```
L24 -- Efficient Page Rank
[Jeff Phillips - Utah - Data Mining]

-------------
MapReduce

Big data D = {D1, D2, ... Dm}
 too big for one machine
 each Di on machine i

 [ Each machine has limited memory! ... compared to data ]

proceeds in rounds (3 parts):
  1:  Mapper
      all d in D  -> (k(d), v(d))
  2:  Shuffle
      moves all (k, v) and (k', v') with k=k' to same machine
  3:  Reducer
      {(k,v1), (k,v2), ...,} -> output usually f(v1,v2,...)

1.5:  Combiner
      if one machine has multiple (k,v1) , (k,v2)
      then performs part of Reduce before Shuffle.

Can think of output of Reducer as Di on machine i.
Then can string multiple MR-rounds together.

*** key-value pairs can encode much deeper computing power
  + Mapper f(Di) -> {(ki,vi)}_j  -> with (ki = i, v_i = input to node i)
*** Provides very rubout system, many fail-safes if node goes down, gets
slow...
*** very simple!

-------- EXAMPLE ----------
Histogram into k bins
 Mapper d in D  -> (k=bin(d), 1)
   (combiner)
 Reducer  (k=i,v) -->  output = sum v
---------------------------

------------------------------------------------------------------
Page Rank:

Internet stored as big matrix M  (size=nxn)
  + sparse, 99%+ of entries are 0
    ([M[a,b] = 0] == no link from page a to page b)
```

```
  + P = beta M + (1-beta) B    where B[a,b] = 1/n
        beta =~ 0.85

page-rank vector: q_* = P^t q  as   t-> infty   (here t = 50 to 75 ok)
   "importance of webpage"  (other details too, but this is computational hard
part)

Problems:
  - M is sparse, but B (implicit) and P^n is dense!  Too BIG to store
    -->  q_i is O(n) can always store, so just compute
              q_{i+1} = beta * M *q_i + (1-beta) e/n
             t times

  - Still very big computation.  Gigabytes.
     Many machines and machine crash!
     -->  MapReduce!

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



simple:  assume q fits in one machine (twice: e.g. q_i and q_{i+1})

  -->  break M into vertical stripes
        M = [M1 M2 ... Mk]
       (and q into q = [q1; q2; ...; qk] = horizontal split)
    then
   Mapper i -> (key=i' in [k] ; val = (row=r of Mi * qi) )
   Reducer:  adds values to get each element q[i'] * beta + (1-beta)/n

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

big q:  what if q does not fit in a single machine?

option 1:  Tiling.

 M into sqrt(k) x sqrt(k) blocks
   M = [M11 M12 .. M1sqrt{k};
        M21 M22 .. M2sqrt{k};
        ...;
        Msqrt{k}1  Msqrt{k}2 .. Msqrt{k}sqrt{k}]

  Mapper:
  k machines each get one block M_{i,j}
    and get sent q_i  for i in [sqrt{k}]

  Reducer:
   on each row i', adds M_{i,j} q_i -> q[i']
```

```
     and does q_+[i'] = q[i'] * beta + (1-beta)/n

 Problems:
   - each q_i (for i in [sqrt{k}]) is sent sqrt{k} places
   - thrashing: on M_{i,j}
        -->  solution: striping -> prefetching
                on q_+  (each column M_{i,j} may add to q_+[i'])
        -->  solution:  blocking on M_{i,j}  (sqrt{k} x sqrt{k} blocks)
                     read M_{i,j} once || read,write q/q_+ sqrt{k} times



--------------------------------------
Example:

M = [0   1/2 0  0  ]
    [1/3 0   1  1/2]
    [1/3 0   0  1/2]
    [1/3 1/2 0  0]

stripe:
 M1 = [0; 1/3; 1/3; 1/3]
    stored as (1:  (1/3,2) (1/3,3) (1/3,4))
 M2 = [1/2; 0; 0; 1/2]
    stored as (2:  (1/2,1) (1/2,4))
 M3 = [0; 1; 0; 0]
    stored as (3:  (1,3))
 M4 = [1/3; 1/2; 0 0]
    stored as (4:  (1/3,1) (1/2,2))

block:
 M11 = [0 1/2; 1/3 0]
    stored as (1: (1/2,2)) (2: (1/3,1))
 M12 = [0 0; 1 1/2]
    stored as (4: (1,1) (1/2,2))
 M21 = [1/3 0; 1/3 1/2]
    stored as (1: (1/3,3)) (2: (1/3,3) (1/2,4))
 M22 = [0 1/2; 0 0]
    stored as (3: (1/2,4))

Note that some blocks have no effect on some vector elements they are
responsible for
  -->  M22 has no effect on q_+[3].
  -->  M12 has no use for q[3].
    This is quite common, and can be used to speed up.
```