Key Workshop 2023

In this workshop

- importing data and pre-processing for analysis
- key techniques
- inverted tables
- batch processing

Importing data

- Not specified in problem statement
- In the session: can use `]Get` (v18.2)

In [ ]:
```
]get ./order_data.csv
3↑order_data
```

In [ ]:
```
⎕PW←3000
```

In [ ]:
```
1↓30↑order_data[;5]
```

⎕CSV

How can we import numeric data as numbers?

In [ ]:
```
1↓30↑5(⎕⍤1)⎕CSV 'order_data.csv' ⍬ 4
```

Question: When is `⎕CSV path ⍬ 4` dangerous?

- Numeric "codes" e.g. US zip codes
- Telephone numbers
- Hexadecimal `12E056`
- Numbers in a text "description" field
- When using comma `,` as a decimal separator e.g. `3,14 ↔ 3.14`

Safer to use full description.
`⎕CSV path ⍬ (2 1 1 1 2 1)`

In [ ]:
```
]repr 1⎕order_data
```

Exercise

Given a known `col_spec` mapping matrix:

```
    path←'/path/to/order_data.csv'
    col_spec←⍪'payment' 'id' 'city' 'state' 'category' 'timestamp'
    col_spec,← 2         2    1      1       1          1
```

We want to write a function with this syntax:

```
    (data header)←col_spec ReadOrderData path
```

Reading the header:

```
tn←'order_data.csv' ⎕ntie 0
(_ header)←⎕CSV ⎕OPT'Records' 1⊢tn ⍬ 1 1
⎕nuntie tn
```

```
    ∇ (data cols)←col_spec ReadOrderData path;tn;types
      tn←path ⎕NTIE 0
      cols←⊃⌽(⎕CSV ⎕OPT'Records' 1)tn ⍬ 1 1
      types←(col_spec[;2],1)[col_spec[;1]⍳cols]
      data←⎕CSV tn ⍬ types
      ⎕NUNTIE tn
    ∇
```

Date times

Were provided as `YYYY-MM-DD hh:mm:ss`.

Some extracted year and month as text. Some extracted integer numbers.

Exercise: Convert `YYYY-MM-DD hh:mm:ss` into `¯1↓⎕TS`-style numeric timestamp.

- For a simple timestamp vector, return a simple numeric vector.
- For a nested list of timestamp vectors, return a nested list of numeric vectors.

```
     Timestamp2TS←{
         ⊃⍣(1=≡⍵)⊢2⊃¨': -'∘⎕VFI¨⊆⍵                    ⍝ ⎕VFI

         (⊃,/)⍣(1=≡⍵)↓⎕CSV((,¨'- :')⎕R','⊆⍵)'N' 2    ⍝ ⎕CSV

         r←(⍎¨∊∘⎕D⊆⊢)¨⊆⍵ ⋄ (1+1=≡⍵)⊃r(⊃r)             ⍝ ⍎¨

         ⍎¨'\d+'⎕S'&'⊢⍵                              ⍝ ⎕S
     }
```

This version was suggested as "barbaric". Note that the use of execute `⍎⍵` above is "safer" because we explicitly *include* digits (`∊∘⎕D`) rather than *exclude* non-digits (`~∊∘'- :'`).

```
    {⍎¨'- :'((~∊⍨)⊆⊢)⍵}'2042-05-23 13:24:44'
```

Note also that `⎕VFI` will recognise any valid APL numeric literal:

```
      ⎕VFI'3,14 3.14 3e14 3j14 ¯3 -3'
┌───────────┬─────────────────────┐
│0 1 1 1 1 0│0 3.14 3E14 3J14 ¯3 0│
└───────────┴─────────────────────┘
```

Whereas `⎕CSV` can convert hyphen-number negatives but not APL high minus or complex numbers:

```
      {(⍬∘≡¨0⍴¨⊃¨⍵)(⍵)}(⎕CSV ⎕OPT'Separator' ' ')'3,14 3.14
3E-14 3j14 ¯3 -3' 'S' 4
┌───────────┬──────────────────────────┐
│0 1 1 0 0 1│┌────┬────┬─────┬────┬──┬──┐│
│           ││3,14│3.14│3E¯14│3j14│¯3│¯3││
│           │└────┴────┴─────┴────┴──┴──┘│
└───────────┴──────────────────────────┘
```

