More About PageRank

Hubs and Authorities (HITS)
Combatting Web Spam
Dealing with Non-Main-Memory Web Graphs

Jeffrey D. Ullman
Stanford University

HITS

Hubs

Authorities

Solving the Implied Recursion

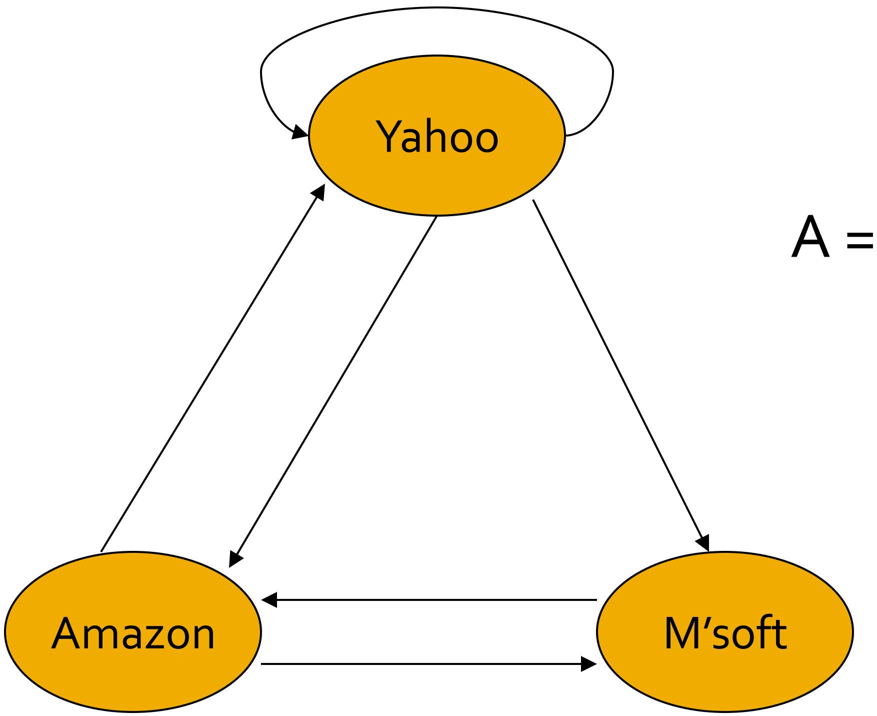
Hubs and Authorities ("HITS")

- Mutually recursive definition:
  - A *hub* links to many authorities;
  - An *authority* is linked to by many hubs.
- Authorities turn out to be places where information can be found.
  - Example: course home pages.
- Hubs tell where the authorities are.
  - Example: departmental course-listing page.

Transition Matrix *A*

- HITS uses a matrix $A[i, j] = 1$ if page i links to page j, 0 if not.
- A^T, the transpose of A, is similar to the PageRank matrix M, but A^T has 1's where M has fractions.
- Also, HITS uses column vectors **h** and **a** representing the degrees to which each page is a hub or authority, respectively.
- Computation of **h** and **a** is similar to the iterative way we compute PageRank.

Example: H&A Transition Matrix



$$A = \begin{array}{c|ccc|} & y & a & m \\ \hline y & 1 & 1 & 1 \\ a & 1 & 0 & 1 \\ m & 0 & 1 & 0 \\ \hline \end{array}$$

Using Matrix *A* for HITS

- Powers of A and A^T have elements whose values grow exponentially with the exponent, so we need scale factors λ and μ.
- Let **h** and **a** be column vectors measuring the "hubbiness" and authority of each page.
- Equations: $\mathbf{h} = \lambda A\mathbf{a}$; $\mathbf{a} = \mu A^T\mathbf{h}$.
  - Hubbiness = scaled sum of authorities of successor pages (out-links).
  - Authority = scaled sum of hubbiness of predecessor pages (in-links).

Consequences of Basic Equations

- From **h** = λA**a**; **a** = μA^T **h** we can derive:
  - **h** = $\lambda\mu AA^T$ **h**
  - **a** = $\lambda\mu A^TA$ **a**
- Compute **h** and **a** by iteration, assuming initially each page has one unit of hubbiness and one unit of authority.
  - Pick an appropriate value of $\lambda\mu$.

