UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

CS411 - Query Execution (Part 2)

illinois.edu

Announcements

- HW3 is out, due next Friday
  - Covers indexing
- Piazza Poll - which advanced topics?
  - Information Retrieval
  - Data Mining
  - Distributed Databases
  - Information Integration
  - Semistructured Data

Review

- Why are we learning algorithms for implementing RA operators?
- Why are we learning multiple ways to implement them?
- What is a “full relational unary operator”?

Review

- Why are we interested in the “iterator model” for our operators?
- What is the “buffer pool”?
- How do we measure the cost of operators?
- What is a “table scan”?

Review

- How did we implement the “tuple at a time” operators?
- What is a “nested loop join”?
- What is a “two phase multiway merge sort”?

One-Pass/NBLJ Summary

Operator	M required	I/O Cost
σ, π	1	B
δ, γ	B	B
$\cup, \cap, -, \bowtie, \times$	min(B(R),B(S))	B(R)+B(S)
$\bowtie$	$M \geq 2$	B(R)B(S)/M

illinois.edu

Sort-Based Algorithms

- Can base implementation of operations on merge sort
  - Special implementation for large data
  - Called *Two-Phase Multiway Merge-Sort*
  - TPMMS

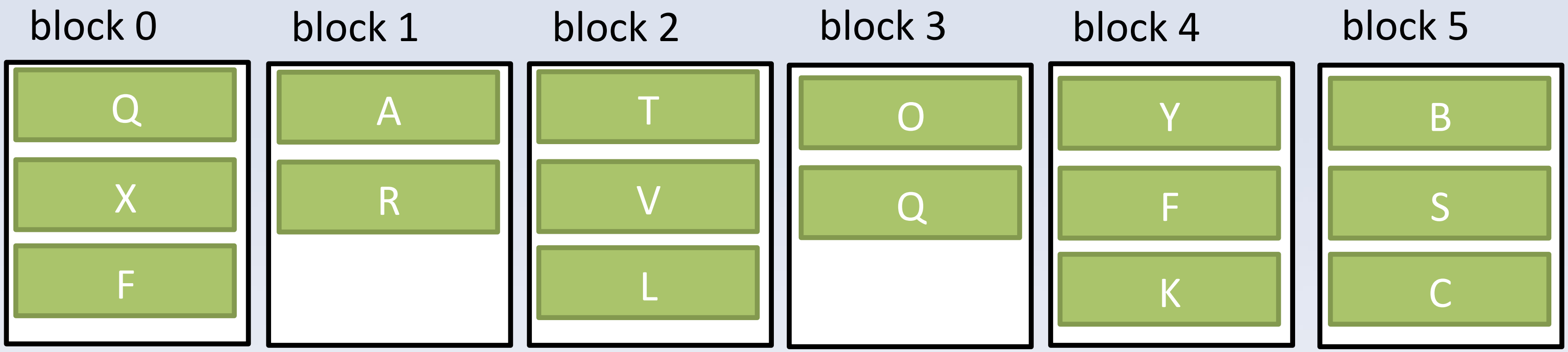
TPMMS

- Has two phases
  - Phase 1: sort memory sized sublists
  - Phase 2: merge sorted sublists
- Requires $\sqrt{B(R)} \leq M$

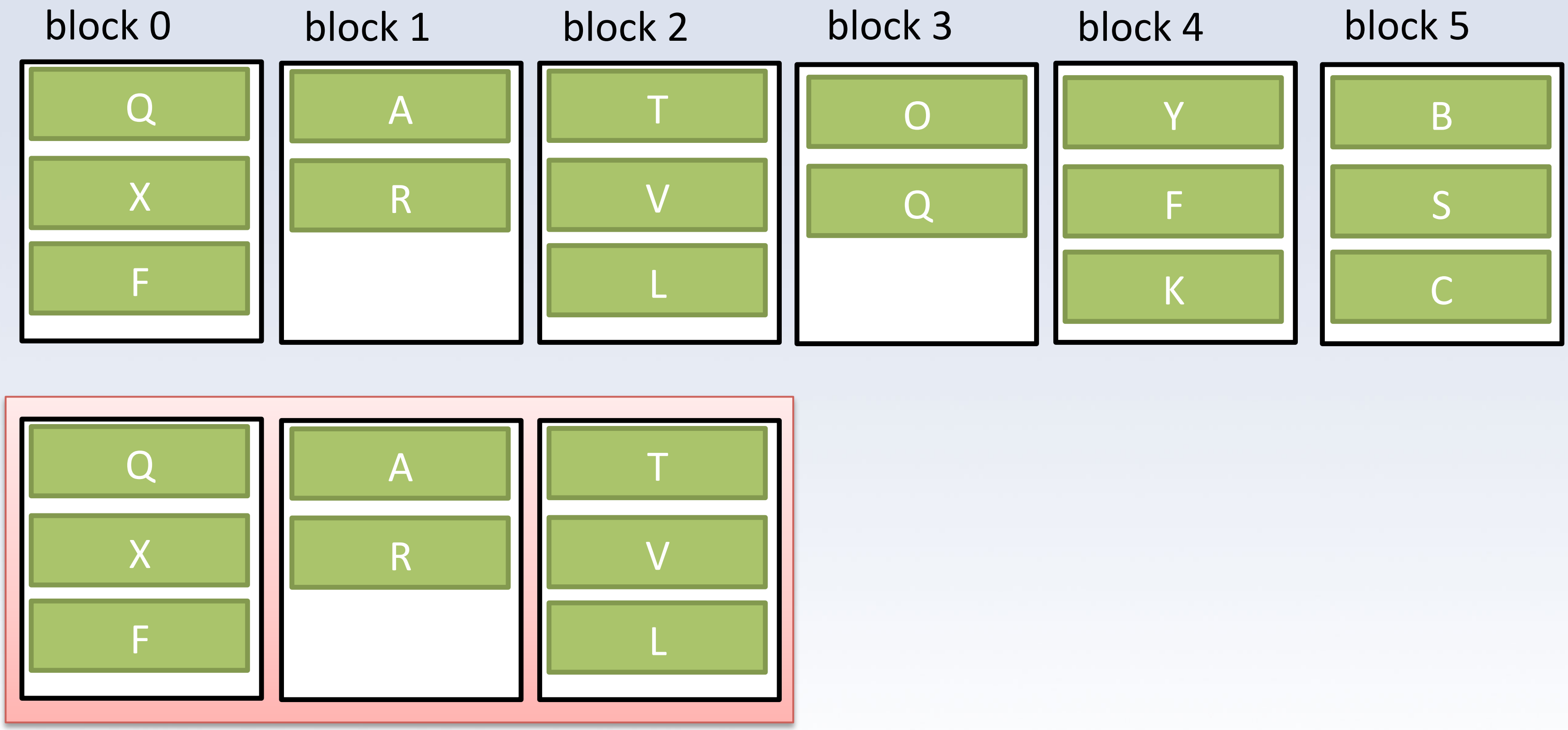
TPMMS

- Phase 1: Repeat until R is exhausted:
    - Fill up *all* M buffers with blocks from R
    - Sort using quicksort
    - Write resulting sublist to disk
- Phase 2: Repeat until sublists exhausted:
    - Load buffers with smallest block for each sublist
    - Identify smallest elements, move to output block
    - When output block full, write to disk
    - If list buffer is empty, get next one from disk

Example

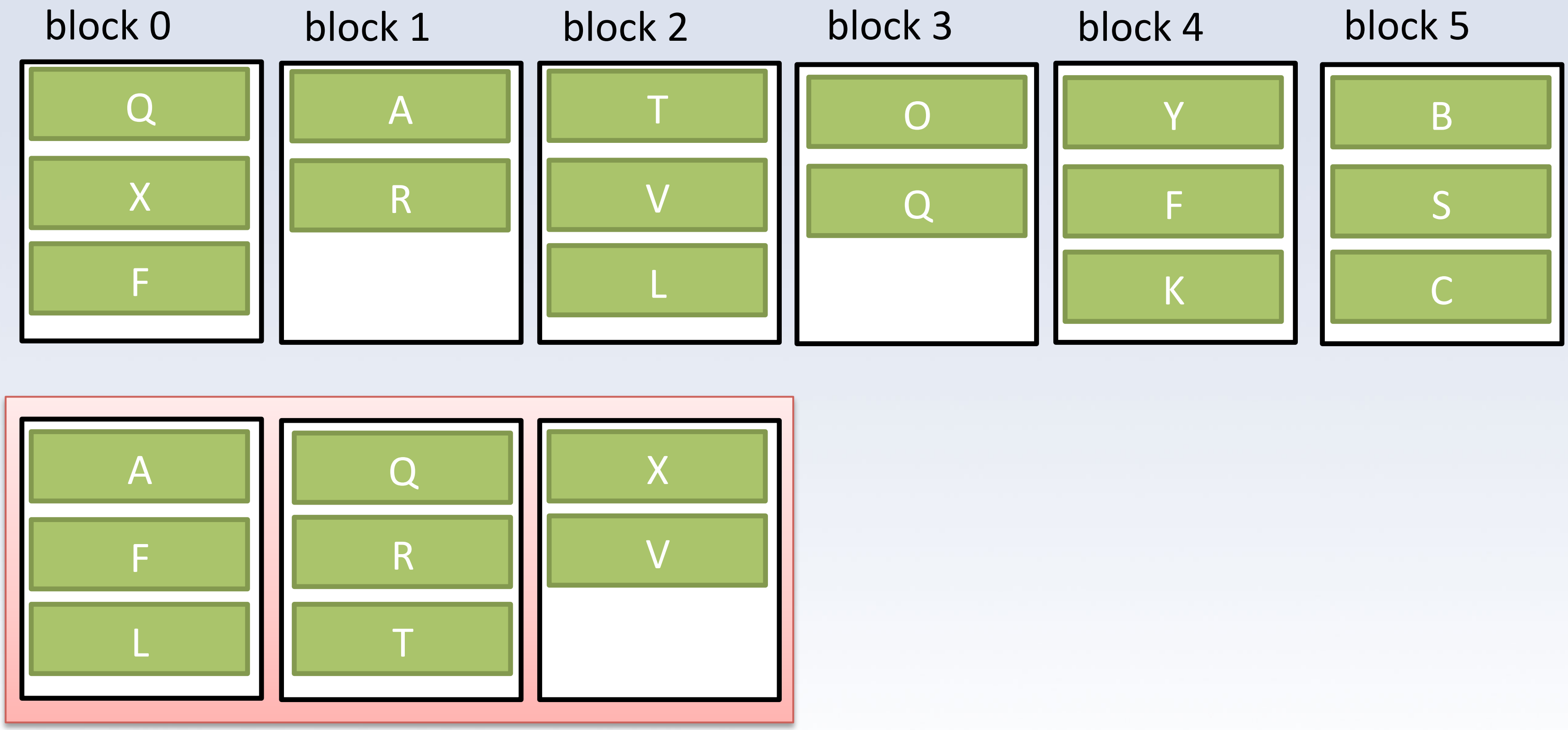

block 0
Q
X
F
block 1
A
R
block 2
T
V
L
block 3
O
Q
block 4
Y
F
K
block 5
B
S
C

Phase 1

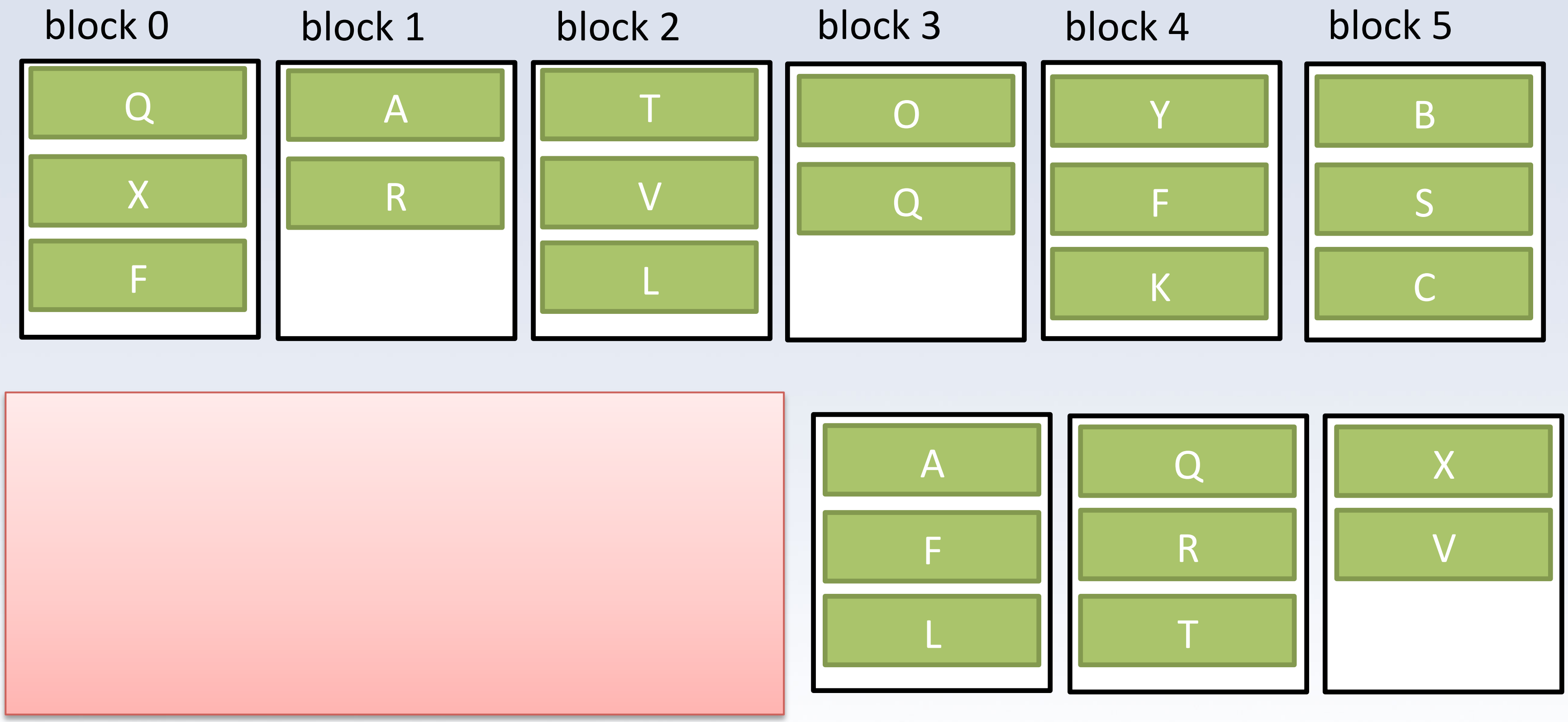


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Q
X
F
A
R
T
V
L

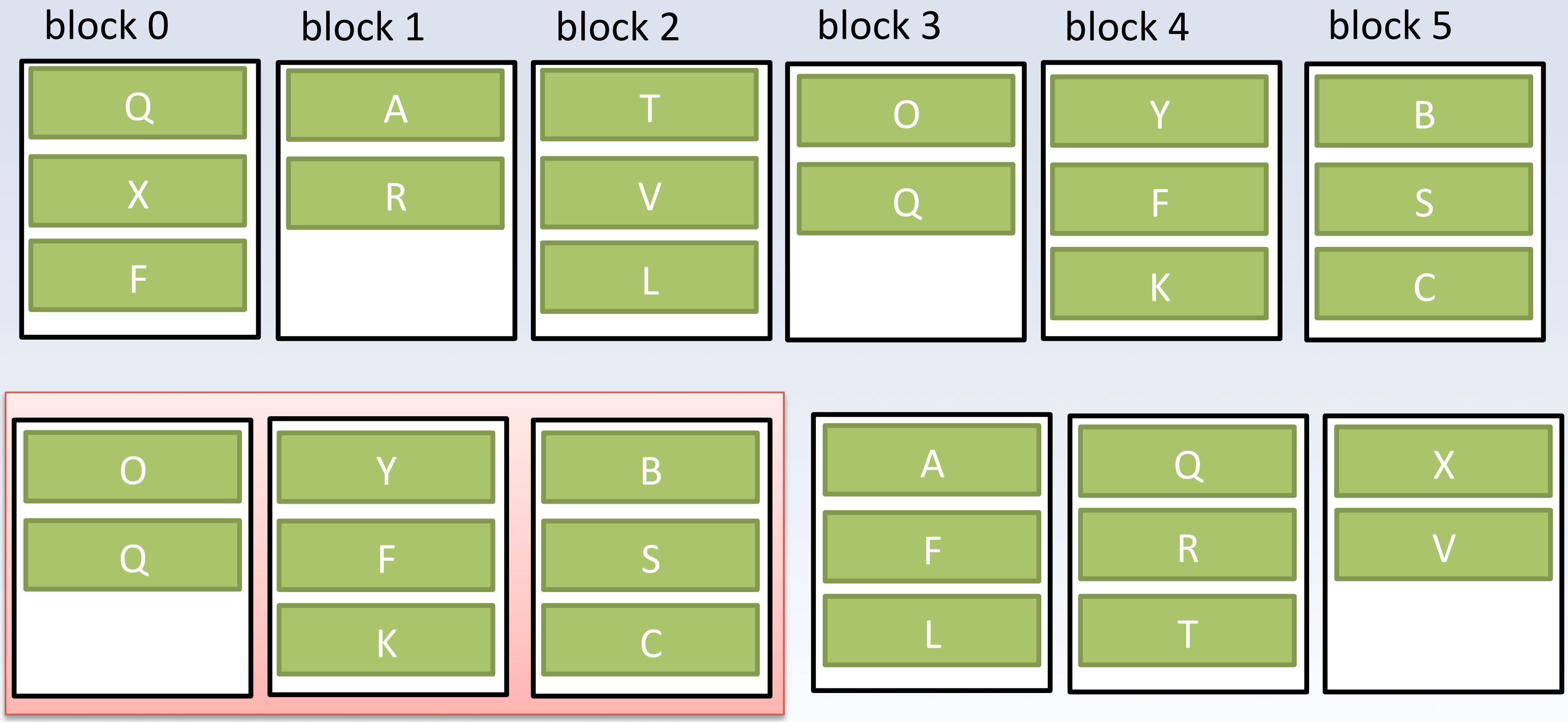
Phase 1


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
A
F
L
Q
R
T
X
V

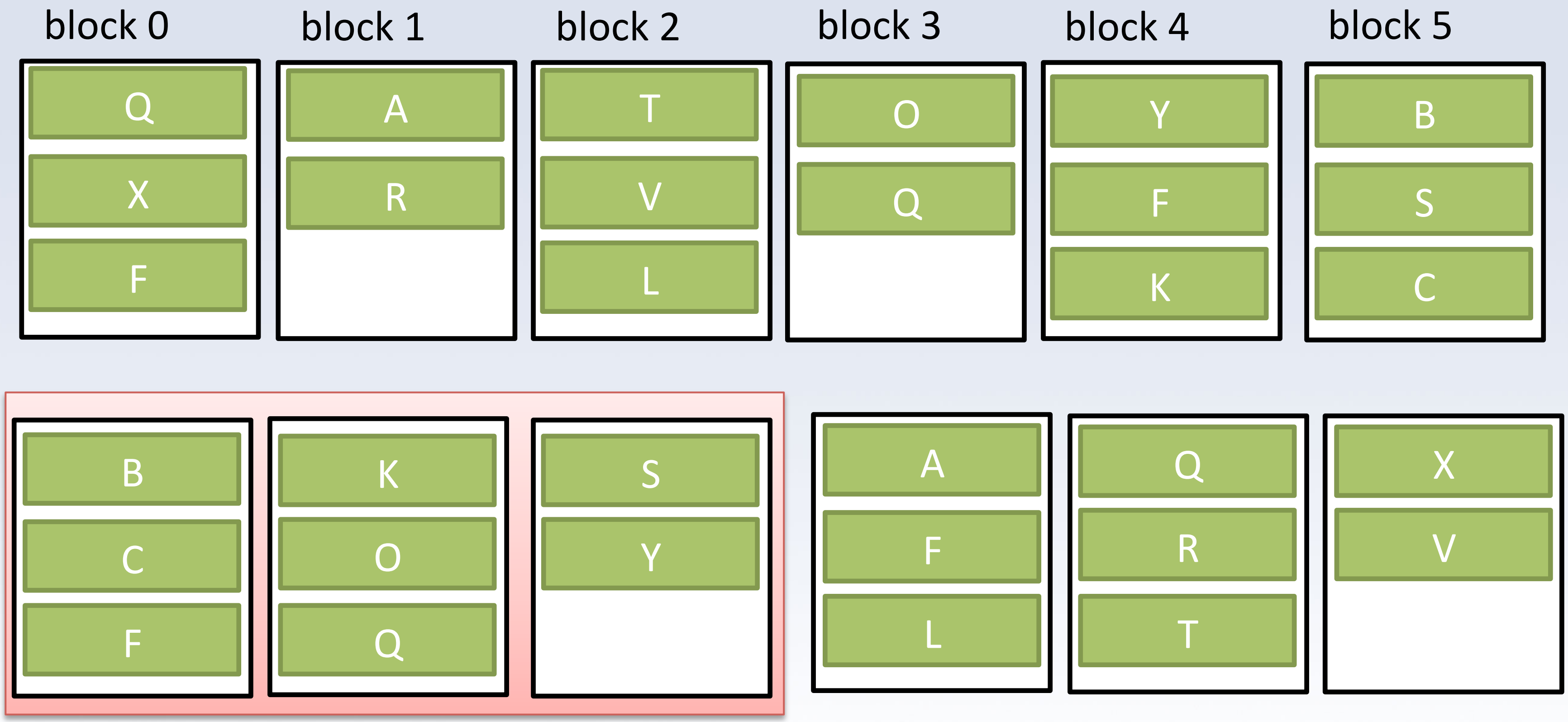
Phase 1


block 0
Q
X
F
block 1
A
R
block 2
T
V
L
block 3
O
Q
block 4
Y
F
K
block 5
B
S
C
A
F
L
Q
R
T
X
V

