sym). Since 4.1t 2023.08.04 if `t` is the name of a table, it is updated in place.
- `w` is a pair of lists of times/timestamps, begin and end
- `c` are the names of the common columns, syms and times, which must have integral types
- `f0`, `f1` are aggregation functions applied to values in q columns `c0`, `c1` over the intervals

returns for each record in `t`, a record with additional columns `c0` and `c1`, which are the results of the aggregation functions applied to values over the matching intervals in `w`.

Typically this might be:

```
wj[w;`sym`time;trade;(quote;(max;`ask);(min;`bid))]
```

A quote is understood to be in existence until the next quote.

To see all the values in each window, pass the identity function `::` in place of the aggregates

E.g. `wj[w;c;t;(q;(::;c0);(::;c1))]`

Multi-column arguments

Since 3.6 2018.12.24, `wj` and `wj1` support multi-col args, forming the resulting column name from the last argument e.g.

```
wj[w; f; t; (q; (wavg;`asize;`ask); (wavg;`bsize;`bid))]
```

Interval behavior

`wj` and `wj1` are both [] interval, i.e. they consider quotes ≥beginning and ≤end of the interval.

For `wj`, the prevailing quote on entry to the window is considered valid as quotes are a step function.

`wj1` considers quotes on or after entry to the window. If the join is to consider quotes that arrive from the beginning of the interval, use `wj1`.

Behavior prior to V3.0

Prior to V3.0, `wj1` considered only quotes in the window except for the window end (i.e. quotes ≥start and <end of the interval).

version	wj1	wj
3.0+	`[]`	prevailing + `[]`
2.7/2.8	`[)`	prevailing + `[]`

W Notation for intervals

```
q)t:([]sym:3#`ibm;time:10:01:01 10:01:04 10:01:08;price:100 101 105)
q)t
sym time     price
------------------
ibm 10:01:01 100
ibm 10:01:04 101
ibm 10:01:08 105

q)a:101 103 103 104 104 107 108 107 108
q)b:98 99 102 103 103 104 106 106 107
q)q:([]sym:`ibm; time:10:01:01+til 9; ask:a; bid:b)
q)q
sym time     ask bid
--------------------
ibm 10:01:01 101 98
ibm 10:01:02 103 99
ibm 10:01:03 103 102
ibm 10:01:04 104 103
ibm 10:01:05 104 103
ibm 10:01:06 107 104
ibm 10:01:07 108 106
ibm 10:01:08 107 106
ibm 10:01:09 108 107

q)f:`sym`time
q)w:-2 1+\:t.time

q)wj[w;f;t;(q;(max;`ask);(min;`bid))]
sym time     price ask bid
--------------------------
ibm 10:01:01 100   103 98
ibm 10:01:04 101   104 99
ibm 10:01:08 105   108 104
```

The interval values may be seen as:

```
q)wj[w;f;t;(q;(::;`ask);(::;`bid))]
sym time     price ask              bid
--------------------------------------------------
ibm 10:01:01 100   101 103          98 99
ibm 10:01:04 101   103 103 104 104 99 102 103 103
ibm 10:01:08 105   107 108 107 108 104 106 106 107
```

Window joins with multiple symbols should be used only with a `` `p#sym `` like schema.

Typical RTD-like `` `g# `` gives undefined results.

Window join is a generalization of as-of join

An as-of join takes a snapshot of the current state, while a window join aggregates all values of specified columns within intervals.

`aj`, `asof`
Joins
Q for Mortals §9.9.9 Window Joins

asof

As-of join

```
t1 asof t2     asof[t1;t2]
```

Where

- `t1` is a table
- `t2` is a table or dictionary
- the last key or column of `t2` corresponds to a time column in `t1`

returns the values from the last rows matching the rest of the keys and time ≤ the time in `t2`.

```
q)show trade asof`sym`time!(`IBM;09:30:00.0)
price| 96.3e
size | 200
stop | 0b
corr | 0
cond | "T"
ex   | "D"

q)show trade asof([]sym:`AAPL`IBM;ex:"TD";time:09:30:00.0)
price size stop corr cond
-------------------------
78.14 100  0    0    T
96.3  200  0    0    T
```