Scale Doesn't Matter

- Remember: it is only the direction of the vectors, or the relative hubbiness and authority of Web pages that matters.
- As for PageRank, the only reason to worry about scale is so you don't get overflows or underflows in the values as you iterate.

Example: Iterating H&A

$$\mathbf{a} = \lambda\mu A^T A\ \mathbf{a};\ \mathbf{h} = \lambda\mu A A^T\ \mathbf{h}$$

$$A = \boxed{\begin{matrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}} \quad A^T = \boxed{\begin{matrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{matrix}} \quad AA^T = \boxed{\begin{matrix} 3 & 2 & 1 \\ 2 & 2 & 0 \\ 1 & 0 & 1 \end{matrix}} \quad A^TA = \boxed{\begin{matrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{matrix}}$$

a(yahoo)	=	1	5	24	114	. . .	$1+\sqrt{3}$
a(amazon)	=	1	4	18	84	. . .	2
a(m'soft)	=	1	5	24	114	. . .	$1+\sqrt{3}$

h(yahoo)	=	1	6	28	132	. . .	1.000
h(amazon)	=	1	4	20	96	. . .	0.735
h(microsoft)	=	1	2	8	36	. . .	0.268

Solving HITS in Practice

- Iterate as for PageRank; don't try to solve equations.
- But keep components within bounds.
  - Example: scale to keep the largest component of the vector at 1.
  - Consequence is that λ and μ actually vary as time goes on.

Solving HITS – (2)

- Correct approach: start with **h** = [1,1,…,1]; multiply by A^T to get first **a**; scale, then multiply by A to get next **h**, and repeat until approximate convergence.
- You may be tempted to compute AA^T and A^TA first, then iterate multiplication by these matrices, as for PageRank.
- Bad, because these matrices are not nearly as sparse as A and A^T.

Web Spam

Term Spamming
Link Spamming

What Is Web Spam?

- *Spamming* = any deliberate action solely in order to boost a Web page's position in search-engine results.
- *Spam* = Web pages that are the result of spamming.
- SEO industry might disagree!
  - *SEO* = search engine optimization

Web Spam Taxonomy

- *Boosting* techniques.
  - Techniques for achieving high relevance/importance for a Web page.
- *Hiding* techniques.
  - Techniques to hide the use of boosting from humans and Web crawlers.

Boosting

- *Term spamming*.
  - Manipulating the text of web pages in order to appear relevant to queries.
- *Link spamming*.
  - Creating link structures that boost PageRank.

Term-Spamming Techniques

- *Repetition* of terms, e.g., "Viagra," in order to subvert TF.IDF-based rankings.
- *Dumping* = adding large numbers of words to your page.
  - Example: run the search query you would like your page to match, and add copies of the top 10 pages.
  - Example: add a dictionary, so you match every search query.
  - Key hiding technique: words are hidden by giving them the same color as the background.

Link Spam

Design of a Spam Farm
TrustRank
Spam Mass

Link Spam

- PageRank prevents spammers from using term spam to fool a search engine.
  - While spammers can still use the techniques, they cannot get a high-enough PageRank to be in the top 10.
- Spammers now attempt to fool PageRank by *link spam* by creating structures on the Web, called *spam farms*, that increase the PageRank of undeserving pages.