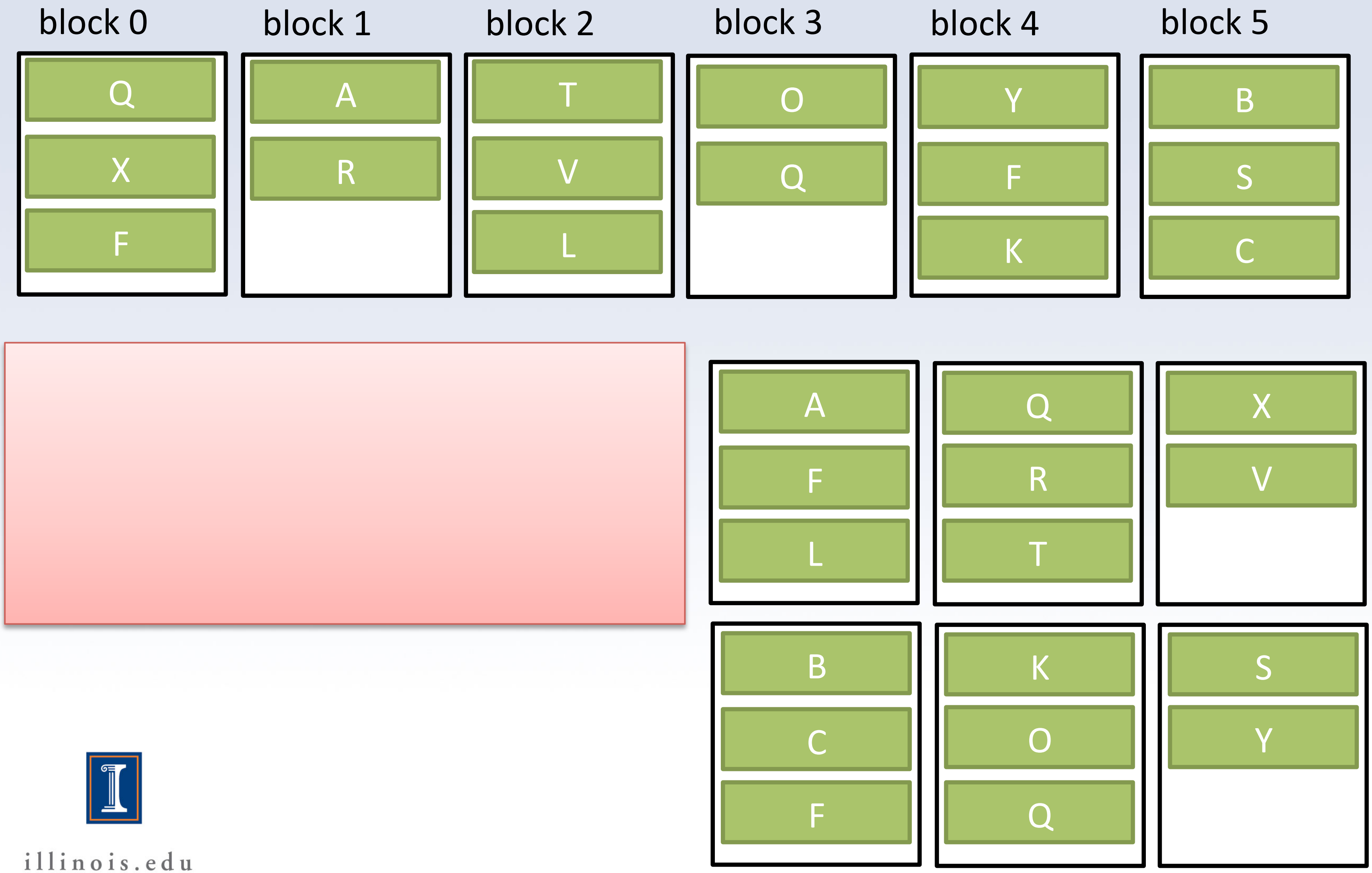
Phase 1


block 0
Q
X
F
block 1
A
R
block 2
T
V
L
block 3
O
Q
block 4
Y
F
K
block 5
B
S
C
O
Q
Y
F
K
B
S
C
A
F
L
Q
R
T
X
V

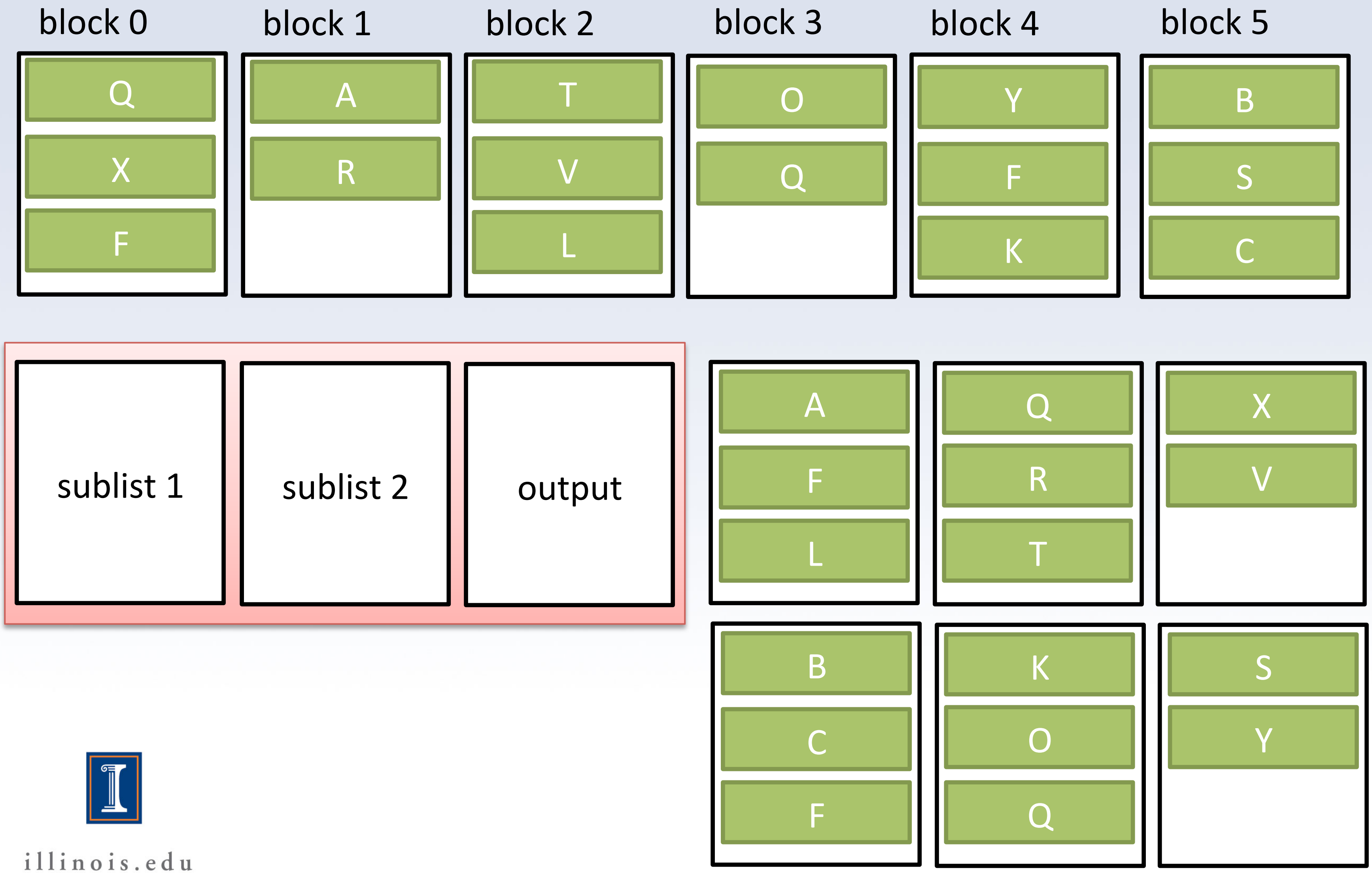
Phase 1


block 0
block 1
block 2
block 3
block 4
block 5
Q X F
A R
T V L
O Q
Y F K
B S C
B C F
K O Q
S Y
A F L
Q R T
X V

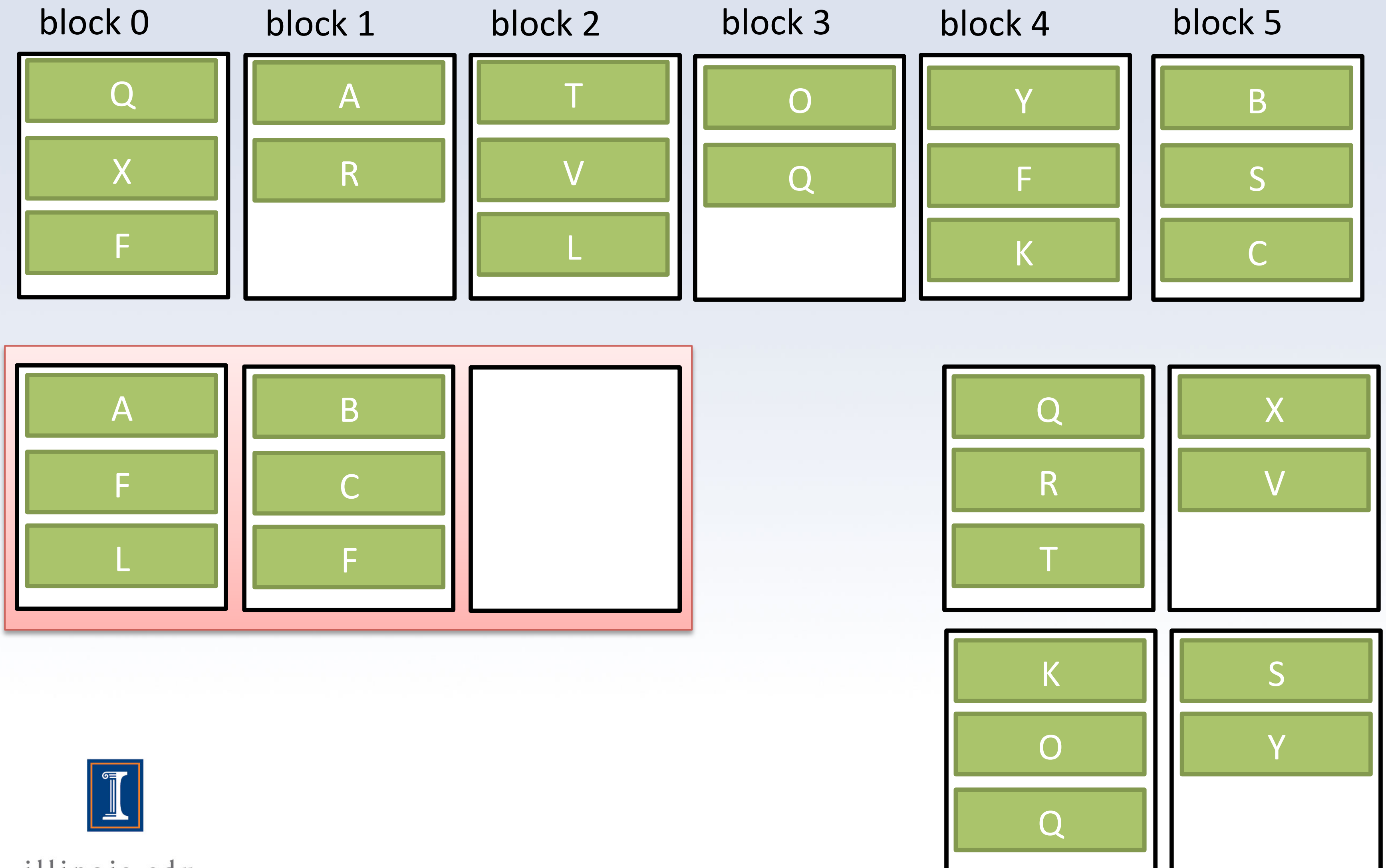
Phase 1


block 0
Q
X
F
block 1
A
R
block 2
T
V
L
block 3
O
Q
block 4
Y
F
K
block 5
B
S
C
A
F
L
Q
R
T
X
V
B
C
F
K
O
Q
S
Y

Phase 1

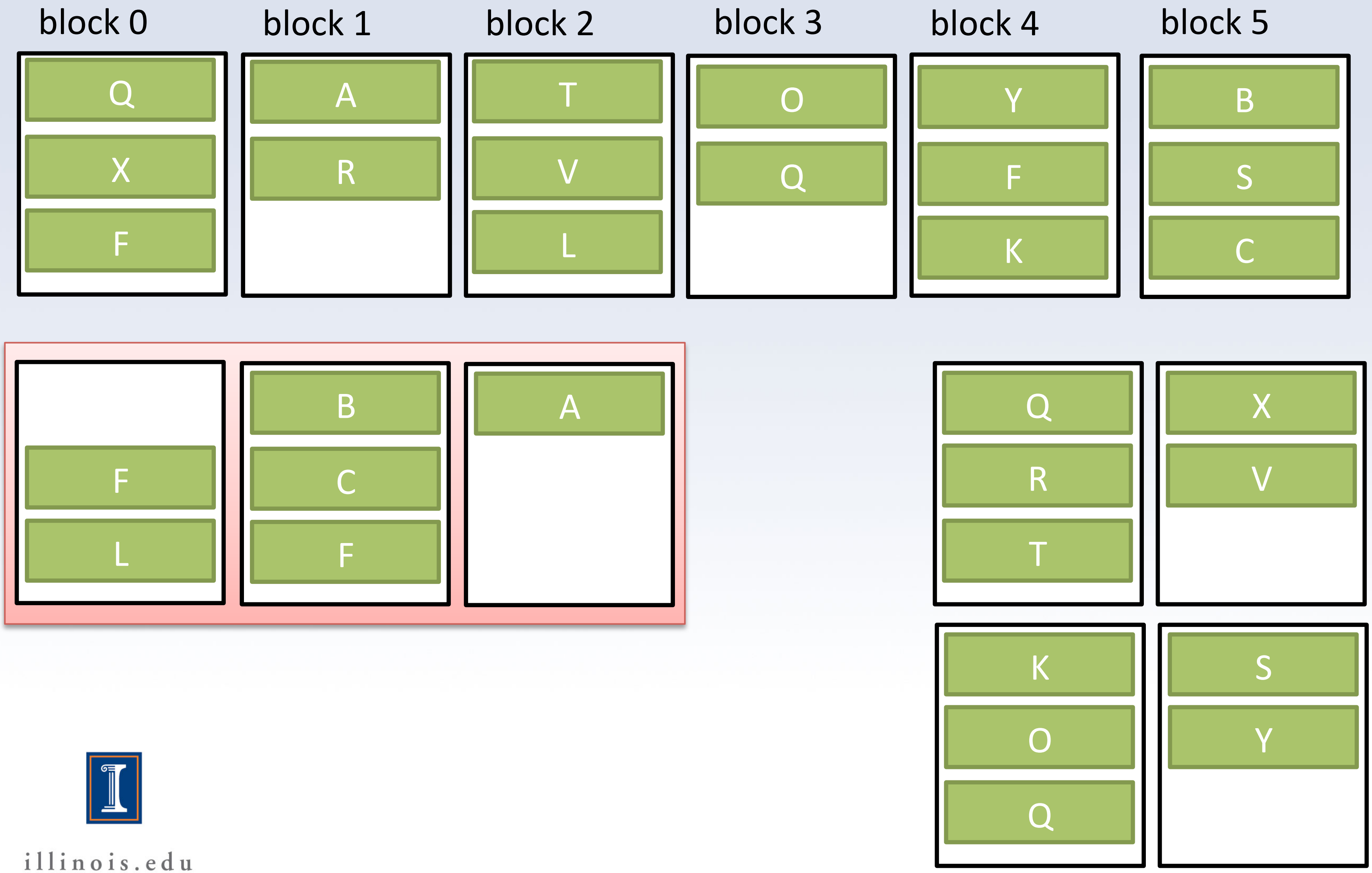

block 0
Q
X
F
block 1
A
R
block 2
T
V
L
block 3
O
Q
block 4
Y
F
K
block 5
B
S
C
sublist 1
sublist 2
output
A
F
L
Q
R
T
X
V
B
C
F
K
O
Q
S
Y