The following examples use the `mas` table from TAQ.

```
q)`date xasc`mas       / sort by date
`mas

q)show a!mas asof a:([]sym:`A`B`C`GOOG;date:1995.01.01)
sym  date      | cusip     name                           wi ex uot
---------------| --------------------------------------------------
A    1995.01.01| 049870207 ATTWOODS PLC ADS REP5 ORD/5PNC 0  N  100
B    1995.01.01| 067806109 BARNES GROUP INCORPORATED      0  N  100
C    1995.01.01| 171196108 CHRYSLER CORP                  0  N  100
GOOG 1995.01.01|                                          0

q)show a!mas asof a:([]sym:`A`B`C`GOOG;date:2006.01.01)
sym  date      | cusip     name                      wi ex uot
---------------| ---------------------------------------------
A    2006.01.01| 00846U101 AGILENT TECHNOLOGIES, INC 0  N  100
B    2006.01.01| 067806109 BARNES GROUP INCORPORATED 0  N  100
C    2006.01.01| 172967101 CITIGROUP                 0  N  100
GOOG 2006.01.01| 38259P508 GOOGLE INC CLASS A        0  T  100
```

```
q)show a!mas asof a:([]sym:`A;date:1993.01.05 1996.05.23 2000.08.04)
sym date      | cusip     name                           wi ex uot
--------------| --------------------------------------------------
A   1993.01.05| 049870207 ATTWOODS PLC ADS REP5 ORD/5PNC 0  N  100
A   1996.05.23| 046298105 ASTRA AB CL-A ADS 1CL-ASEK2.50 0  N  100
A   2000.08.04| 00846U101 AGILENT TECHNOLOGIES  INC      0  N  100
```

`asof` is a multithreaded primitive.

`aj`, `wj`
Joins
Q for Mortals §9.9.8 As-of Joins

pj

Plus join

```
x pj y     pj[x;y]
```

Where

- `x` and `y` are tables. Since 4.1t 2023.08.04 if `x` is the name of a table, it is updated in place.
- `y` is keyed
- the key column/s of `y` are columns of `x`

returns `x` and `y` joined on the key columns of `y`.

`pj` adds matching records in `y` to those in `x`, by adding common columns, other than the key columns. These common columns must be of appropriate types for addition.

For each record in `x`:

- if there is a matching record in `y` it is added to the `x` record.
- if there is no matching record in `y`, common columns are left unchanged, and new columns are zero.

```
q)show x:([]a:1 2 3;b:`x`y`z;c:10 20 30)
a b c
------
1 x 10
2 y 20
3 z 30

q)show y:([a:1 3;b:`x`z]c:1 2;d:10 20)
a b| c d
---| ----
1 x| 1 10
3 z| 2 20

q)x pj y
a b c  d
---------
1 x 11 10
2 y 20 0
3 z 32 20
```

In the example above, `pj` is equivalent to `x+0^y[`a`b#x]` (compute the value of `y` on `a` and `b` columns of `x`, fill the result with zeros and add to `x`).

📖 Joins

Q for Mortals §9.9.6 Plus Join

`ej`

Equi join

```
ej[c;t1;t2]
```

Where

- `c` is a list of column names (or a single column name)
- `t1` and `t2` are tables

returns `t1` and `t2` joined on column/s `c`.

The result has one combined record for each row in `t2` that matches `t1` on columns `c`.

```
q)t:([]sym:`IBM`FDP`FDP`FDP`IBM`MSFT;price:0.7029677 0.08378167 0.06046216
    0.658985 0.2608152 0.5433888)
q)s:([]sym:`IBM`MSFT;ex:`N`CME;MC:1000 250)

q)t
sym  price
---------------
IBM  0.7029677
FDP  0.08378167
FDP  0.06046216
FDP  0.658985
IBM  0.2608152
MSFT 0.5433888

q)s
sym  ex  MC
-------------
IBM  N   1000
MSFT CME 250

q)ej[`sym;s;t]
sym  price     ex  MC
-----------------------
IBM  0.7029677 N   1000
IBM  0.2608152 N   1000
MSFT 0.5433888 CME 250
```

Duplicate column values are filled from `t2`.

```
q)t1:([] k:1 2 3 4; c:10 20 30 40)
q)t2:([] k:2 2 3 4 5; c:200 222 300 400 500; v:2.2 22.22 3.3 4.4 5.5)

q)ej[`k;t1;t2]
k c   v
```

```
-----------
2 200 2.2
2 222 22.22
3 300 3.3
4 400 4.4
```

Joins

Q for Mortals §9.9.5 Equi Join