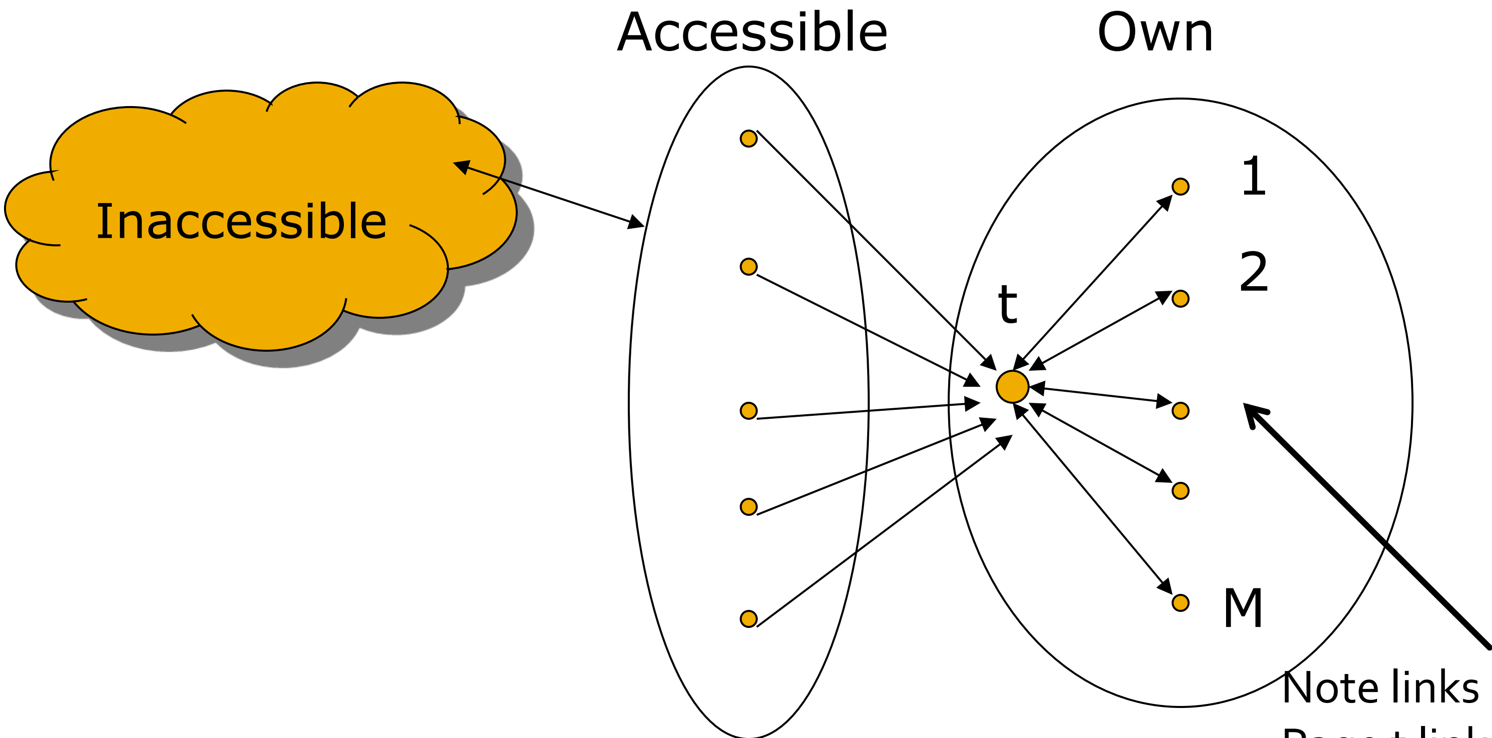
Building a Spam Farm

- Three kinds of Web pages from a spammer's point of view:
  1. *Own pages*.
     - Completely controlled by spammer.
  2. *Accessible pages*.
     - E.g., Web-log comment pages: spammer can post links to his pages.
  3. *Inaccessible pages*.
     - Everything else.

Spam Farms – (2)

- Spammer's goal:
  - Maximize the PageRank of target page *t*.
- Technique:
  1. Get as many links as possible from accessible pages to target page *t*.
  2. Construct a spam farm to get a PageRank-multiplier effect.