Phase 2



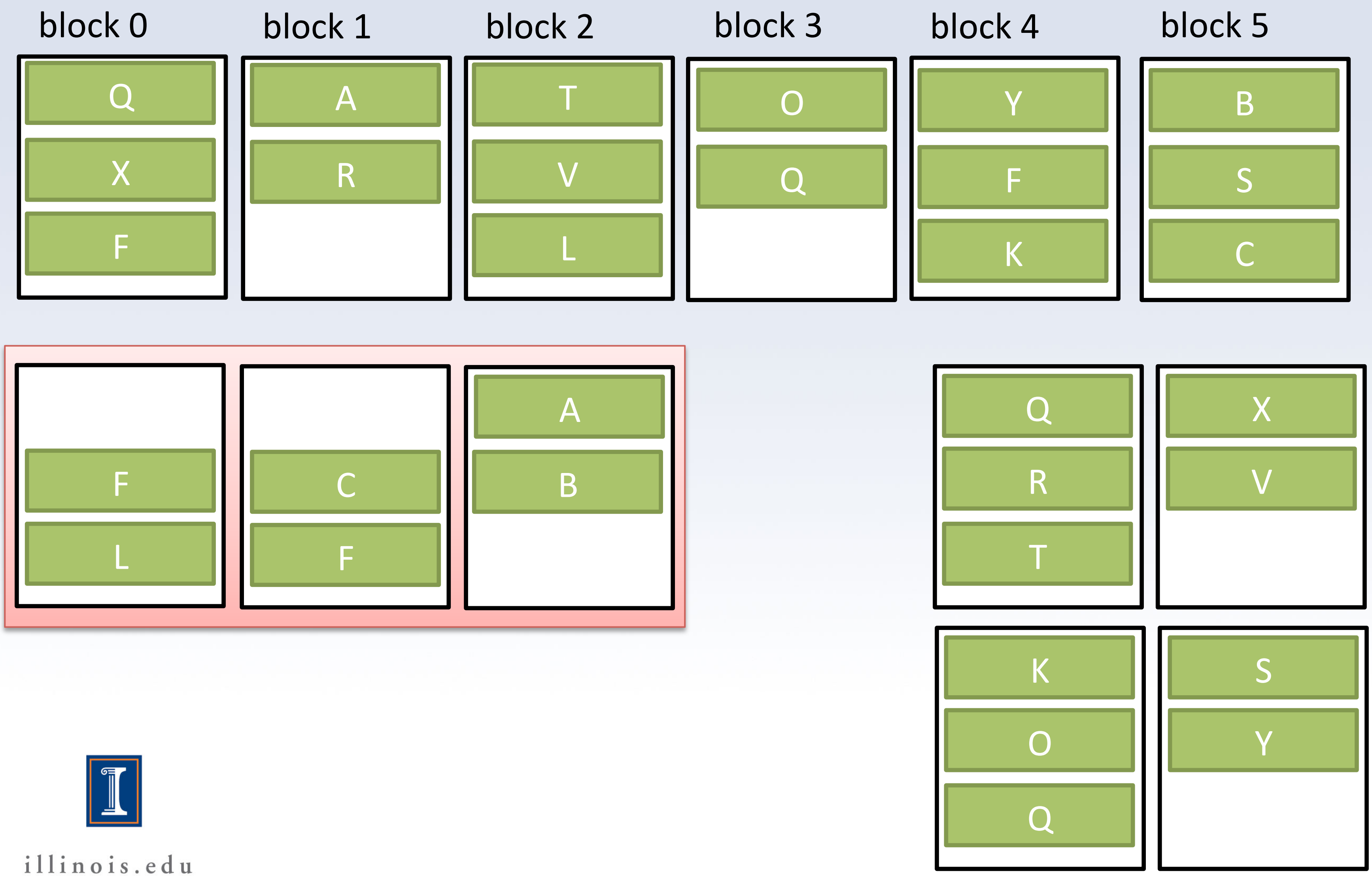
illinois.edu

Phase 2



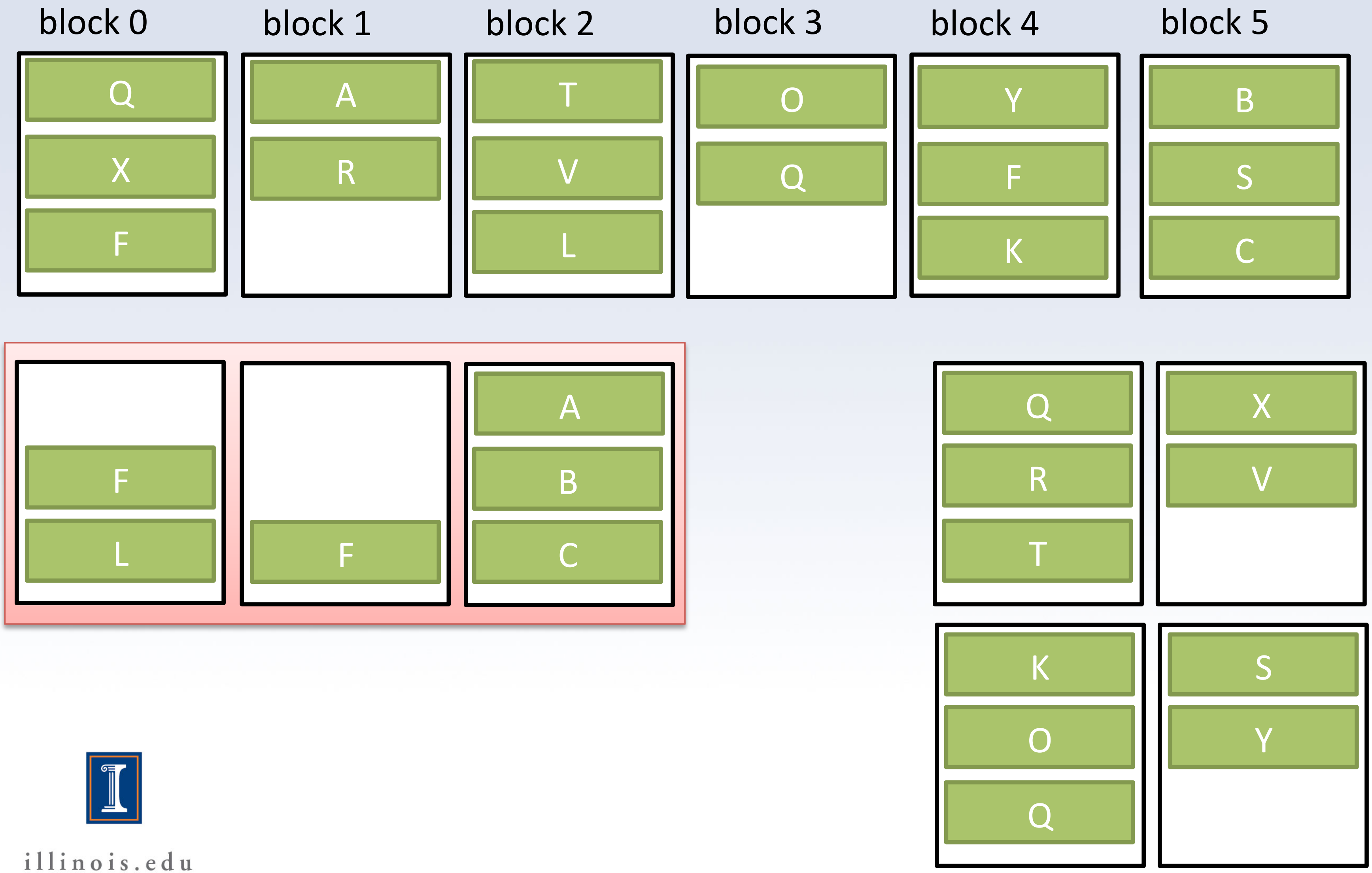
illinois.edu

Phase 2



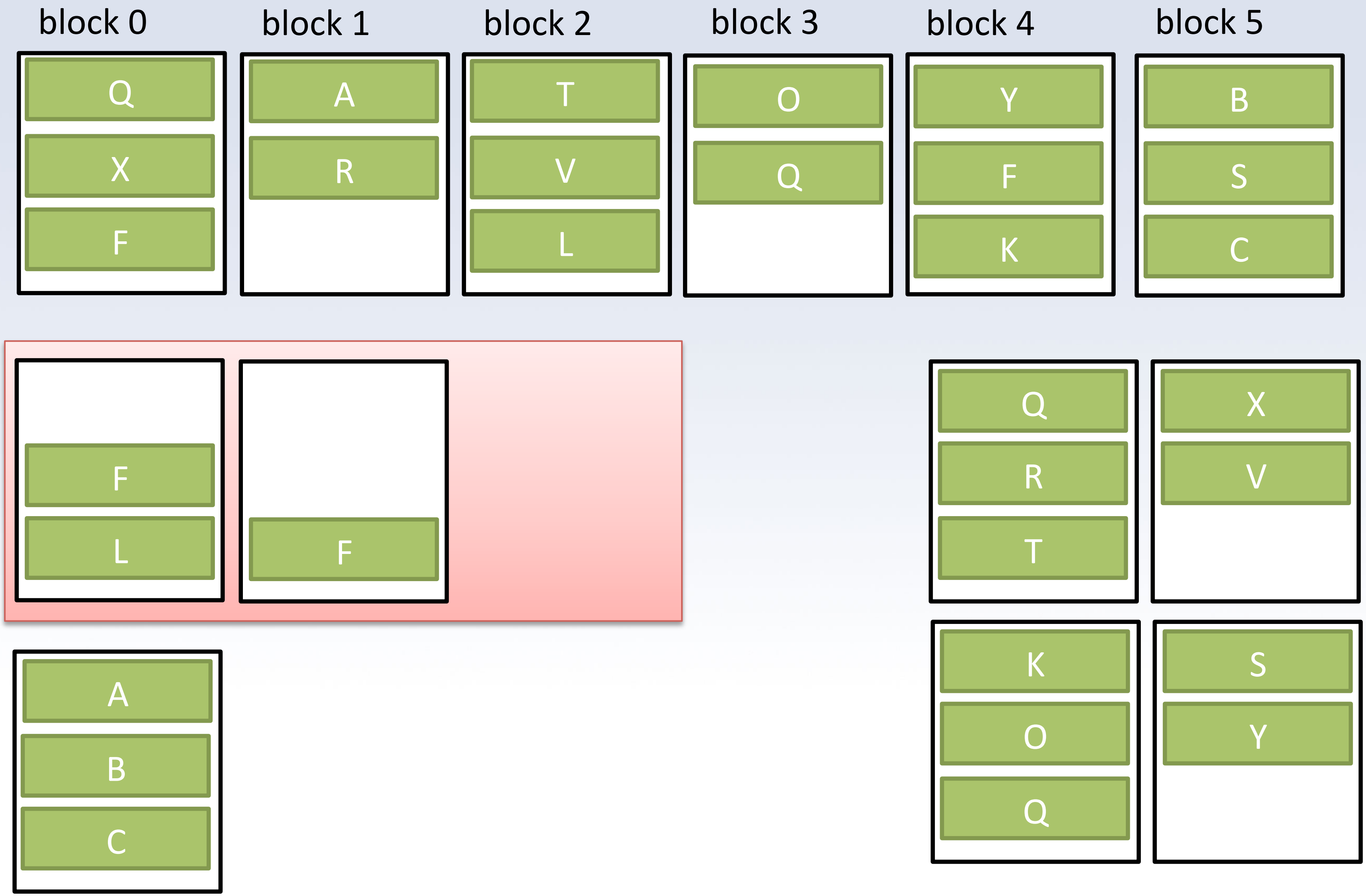
illinois.edu

Phase 2

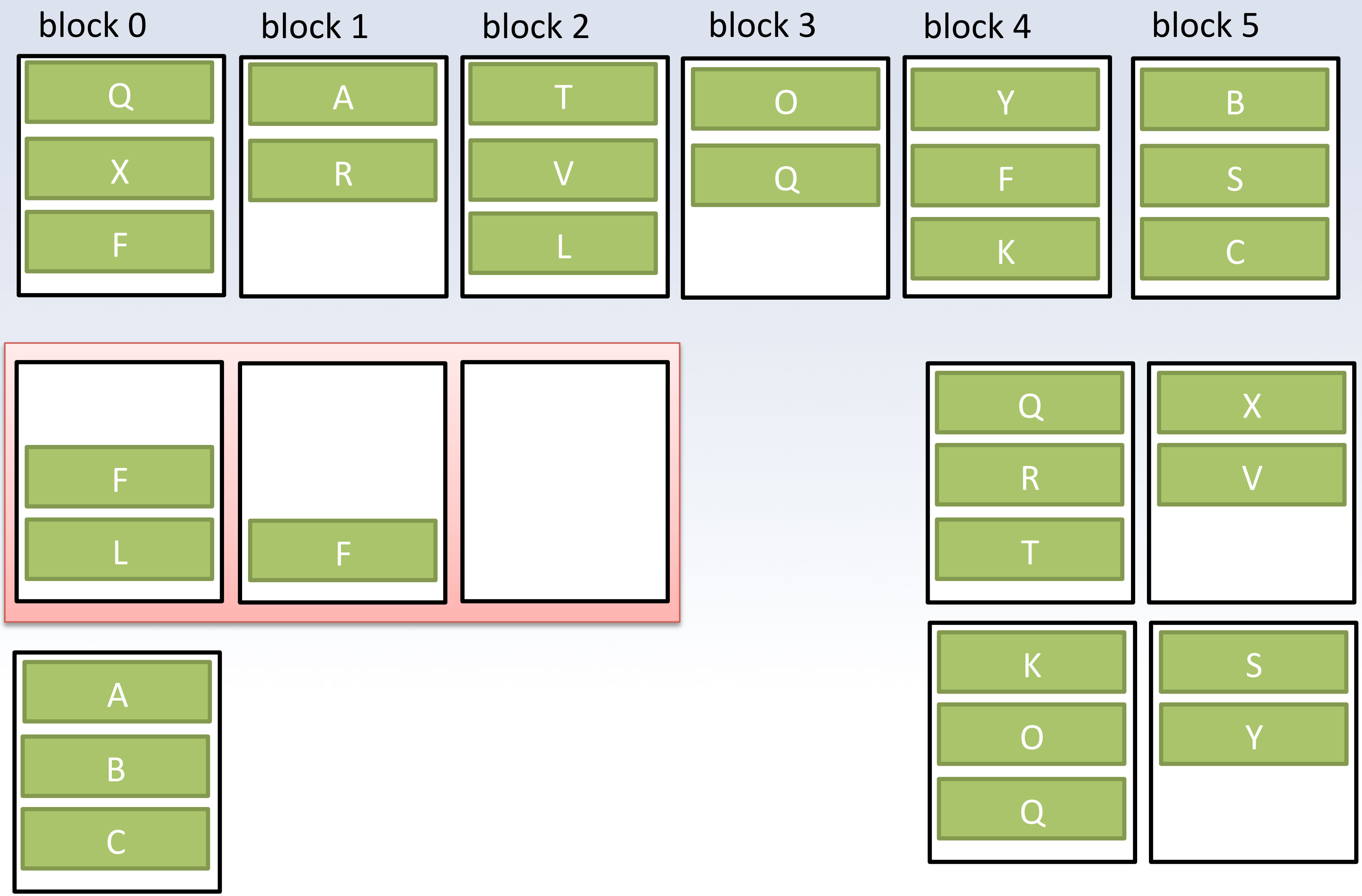


illinois.edu

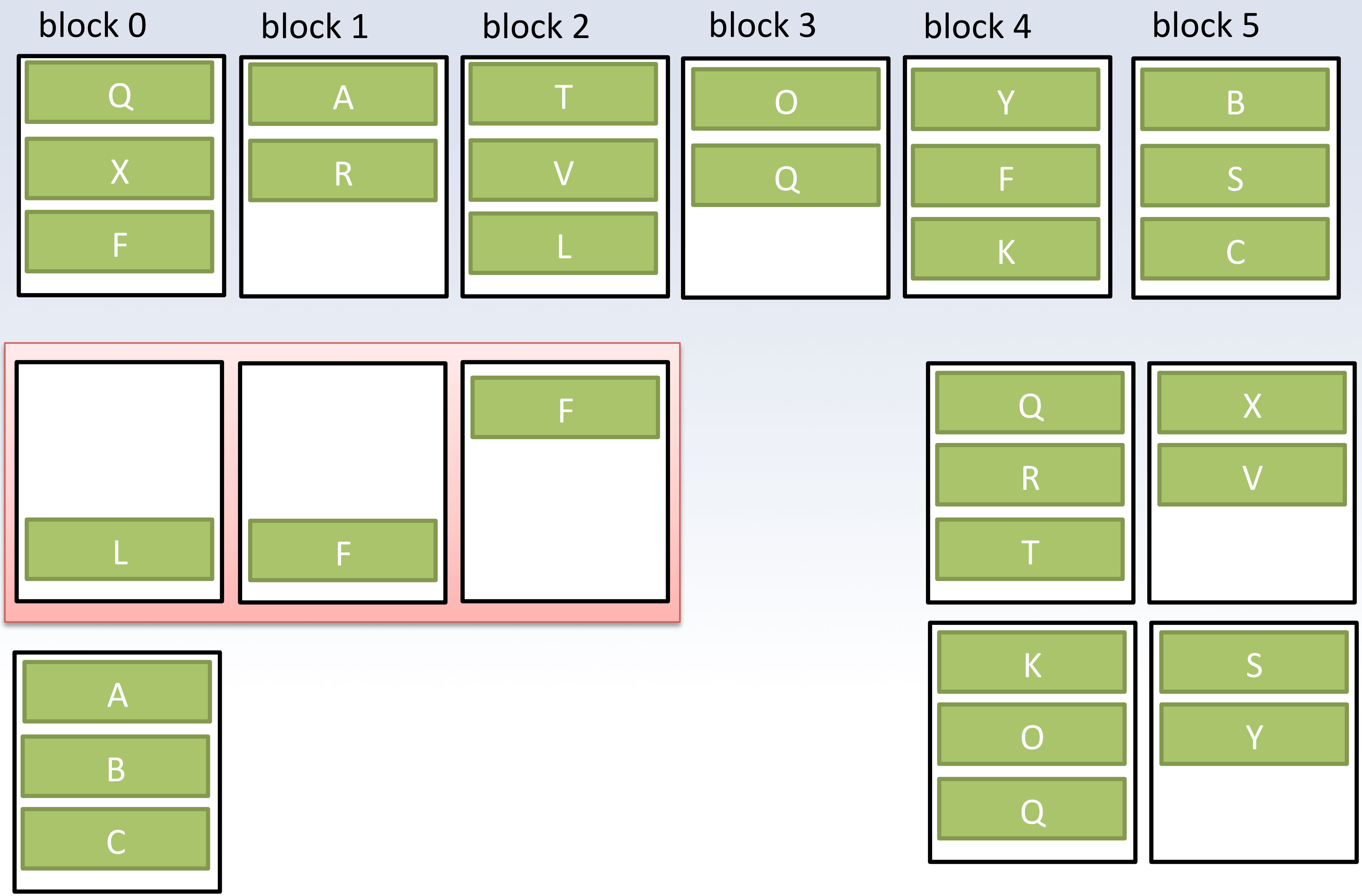
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
F
L
F
Q
R
T
X
V
A
B
C
K
O
Q
S
Y

Phase 2

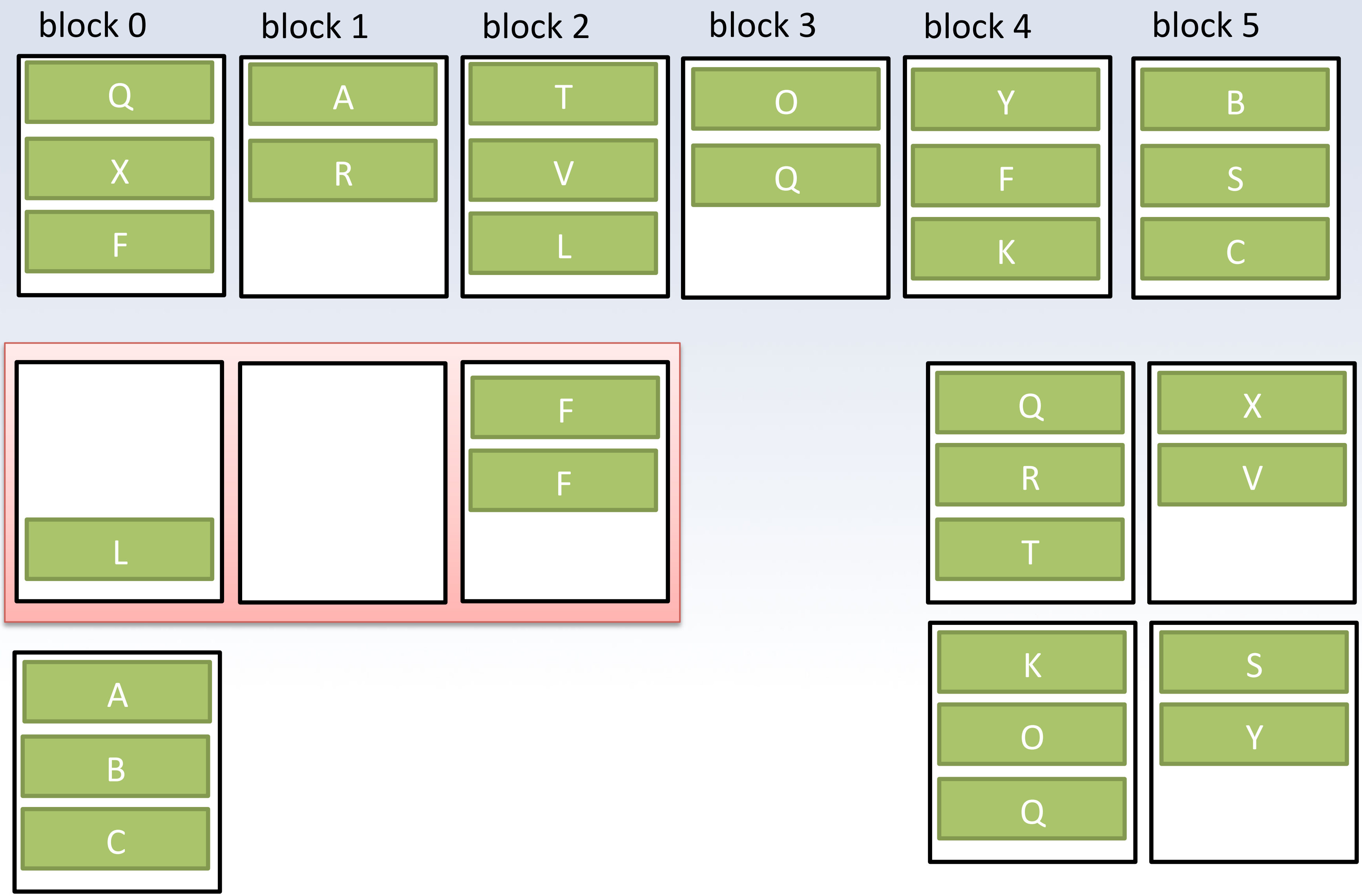

block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
F
L
F
Q
R
T
X
V
A
B
C
K
O
Q
S
Y

Phase 2

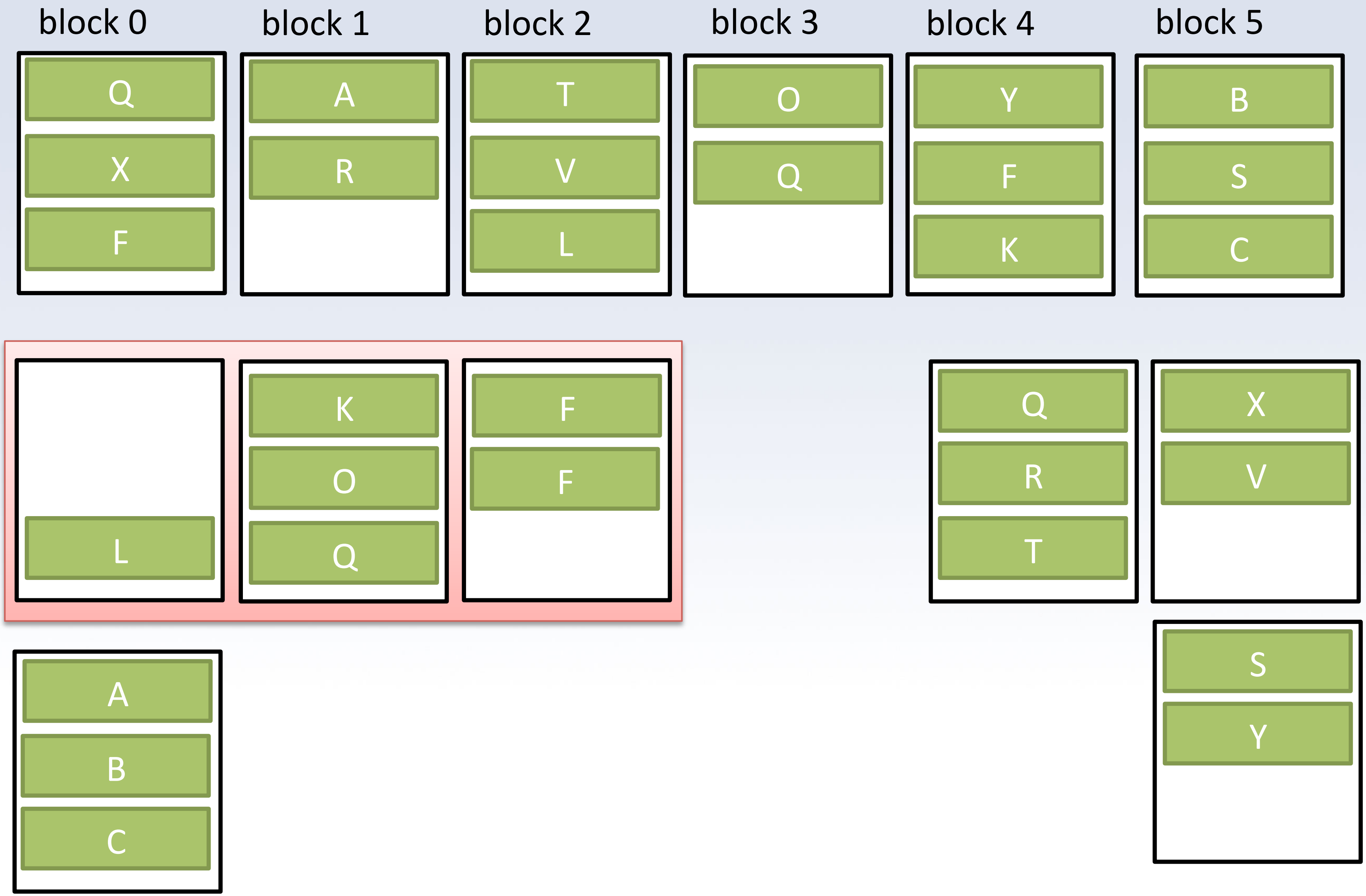


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
L
F
F
A
B
C
Q
R
T
X
V
K
O
Q
S
Y