What is the purpose of these expressions? What are their edge cases?

```
a) ⊂⍣(1=≡⍵)⊢⍵
b) ↓↑⍵
c) (1+80=⎕DR ⍵)⊃⍵(⊂⍵)
```

These are expressions for "enclose-if-simple" **on vectors**. Enclose argument `⍵` if it is not nested.

a is equivalent to `⊆⍵`

b adds spaces to shorter elements of `⍵`

```
      ↓↑'SP' 'RDJ' 'BR'
┌───┬───┬───┐
│SP │RDJ│BR │
└───┴───┴───┘
```

c will not work for some Unicode characters (`⎕DR 160` and `320`)

```
      {(1+80=⎕DR ⍵)⊃⍵(⊂⍵)} 'İstanbul'
İstanbul
```

The subtle fiddliness of scalar/1-elem vector/vector/1-row matrix is a bit pedantic. Just remember to be consistent and document expected arguments and results.

Aggregating data

The main problem:

- select relevant columns (keys)
- apply aggregate function on keys
- ensure correct ordering and shape of result

Payment per state

Write a function `PaymentPerState` which:

- accepts a nested vector of character vectors of state codes
- returns the total payment in each given state across the whole dataset.

```
      PaymentPerState 'GO' 'TO' 'SC'
319766.98 58068.18 579297.8
```

Exercise: Spot the errors
This code is problematic. What issues can you spot?

```
PPS←{
     sp ← (⊂cols⍳'state' 'payment') (⌷⍤1) data
     sp ⌿⍨← (⊣/sp)∊⍵
     (⊣/sp) {+/⍵}⌸ (⊢/sp)
 }
```

1. Valid state missing in data set
2. Data with keys in order found in data, rather than order of ⍵

Several ways to mitigate each.

Exercise: Fix the issues with `PPS`

```
PPS←{
     sp ← (⊂cols⍳'state' 'payment') (⌷⍤1) data
     sp ⌿⍨← (⊣/sp)∊⍵
     (⊣/sp) {+/⍵}⌸ (⊢/sp)
 }
```

1. Prepend data and keys with fill and dictionary
2. Use a lookup `⍺⍳⍵` after ⌸

```
PPS←{
     sp ← (⊂cols⍳'state' 'payment') (⌷⍤1) data
     sp ⌿⍨← (⊣/sp)∊⍵
     (⍵,⊣/sp) {+/⍵}⌸ (-⍵+⍥≢sp)↑(⊢/sp)
 }
```

```
PPS←{
     sp ← (⊂cols⍳'state' 'payment') (⌷⍤1) data
     sp ⌿⍨← (⊣/sp)∊⍵
     (gs tot) ←↓⍉ (⊣/sp) {⍺,+/⍵}⌸ (⊢/sp)
     tot[gs⍳⍵]
 }
```

Note: Performance of `('state' From cols) ∊ states`

- Filtering may be improved by pre-computing numeric "IDs", but lookup still required
- Lookup may be improved by using inverted tables
- Ultimately a storage / database issue

```
      states←data[;header⍳⊂'state']
      us←∪states
      sid←us⍳states

      ]runtime -c "sid∊us⍳'SP' 'TV' 'KD' 'RJ'" "states∊'SP' 'TV'
'KD' 'RJ'"

  sid∊us⍳'SP' 'TV' 'KD' 'RJ' → 3.7E¯5 |      0%
  states∊'SP' 'TV' 'KD' 'RJ' → 6.3E¯3 | +16815%
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

Grouping by date-time / computing intervals

Problems `PaymentPerMonth` and `PaymentPerQuarter` allowed for different approaches:

- Modular approach / code re-use
- Directly compute quarter intervals

Exercise 1: Given `ppm←PaymentPerMonth states`, how can we compute `ppq←PaymentPerQuarter`?