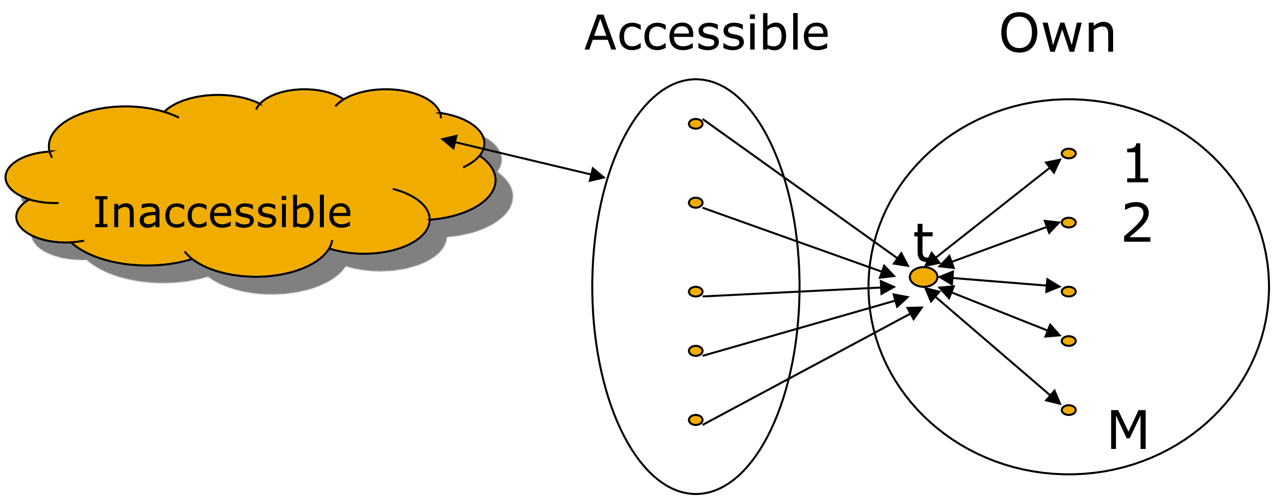
Spam Farms – (3)



Note links are 2-way. Page t links to all M pages and they link back.

Goal: boost PageRank of page *t*.
One of the most common and effective organizations for a spam farm.

Analysis



Suppose rank from accessible pages = x.

PageRank of target page = y.

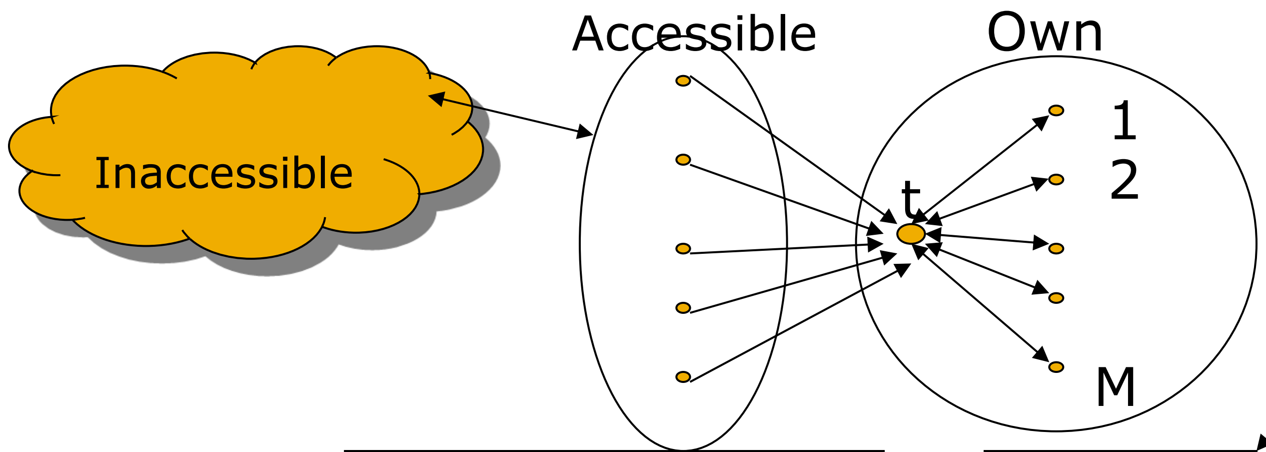
Taxation rate = $1-\beta$.

Rank of each "farm" page = $\boxed{\beta y/M} + \boxed{(1-\beta)/N}$.

Share of "tax";
N = size of the Web.
Total PageRank = 1.

From t; M = number
of farm pages

Analysis – (2)



Tax share for *t*.
Very small; ignore.