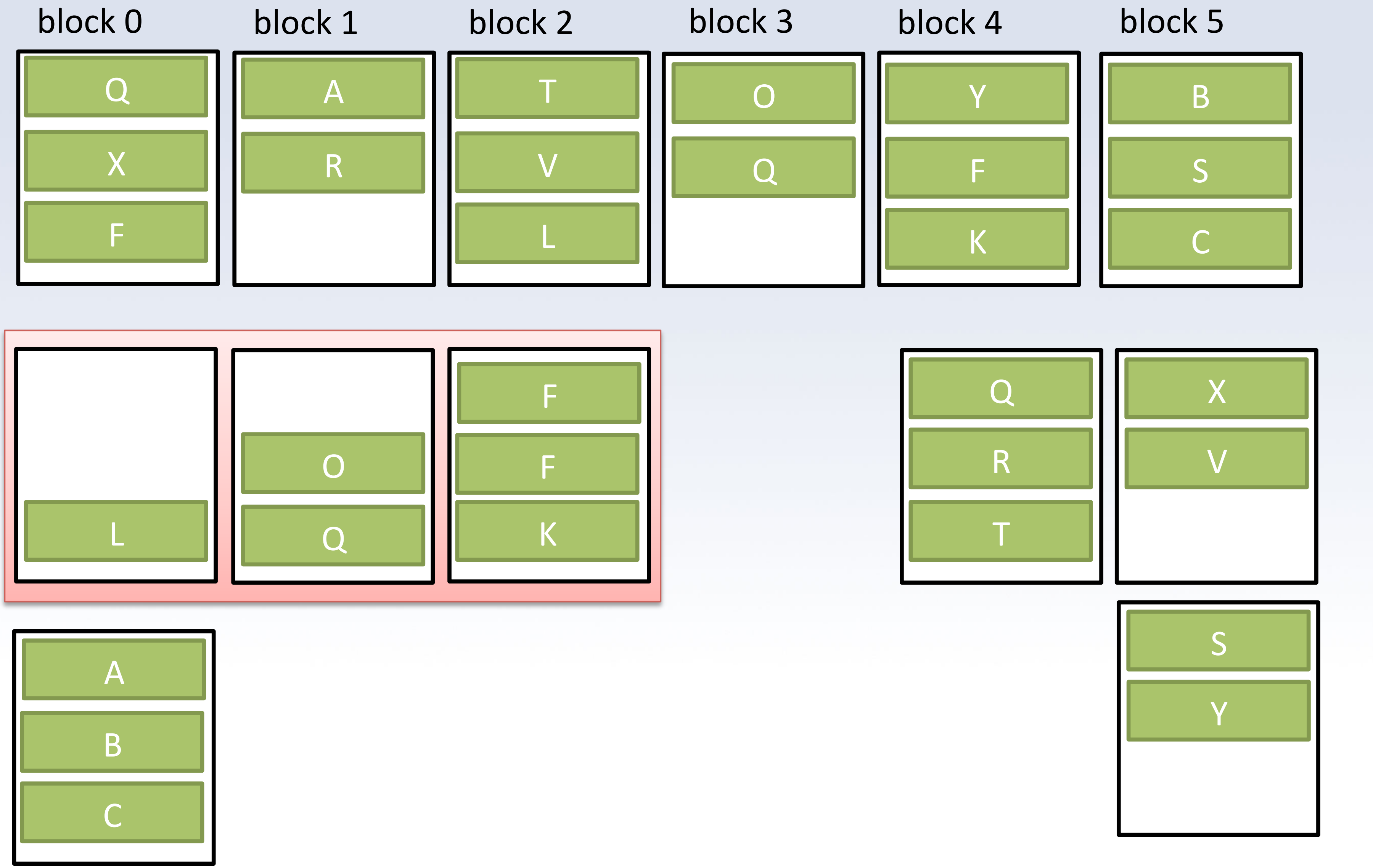
Phase 2


block 0
Q
X
F
block 1
A
R
block 2
T
V
L
block 3
O
Q
block 4
Y
F
K
block 5
B
S
C
L
F
F
Q
R
T
X
V
K
O
Q
S
Y
A
B
C

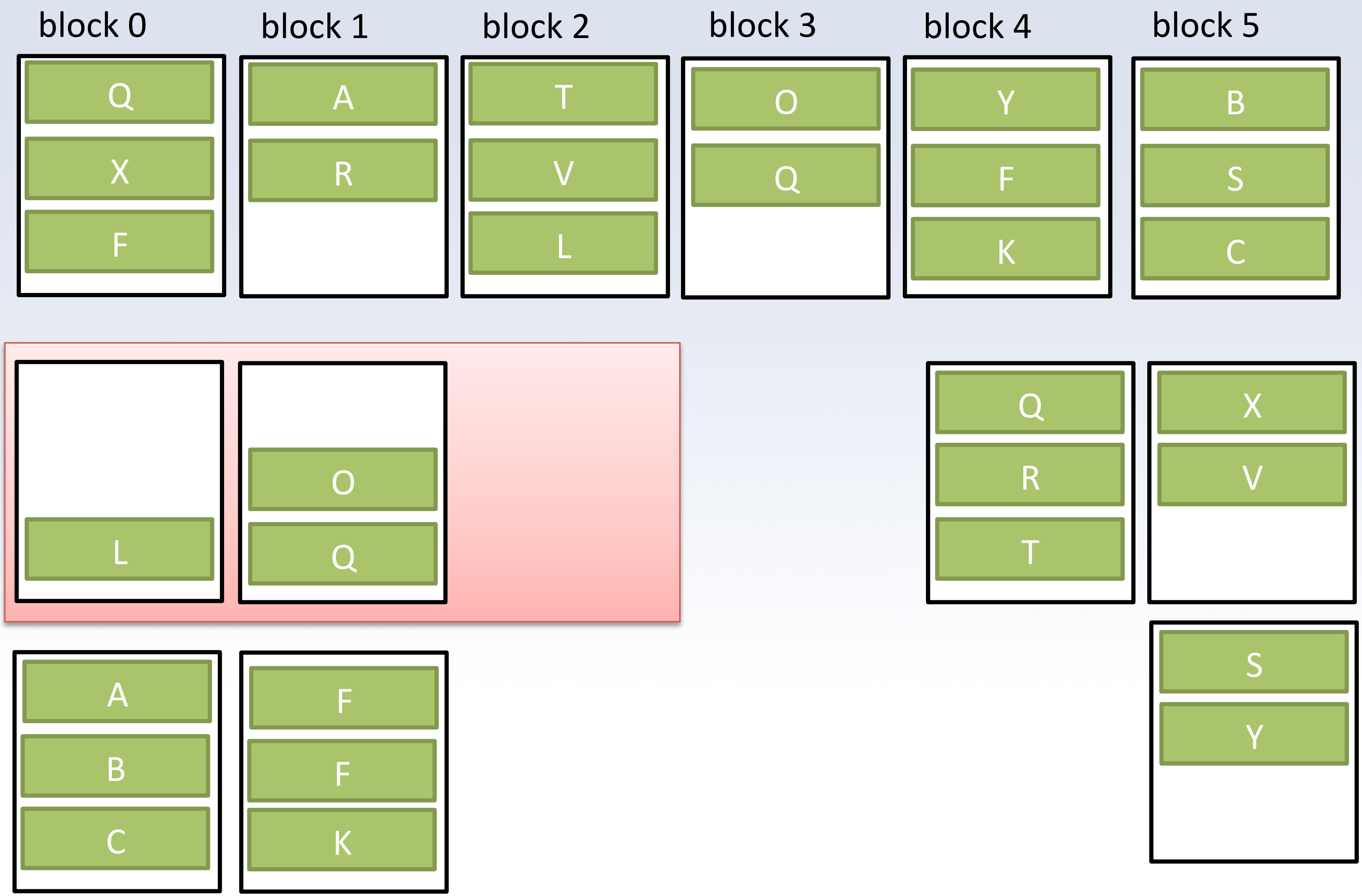
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
L
K
O
Q
F
F
Q
R
T
X
V
S
Y
A
B
C

Phase 2

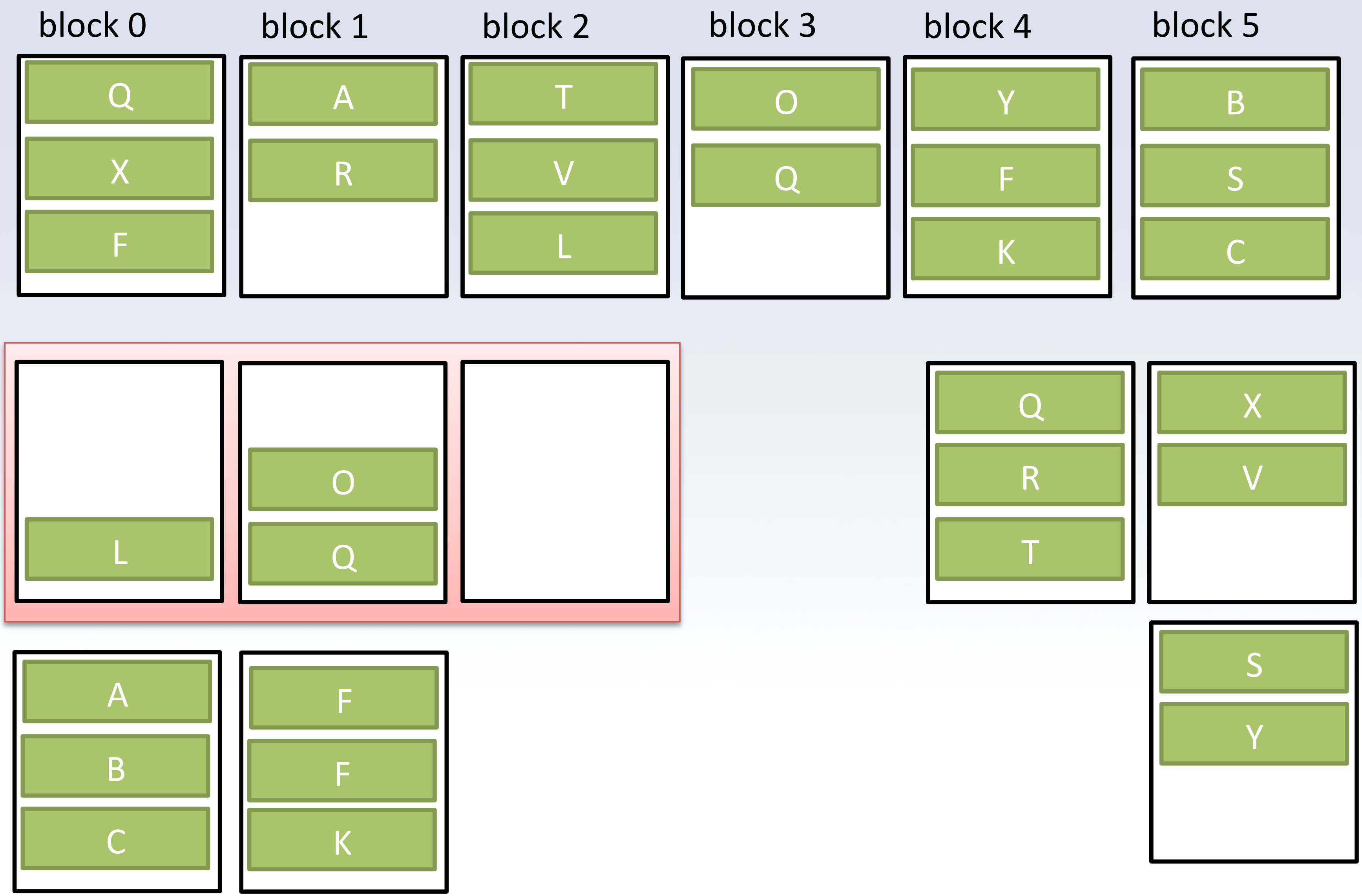

block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
L
O
Q
F
F
K
Q
R
T
X
V
S
Y
A
B
C

Phase 2



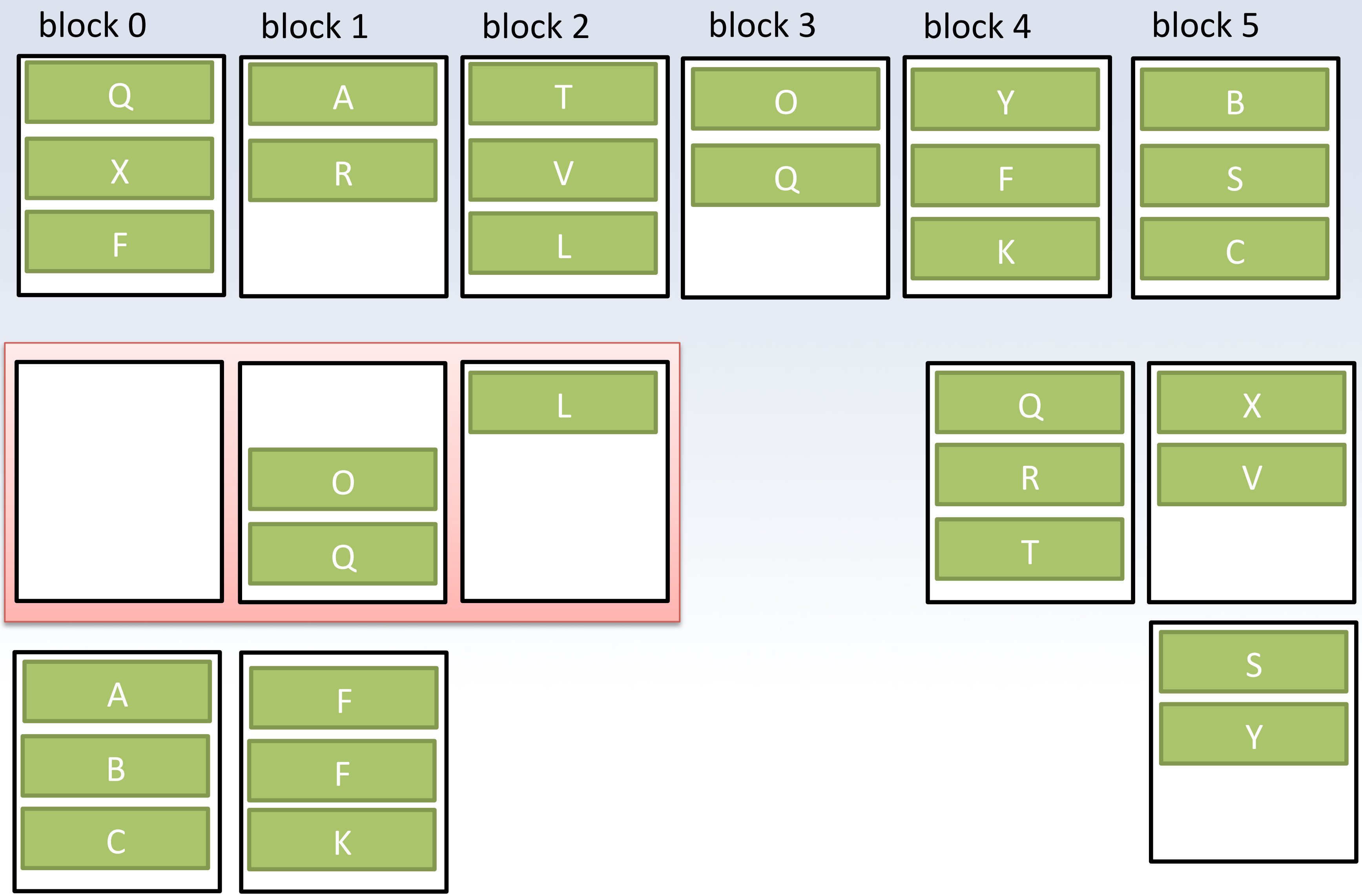
block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
L
O
Q
Q
R
T
X
V
A
B
C
F
F
K
S
Y

Phase 2



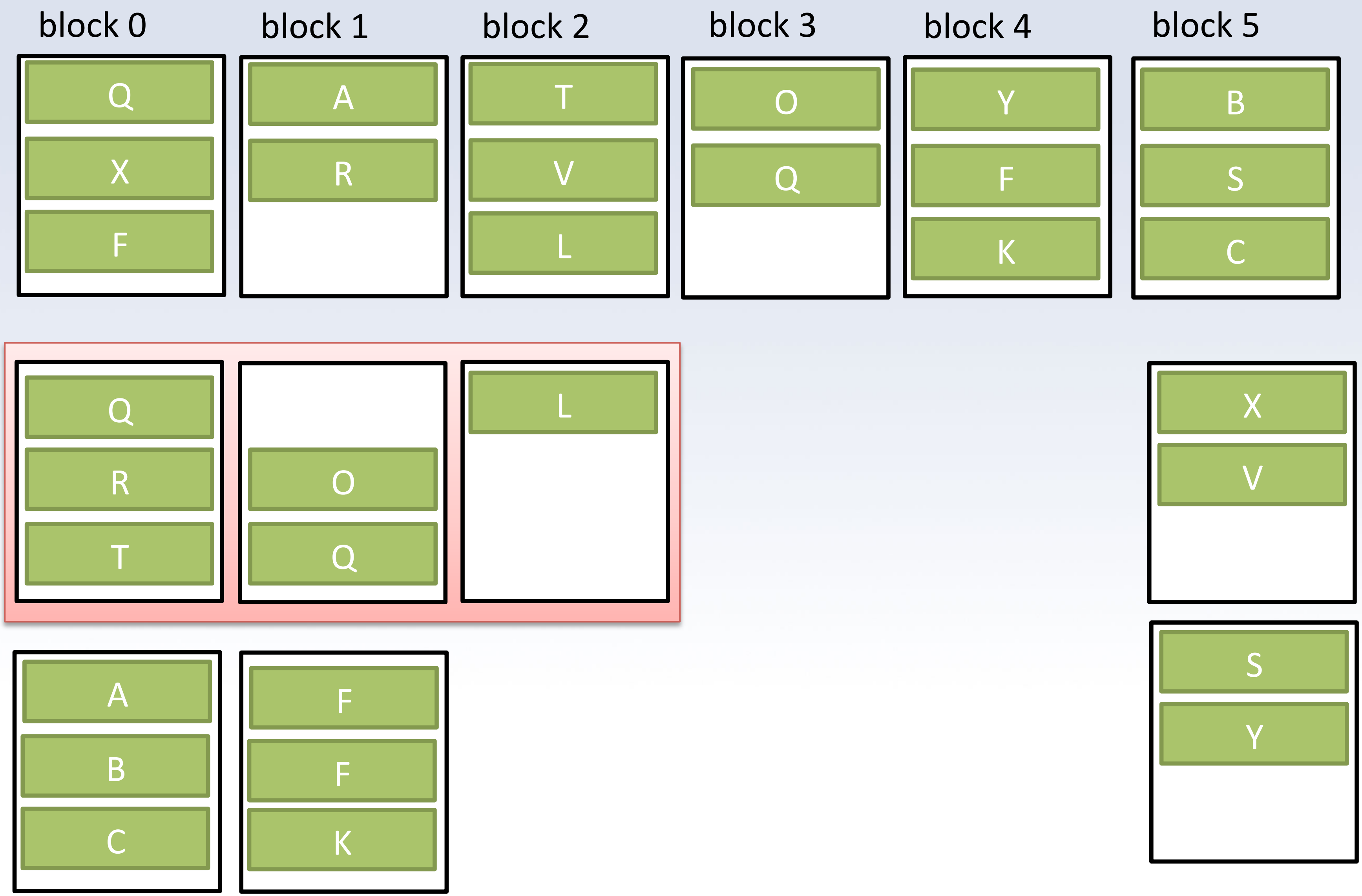
block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
L
O
Q
Q
R
T
X
V
A
B
C
F
F
K
S
Y