```
ppm←PaymentPerMonth 'SP' 'RJ'
      {+/¨(3/⍳4)⊆⍵}ppm
264219.36 504658.15 641944.62 932013.08
105832.49 196499.9  274290.78 399480.18
      {+/(≢⍵)4 3⍴⍵}ppm
264219.36 504658.15 641944.62 932013.08
105832.49 196499.9  274290.78 399480.18
```

Exercise 2: Given numeric months `1...12`, return corresponding quarters `1...4`

```
      months←?10⍴12

      ↑(months)(1 4 7 10⍸months)
9 4 1 10 2 3 10 7 3 4
3 2 1  4 1 1  4 3 1 2

      1 4 7 10(+⌿∘.≤)months
3 2 1 4 1 1 4 3 1 2

      ⌈3÷⍨months
3 2 1 4 1 1 4 3 1 2
```

Exercise: Write the function `PaymentsBetween` which:

- `⍵:` takes as argument a nested vector of character vectors `(1=≢⍴⍵)∧(2=≡⍵)` of dates from oldest (1st element) to most recent (last element)
- `←:` returns a vector of length (`¯1+≢⍵`) of total payments between dates specified

```
PaymentsBetween←{
     dt←¯1 1 ⎕DT Timestamp2TS ⍵
     ddn←¯1 1 ⎕DT Timestamp2TS data[;header⍳⊂'timestamp']
⍝ Note: do not need ⎕DT due to TAO
⍝        intervals ← ⍵⍸data[;header⍳⊂'timestamp']
     intervals←dt⍸ddn
     (int tot)←↓⍉intervals{⍺,+/⍵}⌸data[;header⍳⊂'payment']
     (tot,0)[int⍳⍳¯1+≢dt]
 }
```

Day 2

- Grouping by multiple columns
    - Re-implementing `PaymentPerMonth` using alternative method
- Batch processing
    - Re-implementing `PaymentPerMonth` using batch processing

Grouping by multiple columns

`PaymentPerMonth` requires grouping using multiple columns. Two basic approaches to this are:

1. Multiple uses of key (e.g. `F⌸¨{⊂⍵}⌸data` or `{F⌸⍵}⌸data`)
2. Creating compound keys (e.g. catenate together key columns)

Payment per month

Write a function PaymentPerMonth which:

- accepts a state code or nested vector of state codes
- returns a simple numeric vector (shape `12`) or matrix (shape `(≢⍵),12`) of the total payment in each state in each month of 2017 in order left-to-right from January to December.

```
      months←'Mmm'(1200⌶)29×⍳12
      states←'SP' 'RJ' 'PI' 'MT'

      PaymentPerMonth 'SP'
43103.53 80348.6 140767.23 130989.25 188394.13 185274.77
197902.88 212931.9 231109.84 239321.27 391137.77 301554.04

      ppm←PaymentPerMonth states

      ⍉((⊂''),months)⍪states,ppm
             SP         RJ       PI        MT
 Jan   43103.53   13139.53  1453.98   1922.78
 Feb   80348.6    33197.29  3298.4    3583.36
 Mar  140767.23   59495.67  2582.92   2702.55
 Apr  130989.25   61960.3   2288.91   3912.86
 May  188394.13   75293.52  6679.58   7560.36
 Jun  185274.77   59246.08  2626.96   4788.16
 Jul  197902.88   84167.86  2938.77  11235.49
 Aug  212931.9    85555.98  5072.72   6939.29
 Sep  231109.84  104566.94  3242.68   8101.66
 Oct  239321.27  108026.61  4544.47  12828.51
 Nov  391137.77  166838.56  3745.39  13144.66
 Dec  301554.04  124615.01  3482     10432.55
```

Exercise:

Previously you submitted solutions to `PaymentPerMonth`.

If you used **Method 1** in your solution to `PaymentPerMonth`, write a new solution which uses **Method 2** and vice versa.

1. Method 1
   A. `states {months F⌸⍵}⌸ data`
   B. `{months F⌸⍵}¨ states{⊂⍵}⌸data`
2. Method 2
   `(states,⍤months) F⌸ data`

If you do not have a `PaymentPerMonth` function available, the definition below can be used as a starting point.

It returns a single 12-element vector with payments summed across all states in `⍵`. Modify the definition to return a matrix of payment totals, one row per state and one column per month.