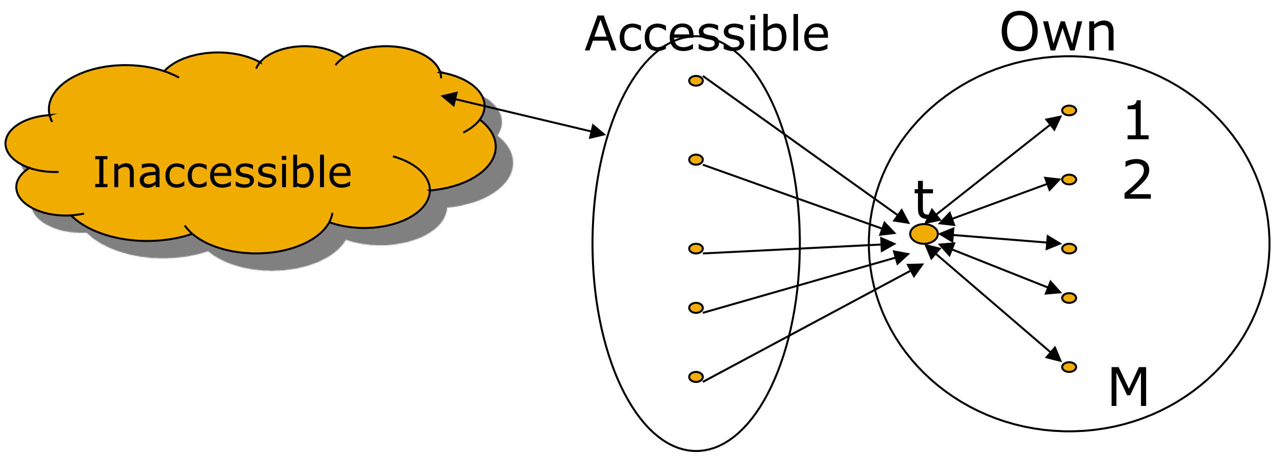
$$y = x + \beta M[\boxed{\beta y/M + (1-\beta)/N}] + \boxed{(1-\beta)/N}$$

$$y = x + \beta^2 y + \beta(1-\beta)M/N$$

$$y = x/(1-\beta^2) + cM/N \text{ where } c = \beta/(1+\beta)$$

PageRank of
each "farm" page

Analysis – (3)



- $y = x/(1-\beta^2) + cM/N$ where $c = \beta/(1+\beta)$.
- For $\beta = 0.85$, $1/(1-\beta^2)= 3.6$.
  - Multiplier effect for "acquired" page rank.
- By making M large, we can make *y* almost as large as we want.

War Between Spammers and Search Engines

- If you design your spam farm just as was described, Google will notice it and drop it from the Web.
- More complex designs might be undetected, but SEO innovations can be tracked by Google et al.
- Fortunately, there are other techniques that do not rely on direct detection of spam farms.

Detecting Link Spam

- Topic-specific PageRank, with a set of “trusted” pages as the teleport set is called *TrustRank*.
- *Spam Mass* = (PageRank – TrustRank)/PageRank.
  - High spam mass means most of your PageRank comes from untrusted sources – you may be link-spam.

Picking the Trusted Set

- Two conflicting considerations:
  - Human may have to inspect each trusted page, so this set should be as small as possible.
  - Must ensure every “good page” gets adequate TrustRank, so all good pages should be reachable from the trusted set by short paths.
    - Implies that the trusted set must be geographically diverse, hence large.

Approaches to Picking the Trusted Set

1. Pick the top *k* pages by PageRank.
   - It is almost impossible to get a spam page to the very top of the PageRank order.
2. Pick the home pages of universities.
   - Domains like .edu are controlled.

- Notice that both these approaches avoid the requirement for human intervention.

Efficiency Considerations for PageRank

Multiplication of Huge Vector and Matrix

Representing Blocks of a Stochastic Matrix

The Problem

- Google computes the PageRank of a trillion pages (at least!).
- The PageRank vector of double-precision reals requires 8 terabytes.
  - And another 8 terabytes for the next estimate of PageRank.

The Problem – (2)

- The matrix of the Web has two special properties:
  1. It is very sparse: the average Web page has about 10 out-links.
  2. Each column has a single value – 1 divided by the number of out-links – that appears wherever that column is not 0.

The Problem – (3)

- Trick: for each column, store n = the number of out-links and a list of the rows with nonzero values (1/n).
- Thus, the matrix of the Web requires at least $(\boxed{4*1}+\boxed{8*10})*10^{12}$ = 84 terabytes.

Integer n

Average 10 links/column,
8 bytes per row number.