Phase 2

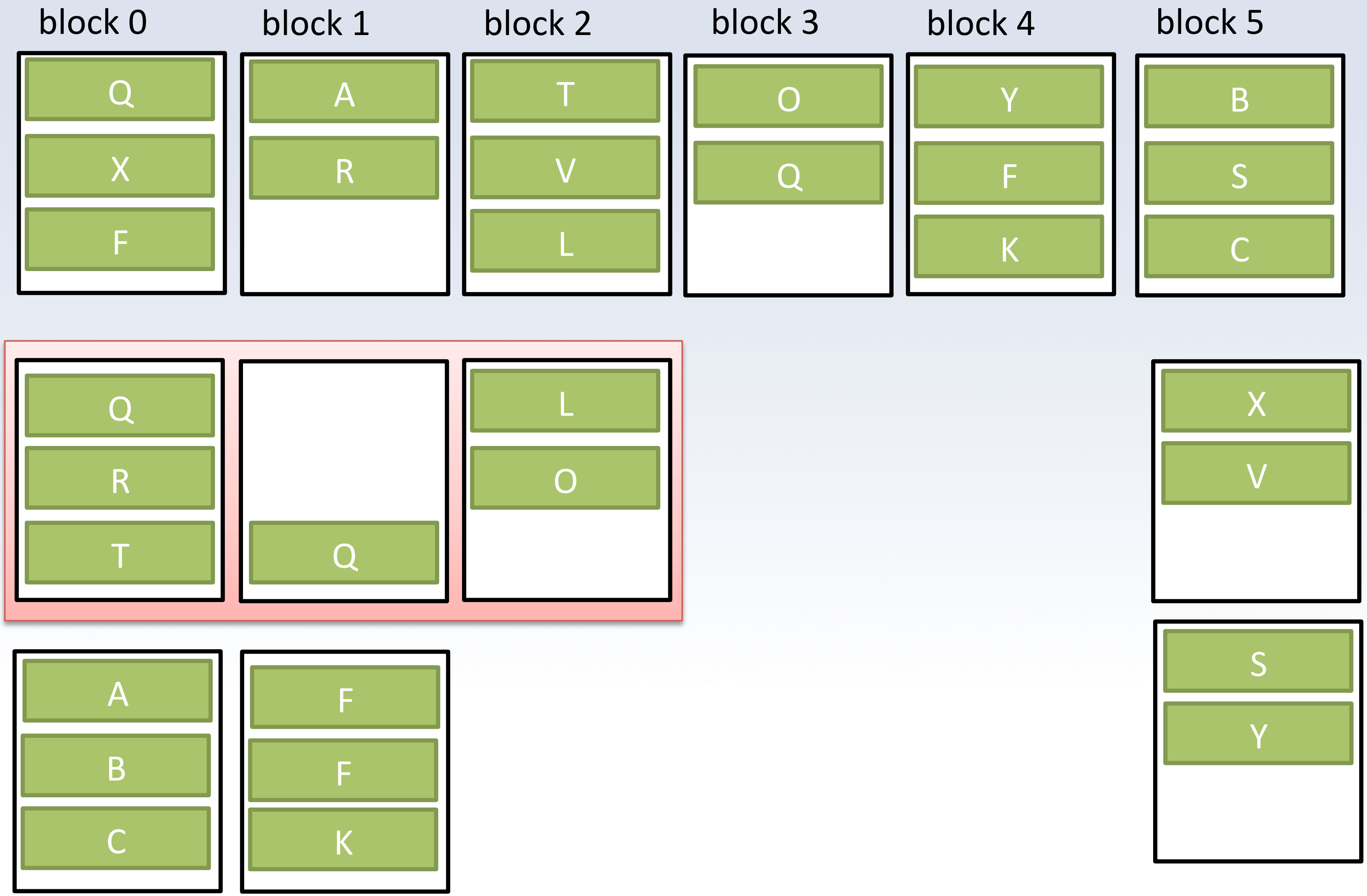


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
O
Q
L
Q
R
T
X
V
S
Y
A
B
C
F
F
K

Phase 2

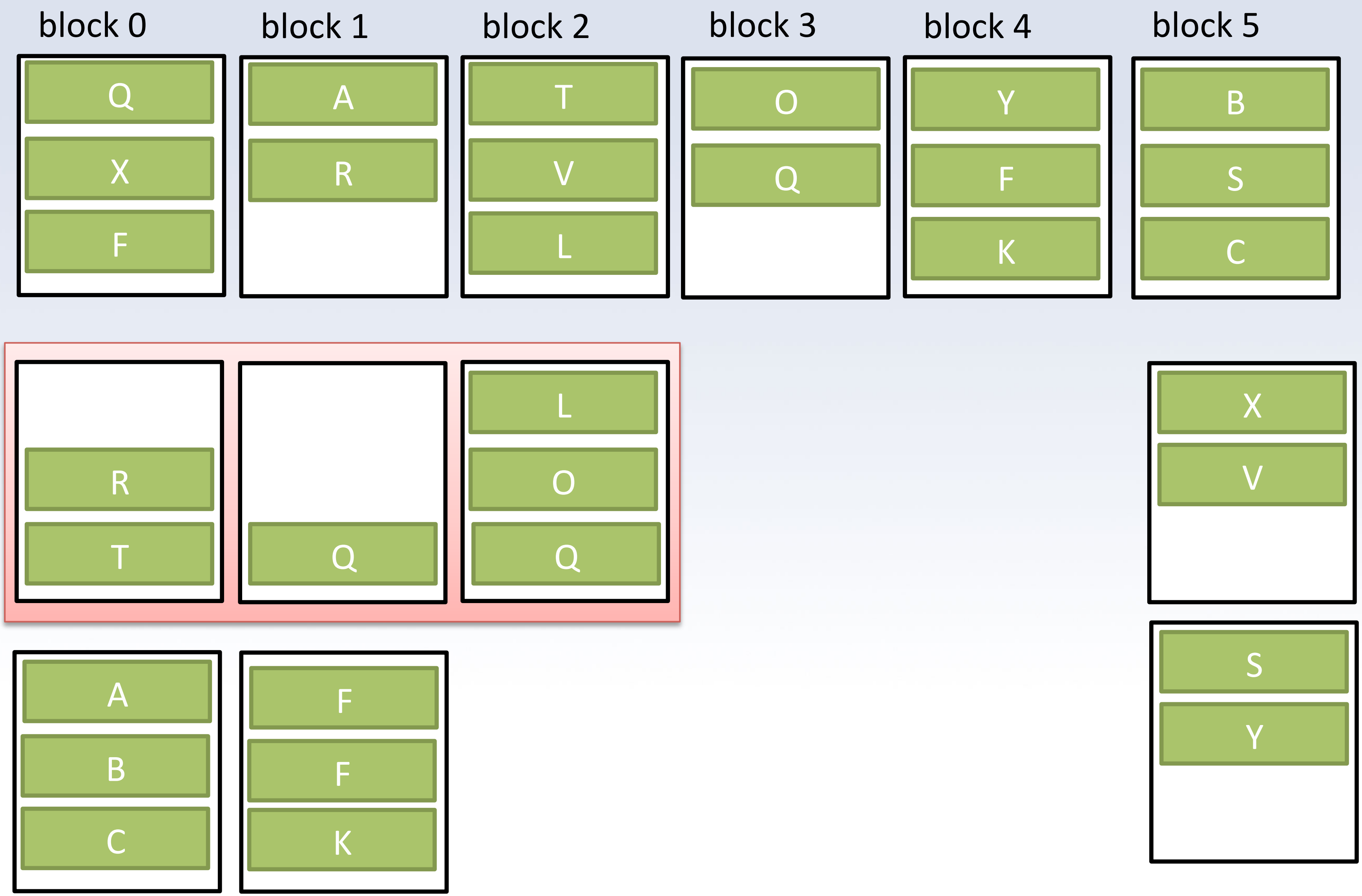

block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Q
R
T
O
Q
L
X
V
A
B
C
F
F
K
S
Y

Phase 2

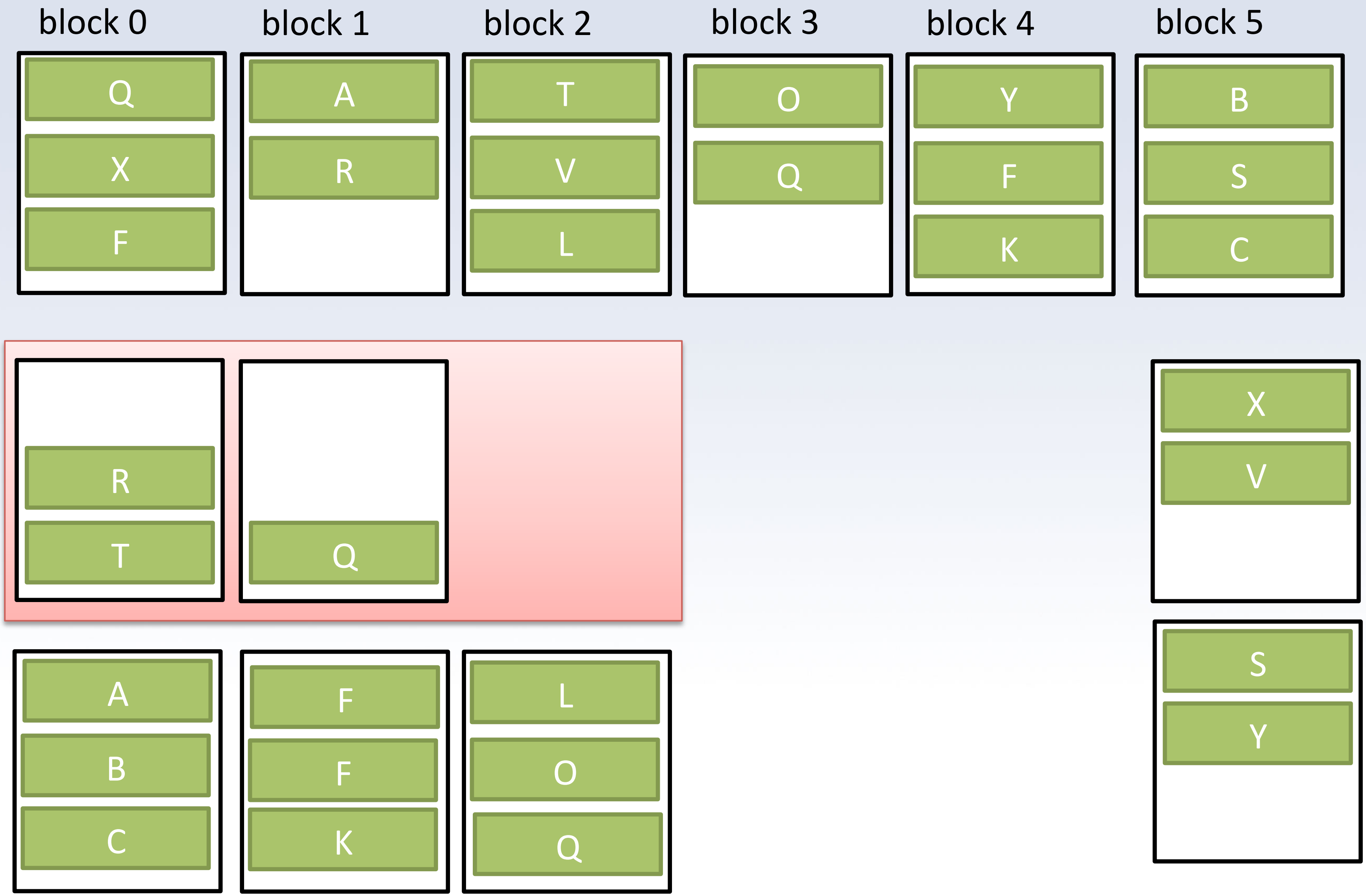


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Q
R
T
Q
L
O
X
V
A
B
C
F
F
K
S
Y

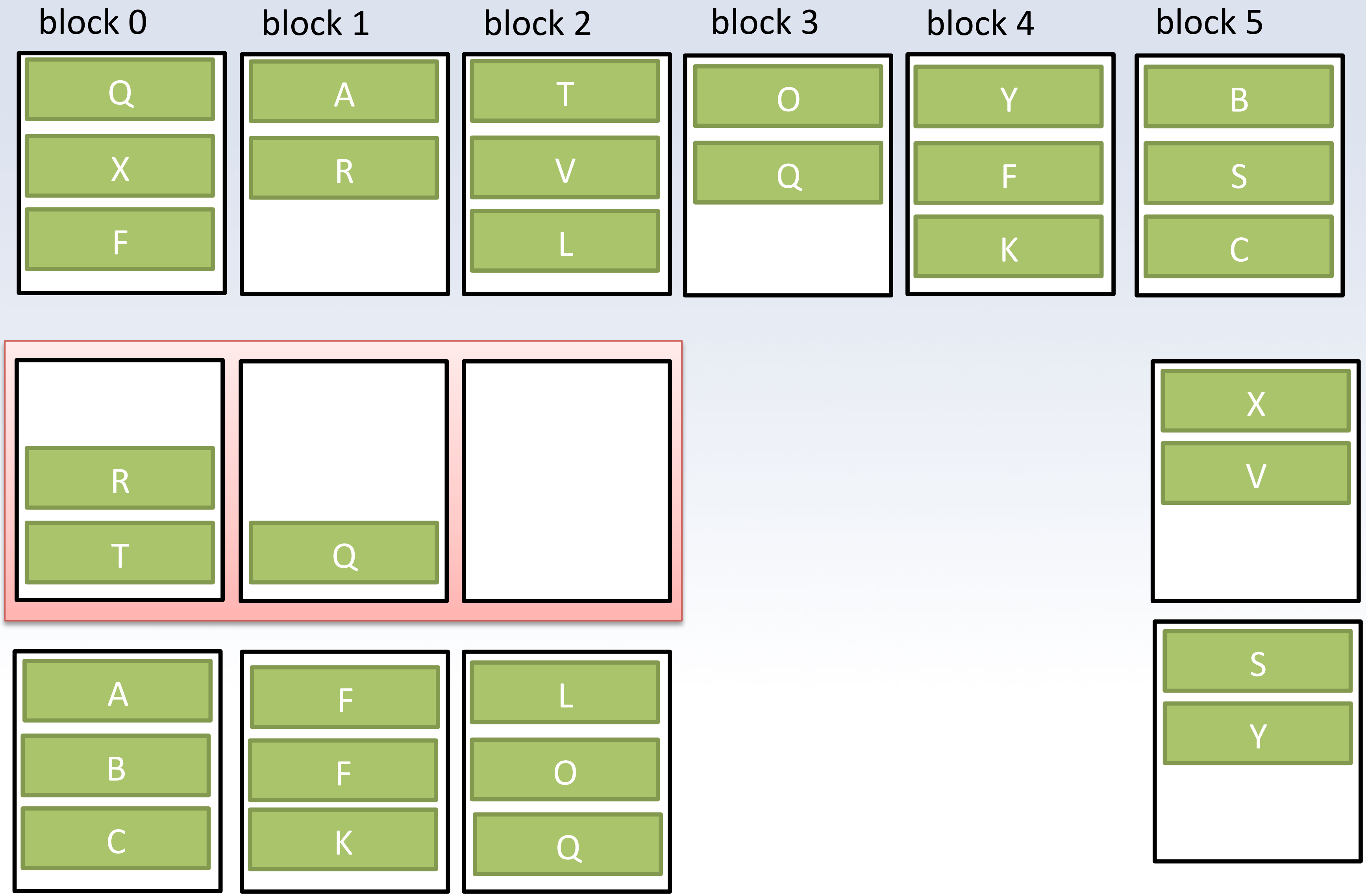
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
R
T
Q
L
O
Q
X
V
A
B
C
F
F
K
S
Y

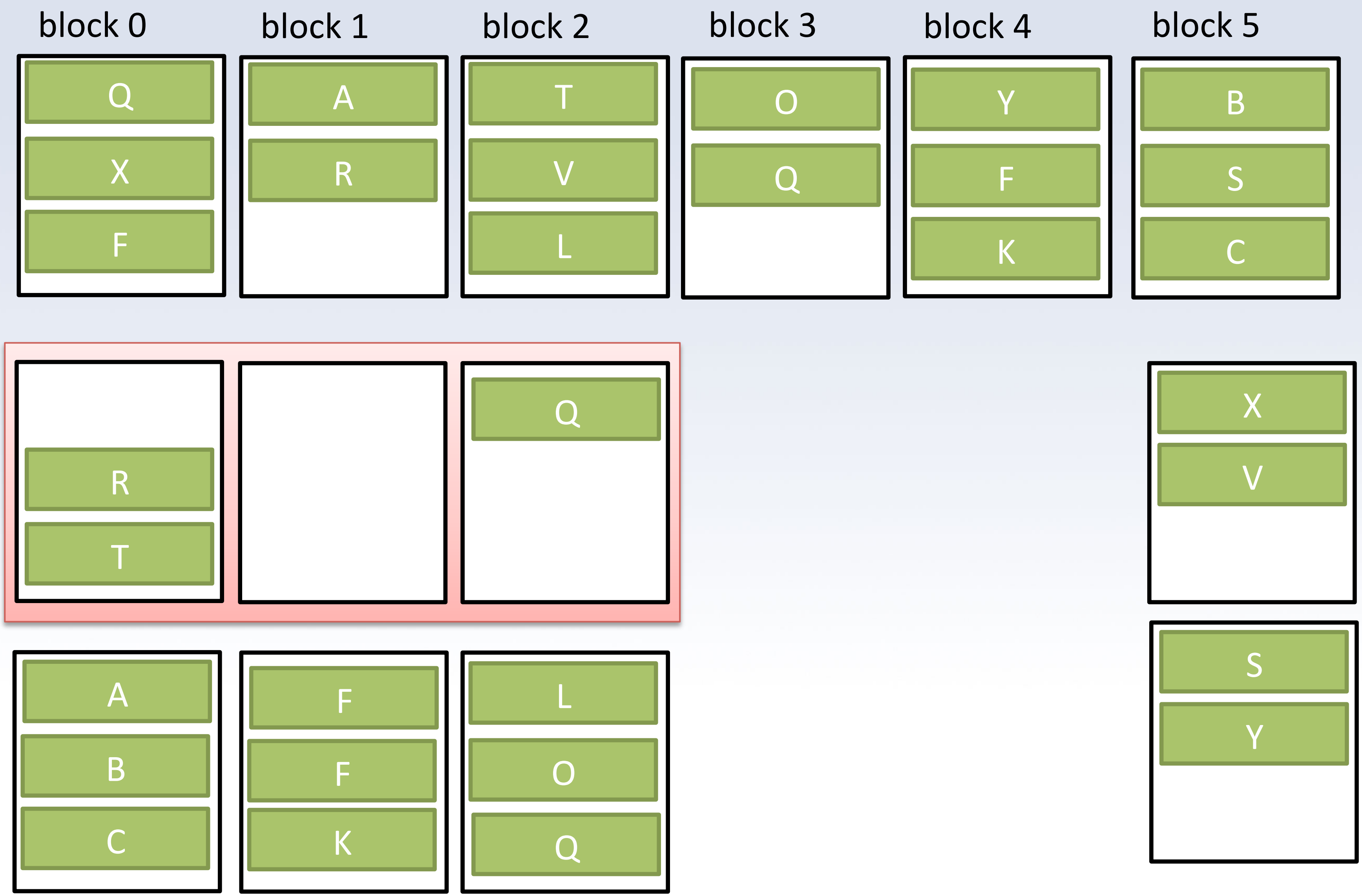
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
R
T
Q
X
V
A
B
C
F
F
K
L
O
Q
S
Y