```
 PaymentPerMonth←{
      (data header)←col_spec ReadOrderData DATA_FILE
      Get←{data[;header⍳⊆⍵]}
      data⌿⍨←(Get'state')∊⍵
      Timestamp2TS←{2⊃¨': -'∘⎕VFI¨⊆⍵}
      (year month)←{(⊃¨⍵)(2⊃¨⍵)}Timestamp2TS Get'timestamp'
      (data year month)⌿⍨←⊂year∊2017
      dict←⍳12
      (dict,month){+/⍵}⌸((≢dict)⍴0),Get'payment'
  }
```

Example solutions:

```
  PaymentPerMonthM1A←{
      (data header)←col_spec ReadOrderData DATA_FILE
      Get←{data[;header⍳⊆ω]}
      data⌿⍨←(Get'state')∊ω
      Timestamp2TS←{2⊃¨': -'∘⎕VFI¨⊆ω}
      (year month)←{(⊃¨ω)(2⊃¨ω)}Timestamp2TS Get'timestamp'
      (data year month)⌿⍨←⊂year∊2017
      dict←⍳12
      (s m p)←↓⍉(Get'state'){α,↓⍉ω[;1]
{α,+/ω}⌸ω[;2]}⌸month,⍪Get'payment'
      i←⊃,/(ω⍳s),¨¨m
      (∊p)@i⊢(≢ω)12⍴0
      ⍝ r←(≢ω)12⍴0 ⋄ r[i]←∊p
  }

   PaymentPerMonthM1B1←{
      (data header)←##.(col_spec ReadOrderData DATA_FILE)
      Get←{data[;header⍳⊆ω]}
      data⌿⍨←(Get'state')∊ω
      Timestamp2TS←{2⊃¨': -'∘⎕VFI¨⊆ω}
      (year month)←{(⊃¨ω)(2⊃¨ω)}Timestamp2TS Get'timestamp'
      (data year month)⌿⍨←⊂year∊2017
      dict←⍳12
      (s by_state)←↓⍉(Get'state'){α,⊂ω}⌸month,⍪Get'payment'
      (m p)←↓⍉↑↓∘⍉¨(⊣/{α,+/ω}⌸⊢/)¨by_state
      i←⊃,/(ω⍳s),¨¨m
      svm←(≢ω)12⍴0
      svm[i]←∊p
      svm
  }

   PaymentPerMonthM1B2←{
      (data header)←col_spec ReadOrderData DATA_FILE
      Get←{data[;header⍳⊆ω]}
      data⌿⍨←(Get'state')∊ω
      Timestamp2TS←{2⊃¨': -'∘⎕VFI¨⊆ω}
      (year month)←{(⊃¨ω)(2⊃¨ω)}Timestamp2TS Get'timestamp'
      (data year month)⌿⍨←⊂year∊2017
      dict←⍳12
      by_state←(ω,Get'state'){⊂ω}⌸((≢ω)2⍴1
0)⍪month,⍪Get'payment'
      ↑{(dict,ω[;1]){+/ω}⌸((≢dict)⍴0),ω[;2]}¨by_state
  }

   PaymentPerMonthM2←{
      (data header)←##.(col_spec ReadOrderData DATA_FILE)
      Get←{data[;header⍳⊆ω]}
      data⌿⍨←(Get'state')∊ω
      Timestamp2TS←{2⊃¨': -'∘⎕VFI¨⊆ω}
      (year month)←{(⊃¨ω)(2⊃¨ω)}Timestamp2TS Get'timestamp'
      (data year month)⌿⍨←⊂year∊2017
      dict←,ω∘.,⍳12
      pay←(dict,(Get'state'),¨month)
```

```
    {+/⍵}⌸((≢dict)⍴0),Get'payment'
         (≢⍵)12⍴pay
     }
```

Batch processing

For very large data, we may not be able to import the entire set into the workspace (`WSFULL` error).

`⎕CSV` lets us process the data in batches - some number of rows at a time.

Exercise

Define the function `PaymentPerMonthBatch` based on the following template:

```
 svm←PaymentPerMonthBatch states
 ⍝svm←payments: ↓states vs →months
  ;LoadBatch;LoadHeader;batch;col_spec;col_types;header;tn;Timestamp2YM