The Solution: Striping

- Divide the current and next PageRank vectors into k *stripes* of equal size.
  - Each stripe is the components in some consecutive rows.
- Divide the matrix into squares whose sides are the same length as one of the stripes.
- Pick k large enough that we can fit a stripe of each vector and a block of the matrix in main memory at the same time.
  - Note: the multiplication may actually be done at many machines in parallel.

Example: k = 3

$$\begin{bmatrix} w1 \\ w2 \\ w3 \end{bmatrix} = \begin{bmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{bmatrix} \begin{bmatrix} v1 \\ v2 \\ v3 \end{bmatrix}$$

At one time, we need wi, vj, and Mij in memory.

Vary v slowest: w1 = M11 v1; w2 = M21 v1; w3 = M31 v1; w1 += M12 v2; w2 += M22 v2; w3 += M32 v2; w1 += M13 v3; w2 += M23 v3; w3 += M33 v3

Representing a Matrix Block

- Each column of a block is represented by:
  1. The number n of nonzero elements in the entire column of the matrix (i.e., the total number of out-links for the corresponding Web page).
  2. The list of rows of that block only that have nonzero values (which must be 1/n).
- I.e., for each column, we store n with each of the k blocks and the out-link with whatever block has the row to which the link goes.

Representing a Block – (2)

- Total space to represent the matrix = $(\boxed{4*k}+\boxed{8*10})*10^{12}$ = 4k+80 terabytes.

Integer n for a column is represented in each of k blocks.

Average 10 links/column, 8 bytes per row number, spread over k blocks.

Needed Modifications

- We are not just multiplying a matrix and a vector.
- We need to multiply the result by a constant to reflect the "taxation."
- We need to add a constant to each component of the result **w**.
- Neither of these changes are hard to do.
  - After computing each component w_i of **w**, multiply by β and then add $(1-\beta)/N$.

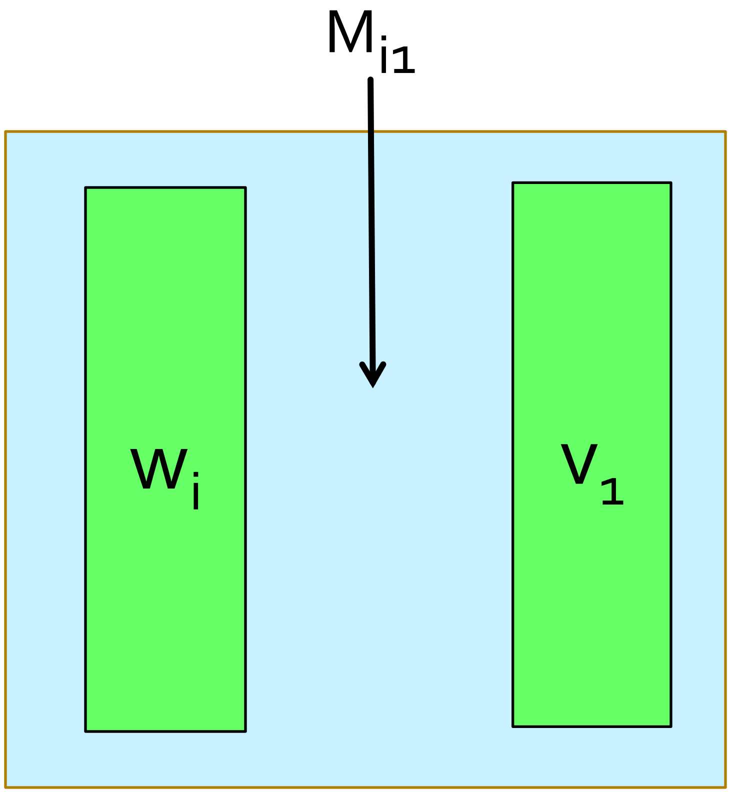
Parallelization

- The strategy described can be executed on a single machine.
- But who would want to?
- There is a simple MapReduce algorithm to perform matrix-vector multiplication.
  - But since the matrix is sparse, better to treat it as a relational join.