Phase 2

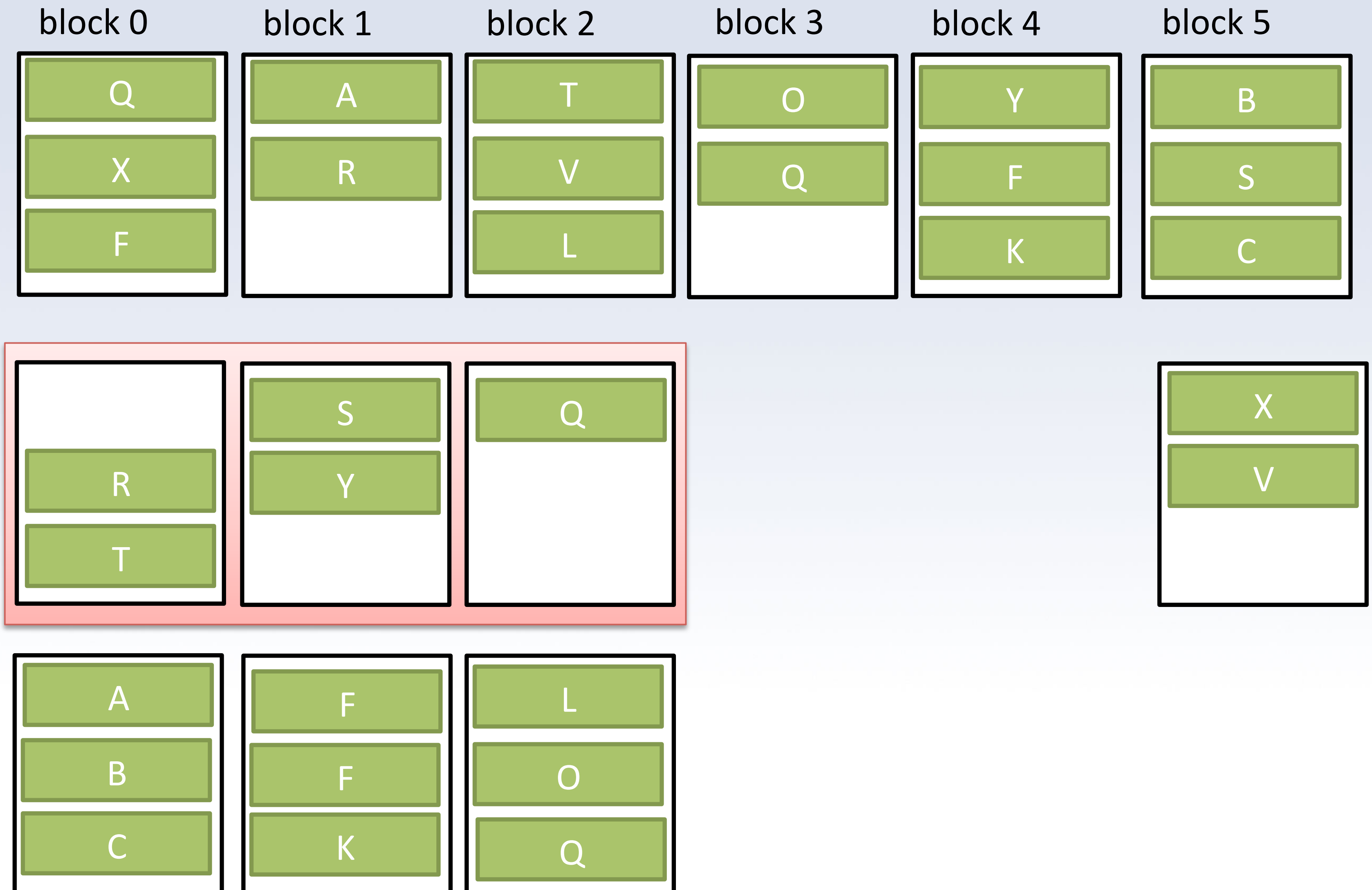

block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
R
T
Q
X
V
A
B
C
F
F
K
L
O
Q
S
Y

Phase 2



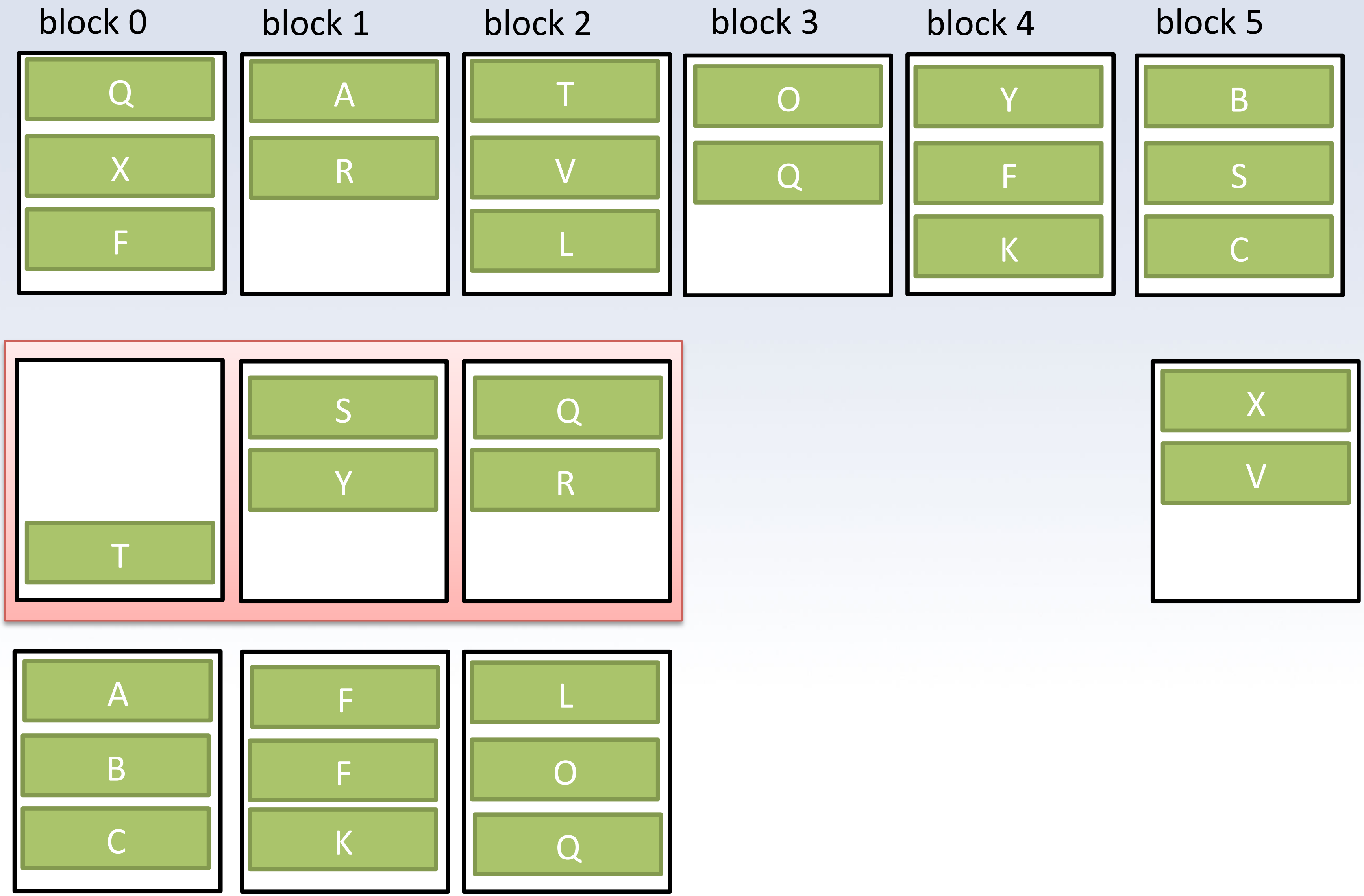
block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
R
T
Q
X
V
A
B
C
F
F
K
L
O
Q
S
Y

Phase 2

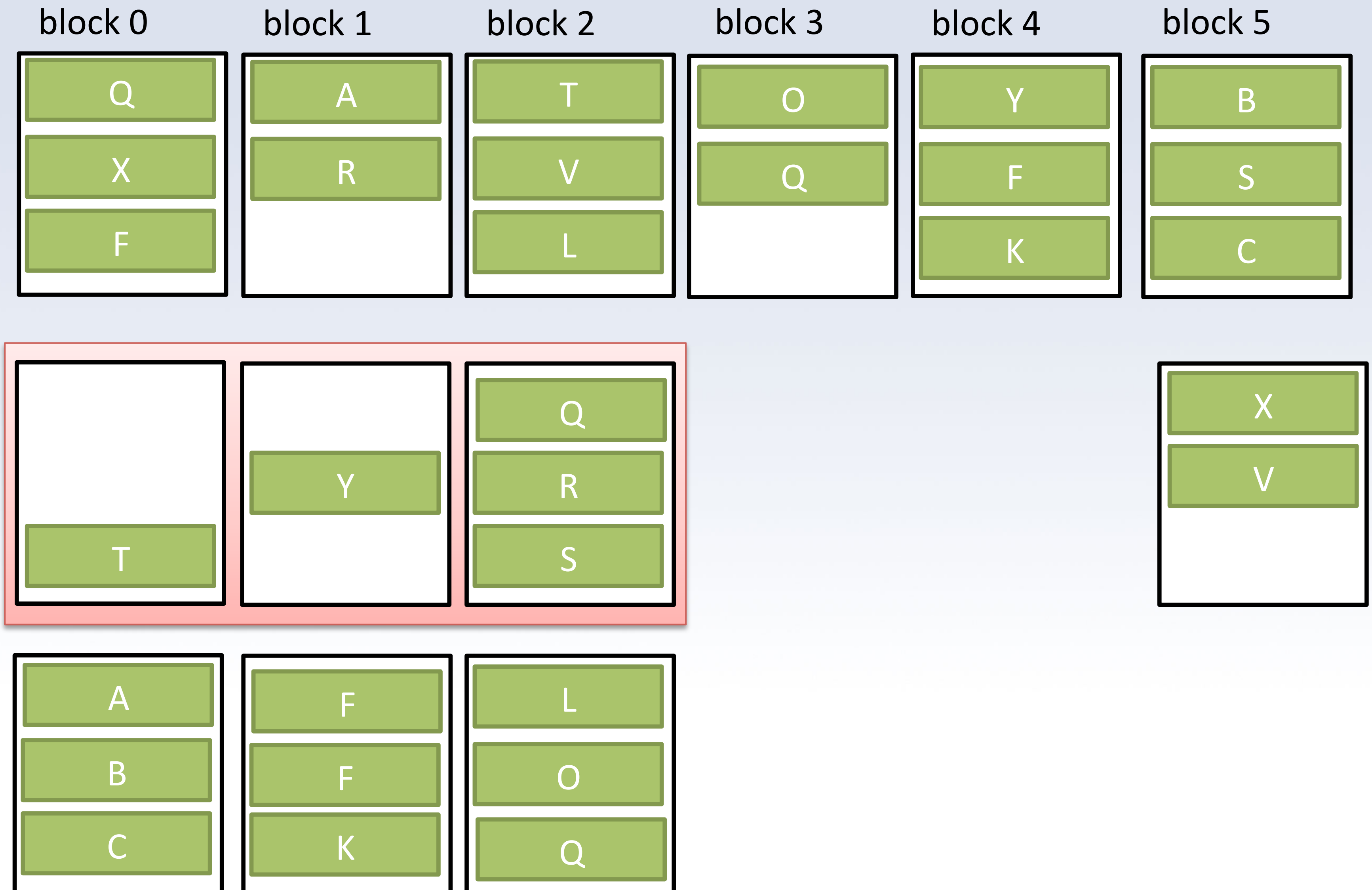


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
R
T
S
Y
Q
X
V
A
B
C
F
F
K
L
O
Q

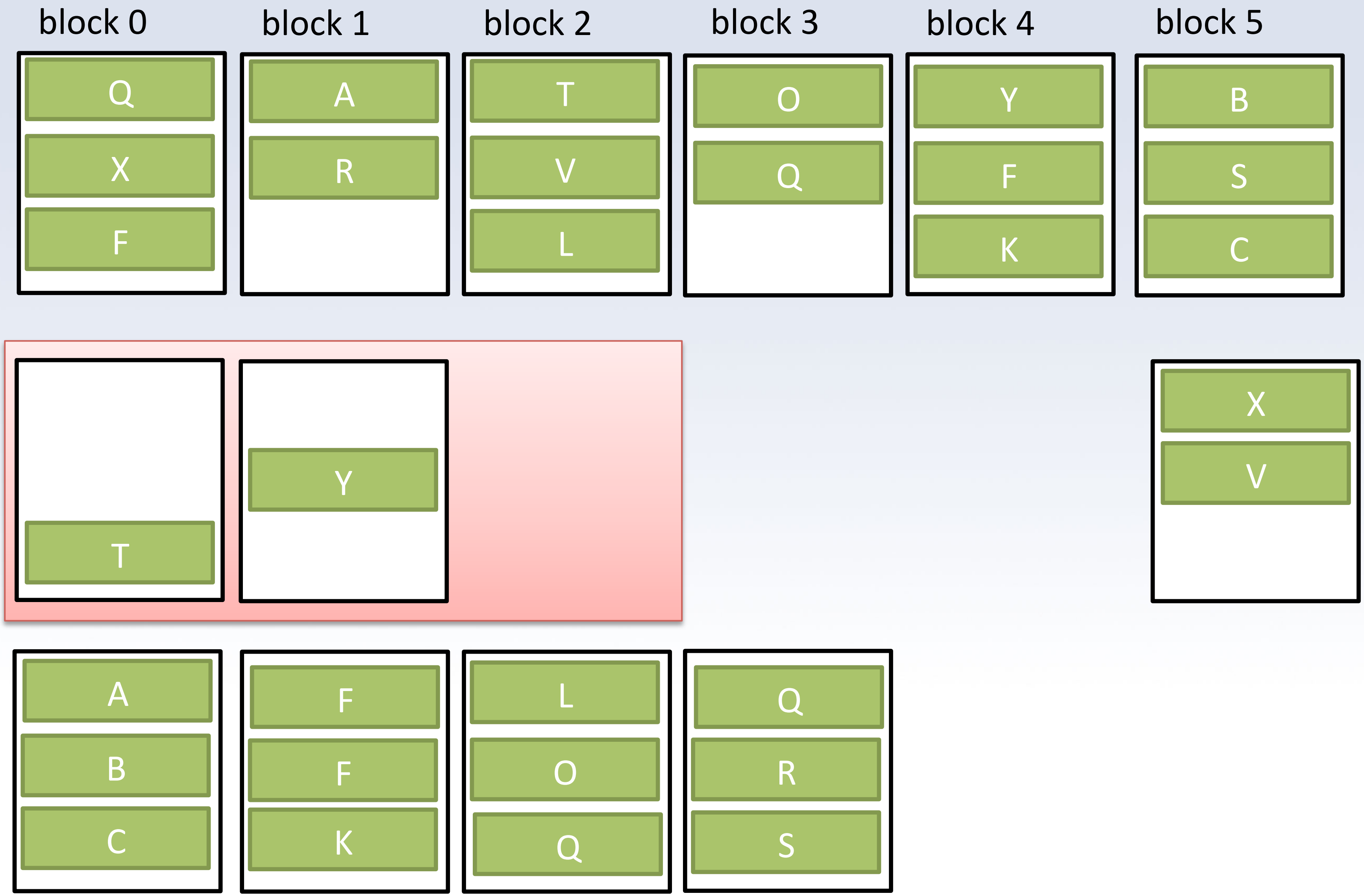
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
T
S
Y
Q
R
X
V
A
B
C
F
F
K
L
O
Q

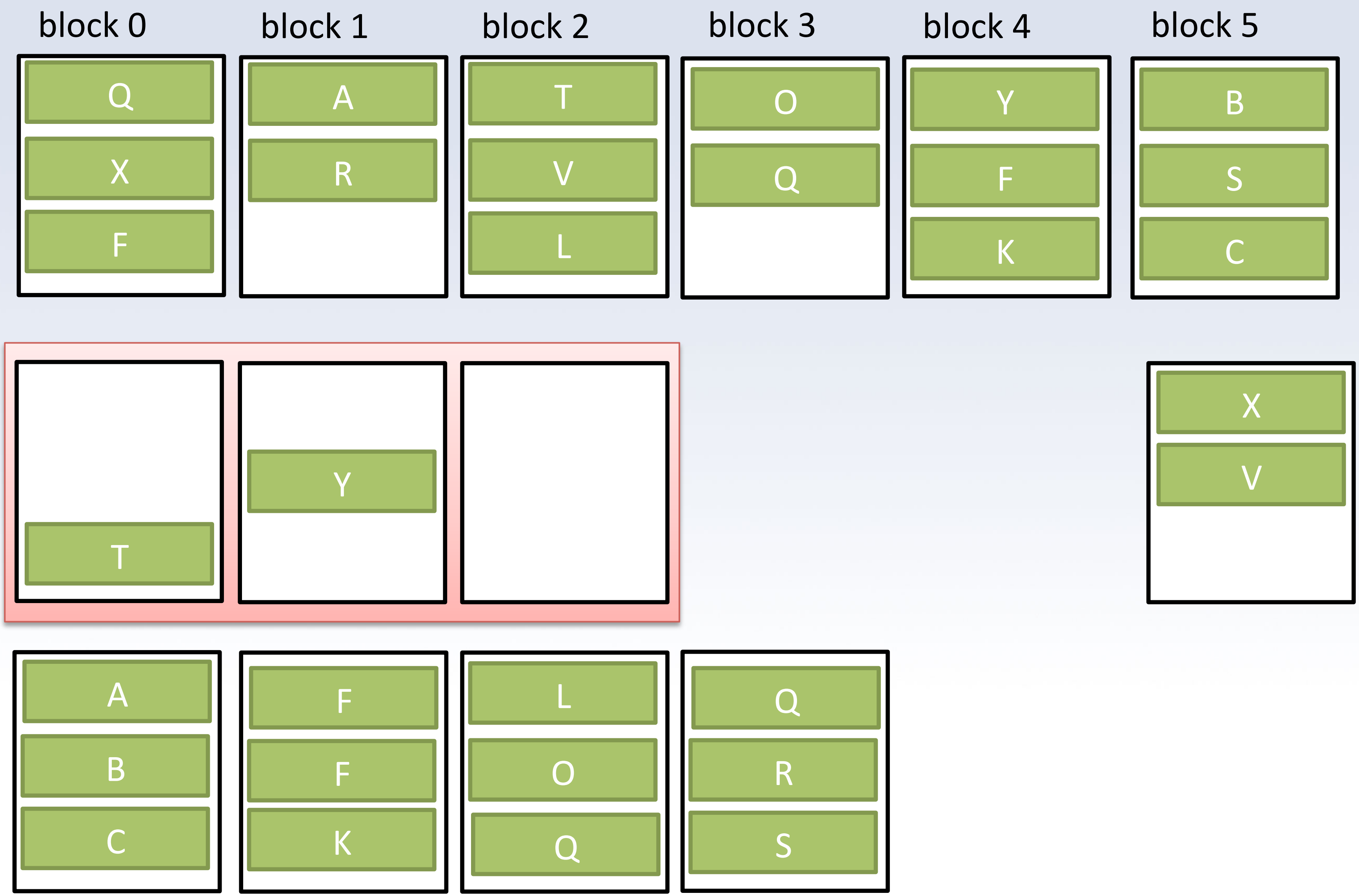
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
T
Y
Q
R
S
X
V
A
B
C
F
F
K
L
O
Q

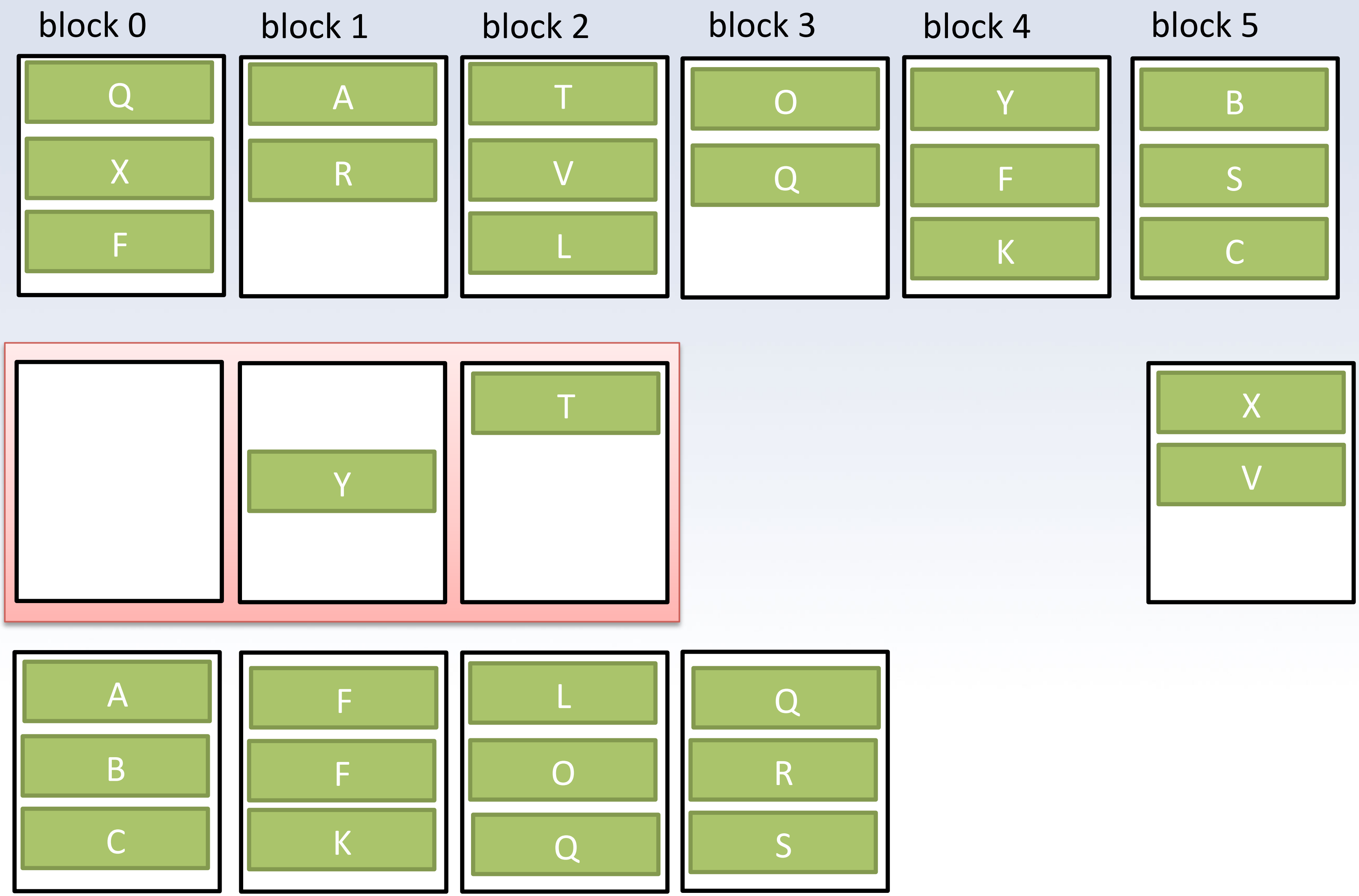
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
T
Y
X
V
A
B
C
F
F
K
L
O
Q
Q
R
S