  LoadHeader←{⊃⌽(⎕CSV ⎕OPT'Records' 1)⍵ ⍬ 1 1}
  LoadBatch←{(⎕CSV ⎕OPT'Records' 1000)⍵ ⍬ ⍺}
  Timestamp2YM←{↓⍉↑2↑¨2⊃¨'- :'∘⎕VFI¨⍵}

  col_spec←⍪'payment' 'id' 'city' 'state' 'category' 'timestamp'
  col_spec,←2 2 1 1 1 1

  tn←'C:\g\2023-KeyWorkshop\order_data.csv'⎕NTIE 0

  header←LoadHeader tn
  col_types←(col_spec[;2],1)[col_spec[;1]⍳header]
  ⍝⍝⍝ Your code here ↓↓↓

  :While 0<≢batch←col_types LoadBatch tn

  :EndWhile

  ⍝⍝⍝ Your code here ↑↑↑
  ⎕NUNTIE tn
```

The function returns a simple numeric matrix (shape `(≢⍵),12`) of the total payment in each state in each month of 2017 in order left-to-right from January to December.

1. Decide whether you are going to use key multiple times, or once with compound keys.
2. Decide how to have the correct result order - prepend with a dictionary or using a lookup?
3. Complete the `PaymentPerMonthBatch` function.

Example solution:

```
svm←PaymentPerMonthBatch states;Get;dict;month;payment;state;year
⍝svm←payments: ↓states vs →months
 ;LoadBatch;LoadHeader;batch;col_spec;col_types;header;tn;Timestamp2YM

 LoadHeader←{⊃⌽(⎕CSV ⎕OPT'Records' 1)⍵ ⍬ 1 1}
 LoadBatch←{(⎕CSV ⎕OPT'Records' 1000)⍵ ⍬ ⍺}
 Timestamp2YM←{↓⍉↑2↑¨2⊃¨'- :'∘⎕VFI¨⍵}

 col_spec←⍪'payment' 'id' 'city' 'state' 'category' 'timestamp'
 col_spec,←2 2 1 1 1 1

 tn←'C:\g\2023-KeyWorkshop\order_data.csv'⎕NTIE 0

 header←LoadHeader tn
 col_types←(col_spec[;2],1)[col_spec[;1]⍳header]
 ⍝⍝⍝ Your code here ↓↓↓
 Get←{batch[;header⍳⊆⍵]}
 svm←(12×≢states)⍴0
 dict←,states∘.,⍳12
 :While 0<≢batch←col_types LoadBatch tn
     (year month)←Timestamp2YM Get'timestamp'
     payment←Get'payment' ⋄ state←Get'state'
     (payment month state)⌿⍨←⊂(state∊states)∧year=2017
     svm+←(dict,state,¨month){+/⍵}⌸((≢dict)⍴0),payment
 :EndWhile
 svm⍴⍨←(≢states)12
 ⍝⍝⍝ Your code here ↑↑↑
 ⎕NUNTIE tn
```

Proposal for extension to key ⌸

As you will have noticed, using the key operator often requires extra consideration to account for the desired order of results and possible missing values.

In the monadic case, we can use a dictionary left operand, ensuring to filter out non-members in our data (`⍵∩⍺⍺`):

```
      ⎕A{1↓¨{⊂⍵}⌸⍺⍺,⍵∩⍺⍺}'ABRACADABRA!!!'
┌──────────────┬─────┬──┬──┬┬┬┬┬┬┬┬┬┬┬┬┬┬─────┬┬┬┬┬┬┬┬┐
│27 30 32 34 37│28 35│31│33│││││││││││││││29 36│││││││││
└──────────────┴─────┴──┴──┴┴┴┴┴┴┴┴┴┴┴┴┴┴─────┴┴┴┴┴┴┴┴┘
      ≢¨⎕A{1↓¨{⊂⍵}⌸⍺⍺,⍵∩⍺⍺}'ABRACADABRA!!!'
5 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0
```

In the dyadic case, we can create a mask to filter our keys and values:

```
      keys←'11222114443333'
      dict←'13'
      _K←{mask←⍺∊⍺⍺ ⋄ 1↓¨(⍺⍺,mask/⍺){⊂⍵}⌸⍺⍺,mask/⍵}
      _K←{1↓¨⍺{⊂⍵}⌸⍥(⍺⍺,(⍺∊⍺⍺)∘/)⍵}   ⍝ compact version
      vals←'ABRACADABRA!!!'
      keys(dict _K)vals
┌────┬────┐
│ABAD│A!!!│
└────┴────┘
```

A more general version is available from github.com/abrudz/dyalog_vision/blob/main/QuadEqual.aplo.

BONUS Inverted tables

Nested arrays conveniently represent tables (e.g. database, spreadsheet). However, there are more efficient ways to store and manage data in APL.

"**Inverted tables**" is APL-speak for **columnar database** or **column-store database**.

Flat, homogeneous arrays are stored sequentially in memory. Their shape information at the front (counting elements is fast) and the ravel of elements afterwards.

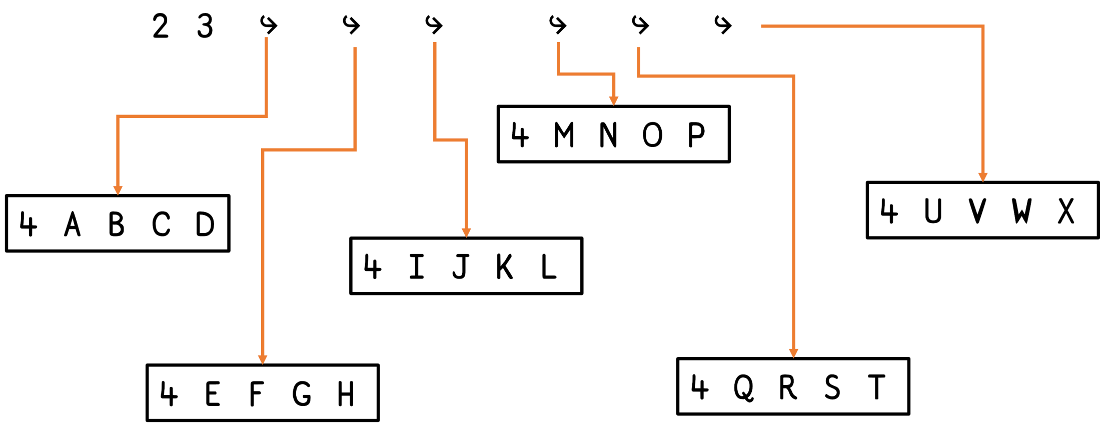
```
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

2 3 4   A B C D E F G H I J K L M N O...
```

Nested arrays are pointer arrays internally, so it can take longer for the interpreter to traverse memory looking for the data.

```
┌────┬────┬────┐
│ABCD│EFGH│IJKL│
├────┼────┼────┤
│MNOP│QRST│UVWX│
└────┴────┴────┘
```



APL arrays are stored in memory in **row-major** order (also known as **ravel order**). Inverted tables instead store each column as a contiguous array, making lookups and selection within individual columns faster.

There is an I-beam (`8⌶`) which implements **index-of** (`⍺⍳⍵`) efficiently for inverted tables.

- Documentation for inverted-table index-of: https://help.dyalog.com/latest/#Language/I%20Beam%20Functions/Inverted%20Table%20Index%20Of.htm
- Inverted Tables // Roger Hui // Dyalog '18: https://www.youtube.com/watch?v=IOWDkqKbMwk&t=20m28s

This is how to read a **.csv** file as an inverted table using `⎕CSV` :

```
In [ ]: path←'C:\g\2023-KeyWorkshop\order_data.csv'
        (data header)←(⎕CSV⎕OPT'Invert'1)path ⍬ (2 1 1 1 2 1) 1
        ⎕←header
        ⎕←3↑¨data
```

```
Out[ ]: ┌──┬─────────┬────┬─────┬───────┬────────┐
        │id│timestamp│city│state│payment│category│
        └──┴─────────┴────┴─────┴───────┴────────┘

Out[ ]: ┌─────┬───────────────────┬────────────────────────────────┬──┬───────
       ─────────┬───────────────────────────────────────┐
        │1 2 3│2017-10-02 10:56:33│sao paulo                       │SP│18.12 14
       1.46 179.12│housewares                             │
        │     │2018-07-24 20:41:37│barreiras                       │BA│
        │perfumery                              │
        │     │2018-08-08 08:38:49│vianopolis                      │GO│
        │auto                                   │
        └─────┴───────────────────┴────────────────────────────────┴──┴───────
       ─────────┴───────────────────────────────────────┘
```

Exercise:

Modify `PaymentPerMonthBatch` to use inverted tables instead of nested matrices.

Note: Inverted-table versions of some primitives can be found on APLCart.