Parallelization – (2)

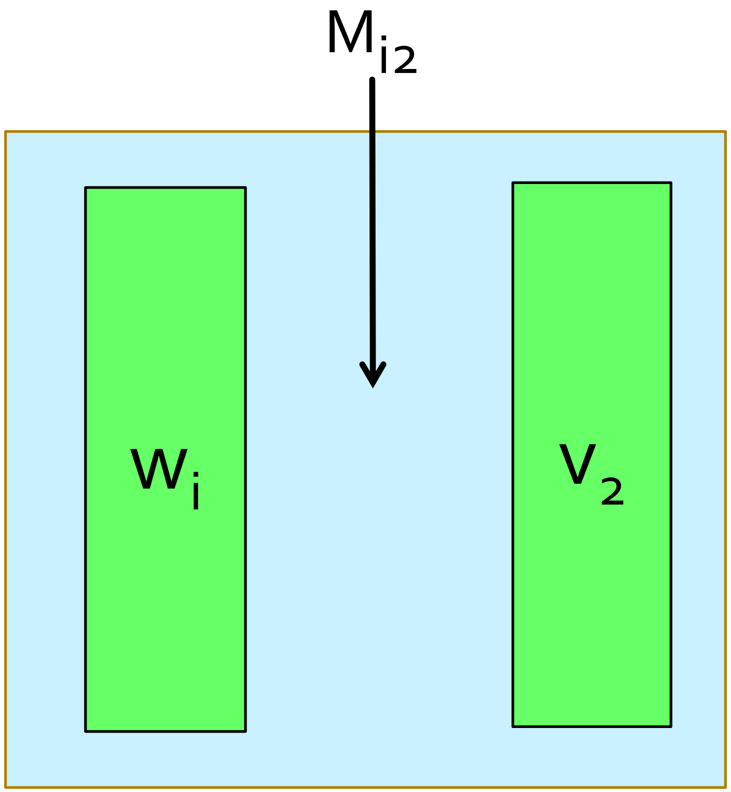
- Another approach is to use many jobs, each to multiply a row of matrix blocks by the entire **v**.
- Use main memory to hold the one stripe of **w** that will be produced.
- Read one stripe of **v** into main memory at a time.
- Read the block of M that needs to multiply the current stripe of **v**, a tiny bit at a time.
- Works as long as k is large enough that stripes fit in memory.
- M read once; **v** read k times, among all the jobs.
  - OK, because M is much larger than **v**.

Animation



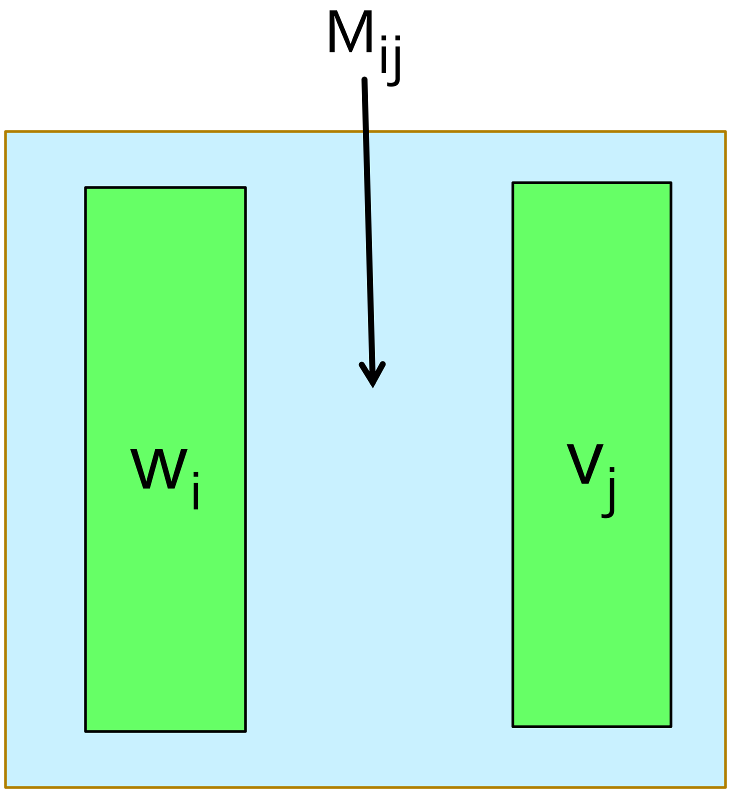
Main Memory for job i

Animation



Main Memory for job i

Animation . . .



Main Memory for job i