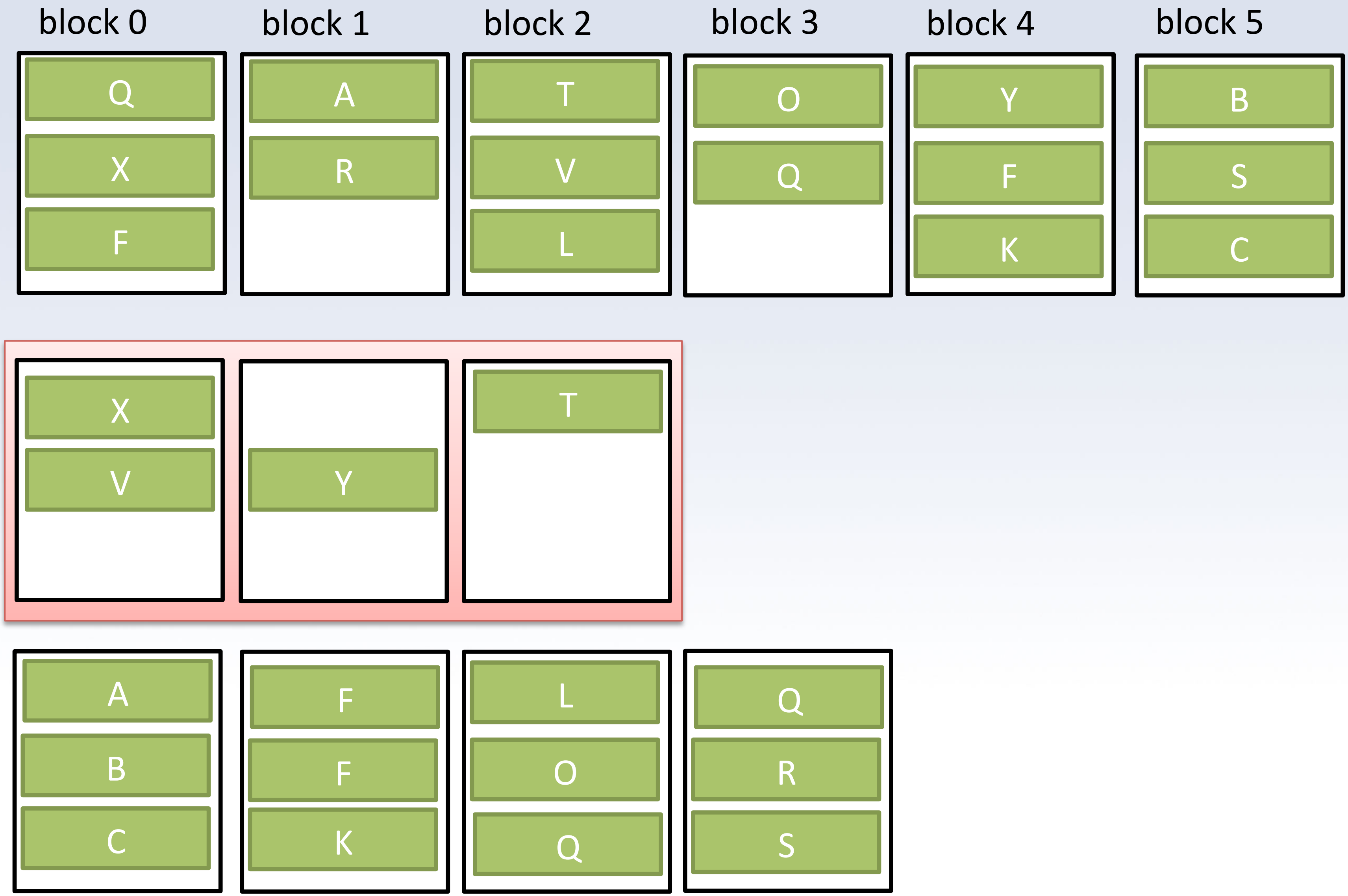
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
T
Y
X
V
A
B
C
F
F
K
L
O
Q
Q
R
S

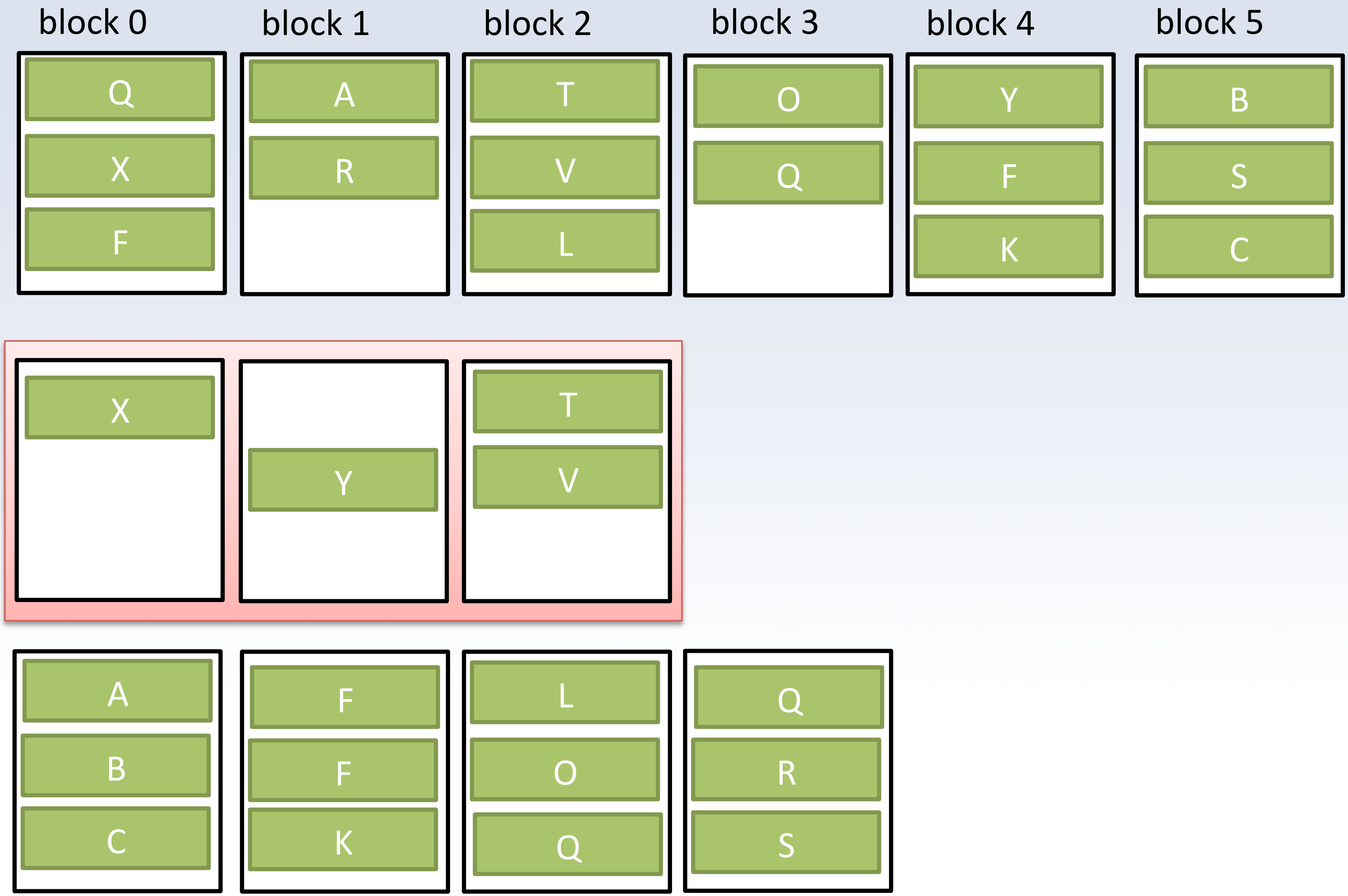
Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Y
T
X
V
A
B
C
F
F
K
L
O
Q
Q
R
S

Phase 2

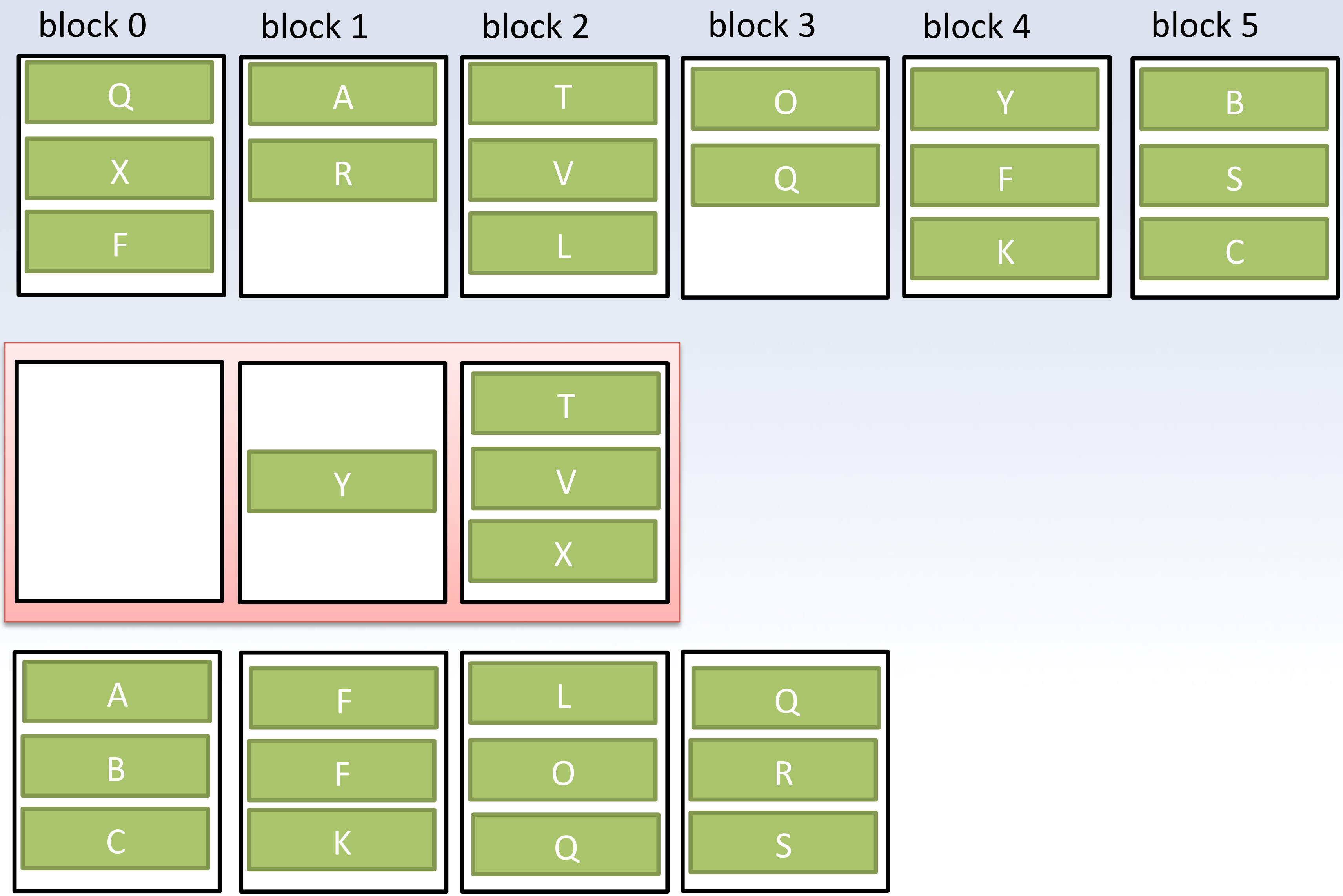

block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
X
V
Y
T
A
B
C
F
F
K
L
O
Q
Q
R
S

Phase 2

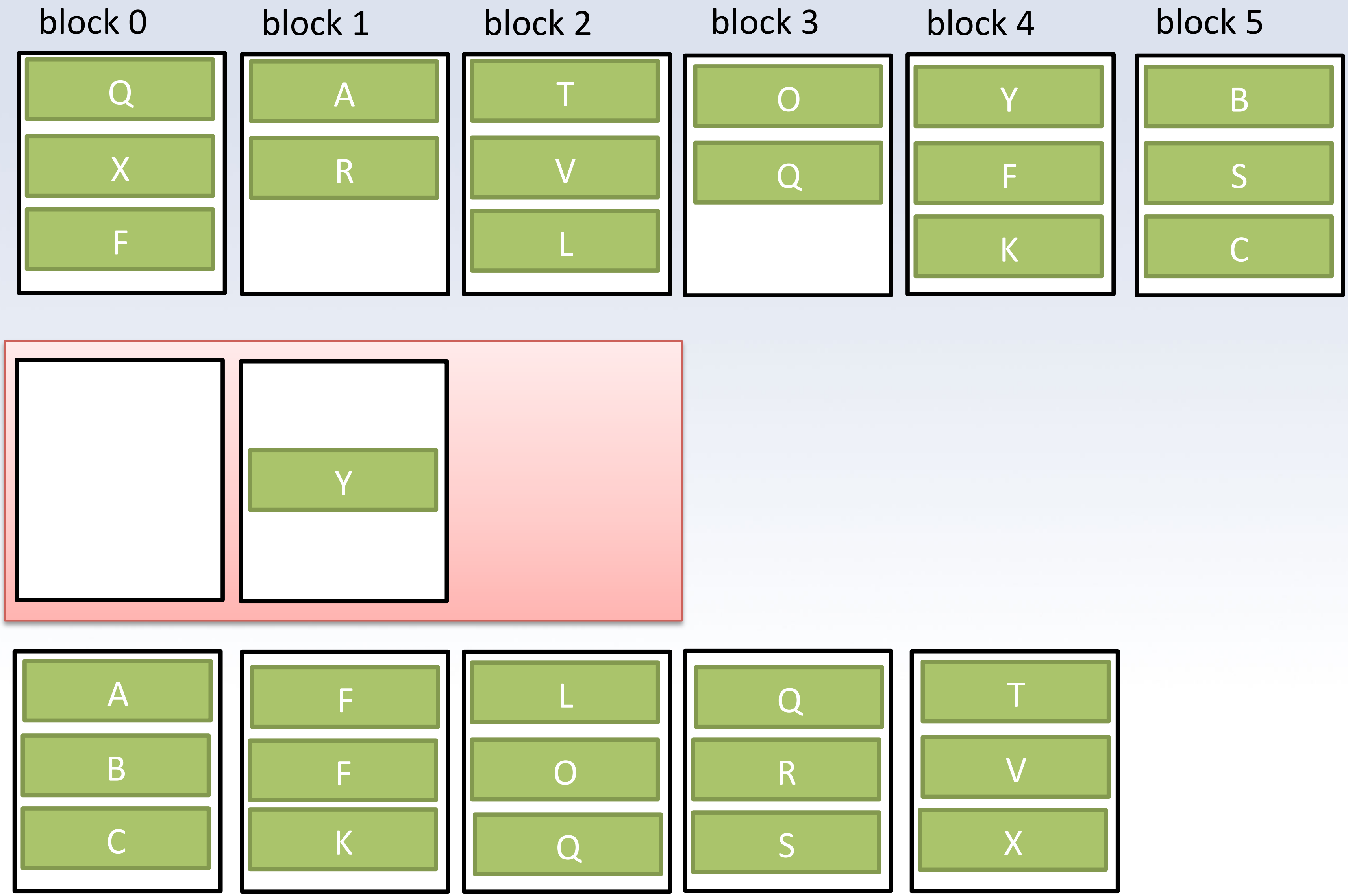


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
X
Y
T
V
A
B
C
F
F
K
L
O
Q
Q
R
S

Phase 2

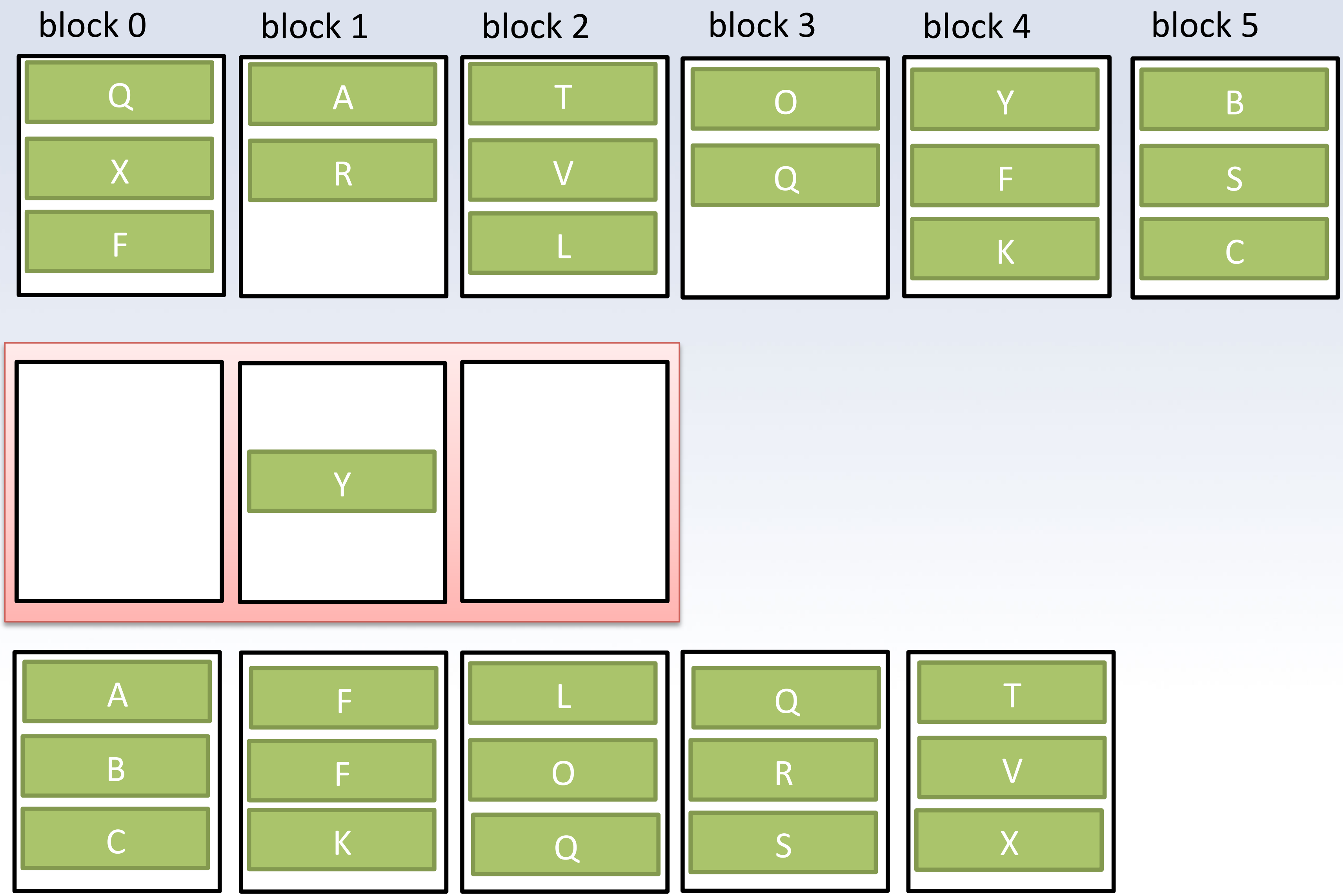

block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Y
T
V
X
A
B
C
F
F
K
L
O
Q
Q
R
S

Phase 2



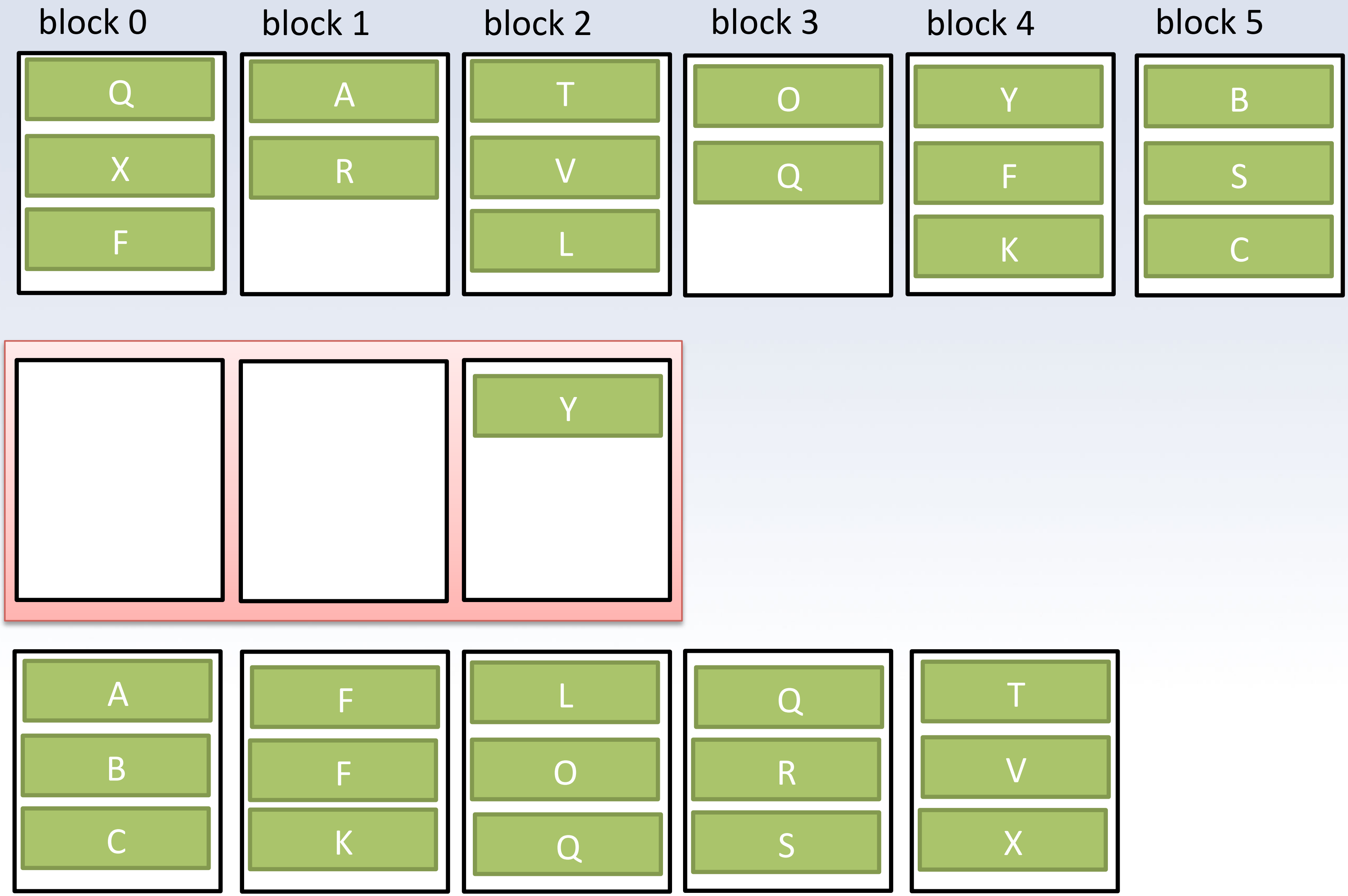
block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Y
A
B
C
F
F
K
L
O
Q
Q
R
S
T
V
X

Phase 2



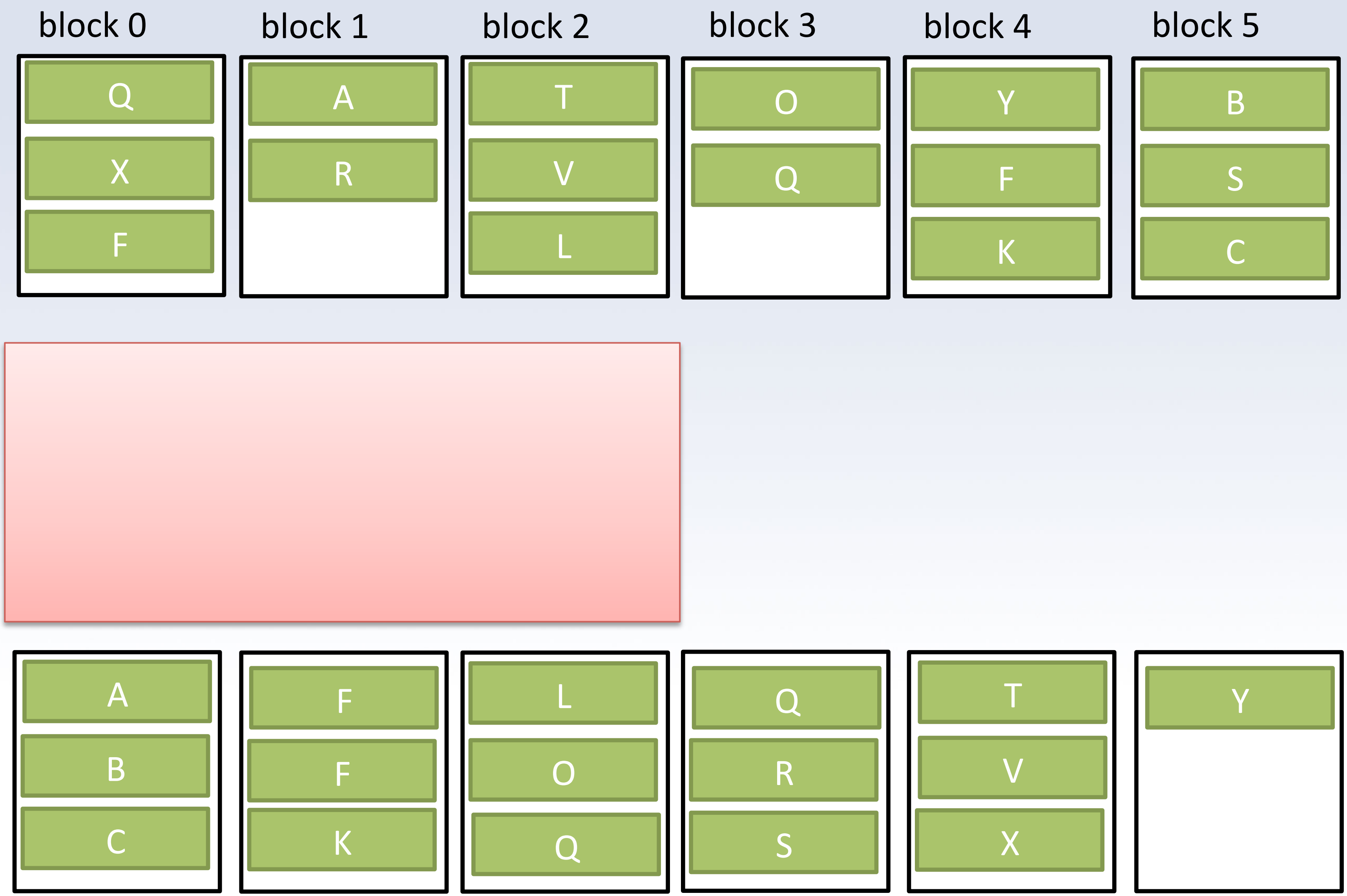
block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Y
A
B
C
F
F
K
L
O
Q
Q
R
S
T
V
X

Phase 2



block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
Y
A
B
C
F
F
K
L
O
Q
Q
R
S
T
V
X

Phase 2


block 0
block 1
block 2
block 3
block 4
block 5
Q
X
F
A
R
T
V
L
O
Q
Y
F
K
B
S
C
A
B
C
F
F
K
L
O
Q
Q
R
S
T
V
X
Y

Analysis

- Number of sublists is B/M
- To fit one block for each sublist, $B/M \leq M-1$
- Rewritten, $B \leq M(M-1)$
- Requires that $\sqrt{B(R)} \leq M$

Analysis

- Read B(R) blocks in Phase 1
- Write B(R) blocks in Phase 1
- Read B(R) blocks in Phase 2
- Write B(R) blocks in Phase 2

Analysis

- Read B(R) blocks in Phase 1
- Write B(R) blocks in Phase 1
- Read B(R) blocks in Phase 2
- ~~Write B(R) blocks in Phase 2~~ Output

Analysis

- Read B(R) blocks in Phase 1
- Write B(R) blocks in Phase 1
- Read B(R) blocks in Phase 2
- ~~Write B(R) blocks in Phase 2~~ Output
- Cost is therefore 3B(R)

Sorting-based δ

- We can implement δ operation with our TPMMS as follows:
  1. Create all of the sorted sublists as usual
  2. In phase 2, don't output duplicate entries (keep track of last record output and skip identical records)

Sorting-based γ

- We can implement γ operation with our TPMMS as follows:
  1. Create all of the sorted sublists as usual
  2. In phase 2, don't output duplicate entries. Instead, compute aggregation functions on duplicate entries

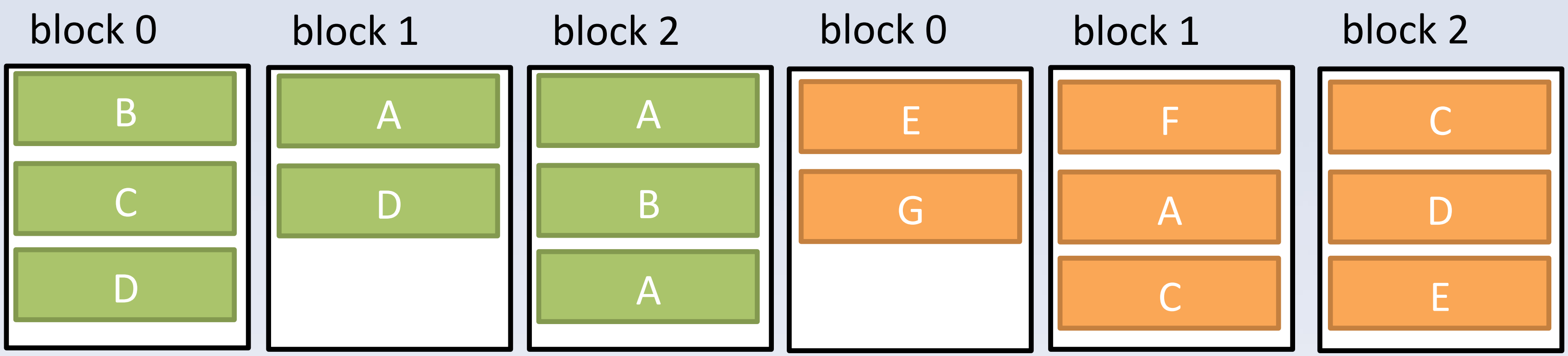
Sorting-based ∪

- We can implement ∪ operation with our TPMMS as follows:
  1. Create all of the sorted sublists for both R *and* S
  2. In phase 2, bring sorted lists for BOTH relations into buffers. Don't output duplicate entries.
- Essentially, treat R and S as one relation

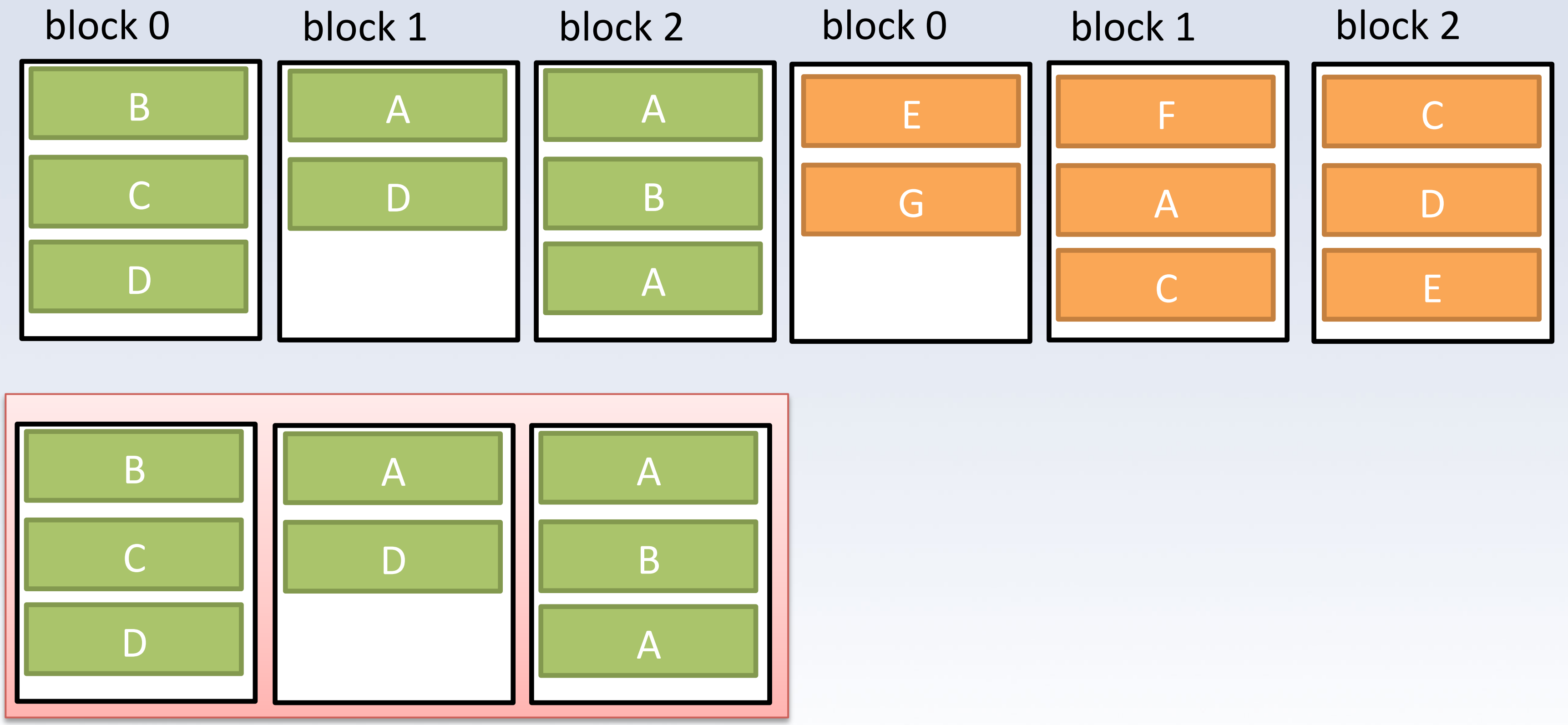
Sorting-based ∩

- We can implement ∩ operation with our TPMMS as follows:
  1. Create all of the sorted sublists for both R *and* S
  2. In phase 2, bring sorted lists for both relations into buffers, but keep them separate
  3. Only output when both relations contain the same value

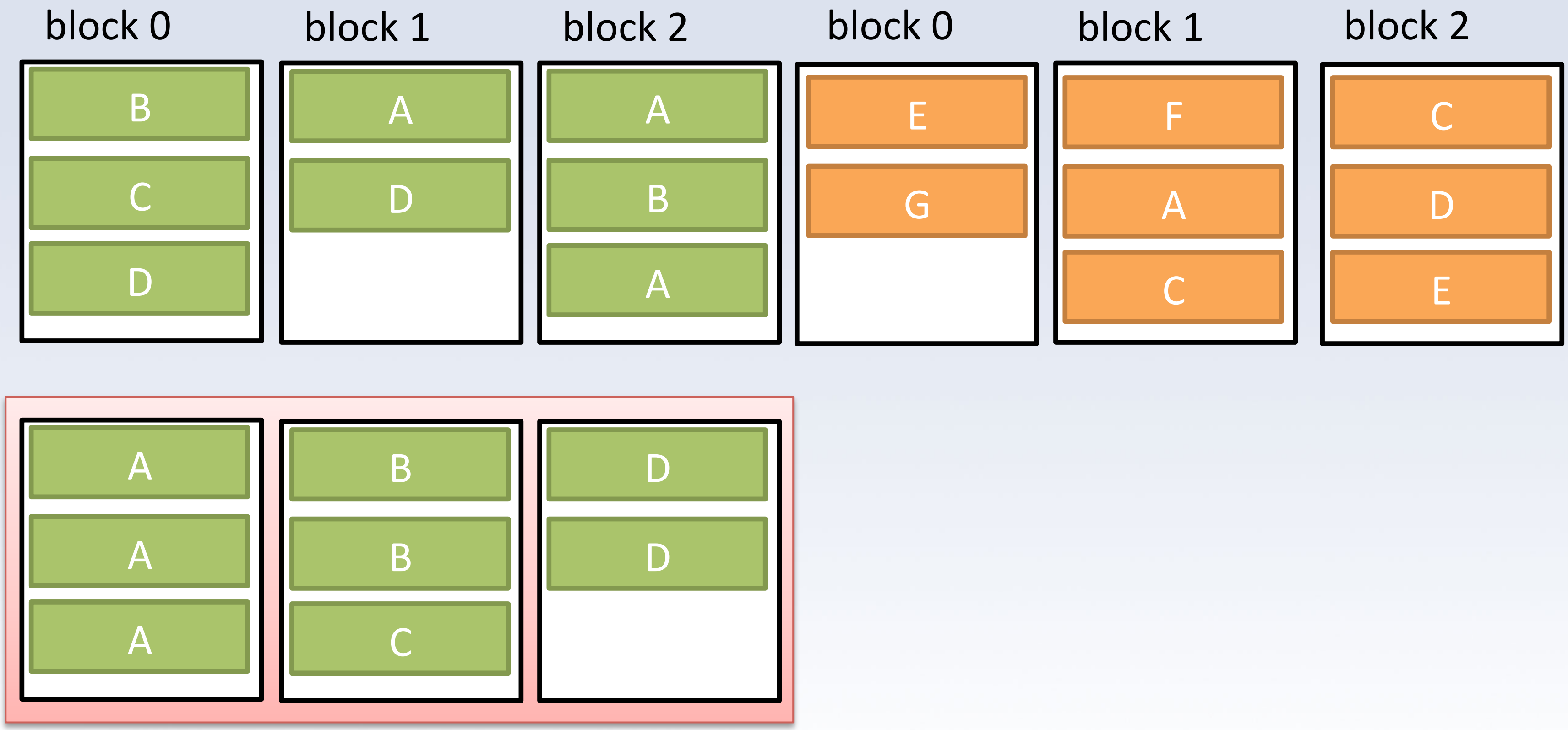
Example R ∩ S


block 0
B
C
D
block 1
A
D
block 2
A
B
A
block 0
E
G
block 1
F
A
C
block 2
C
D
E

Example R ∩ S

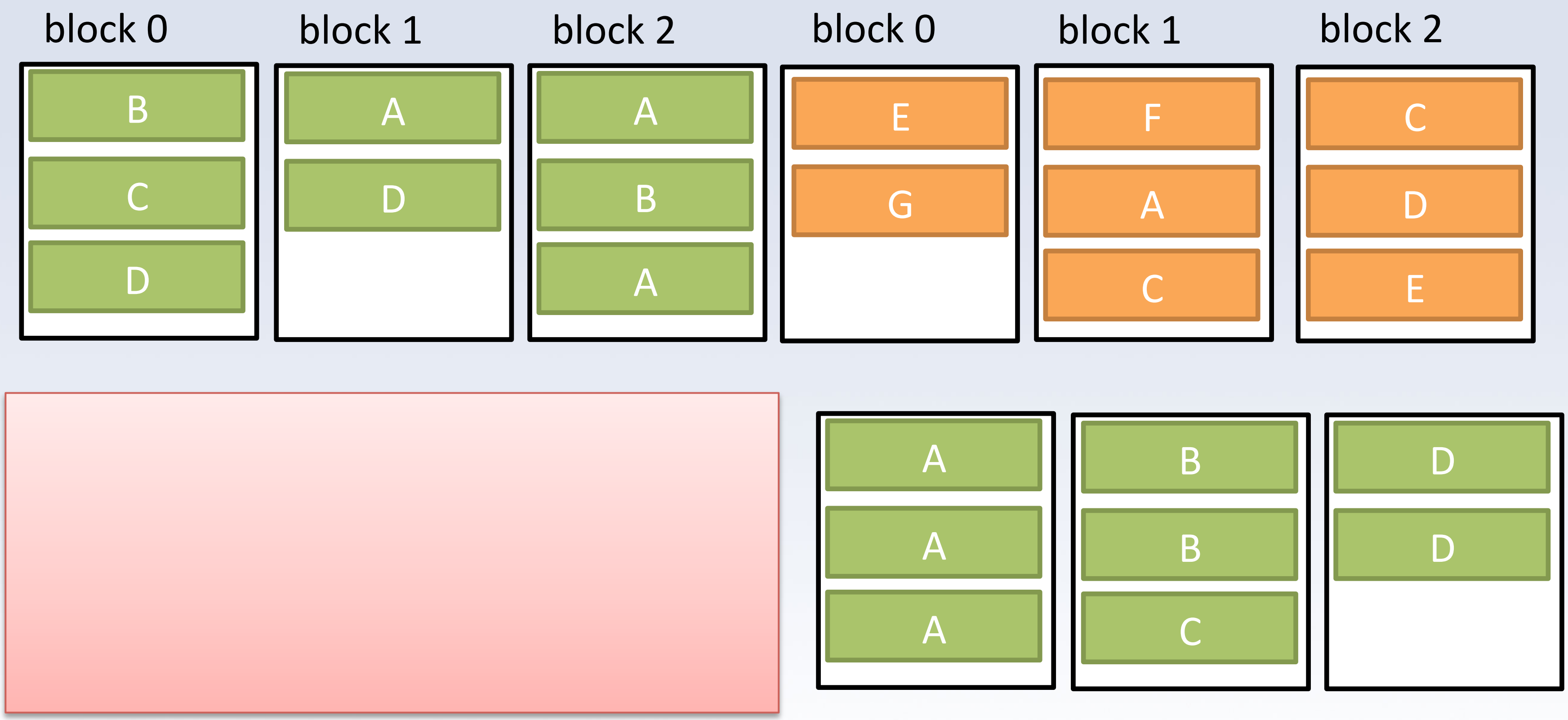

block 0
B
C
D
block 1
A
D
block 2
A
B
A
block 0
E
G
block 1
F
A
C
block 2
C
D
E
B
C
D
A
D
A
B
A

Example R ∩ S



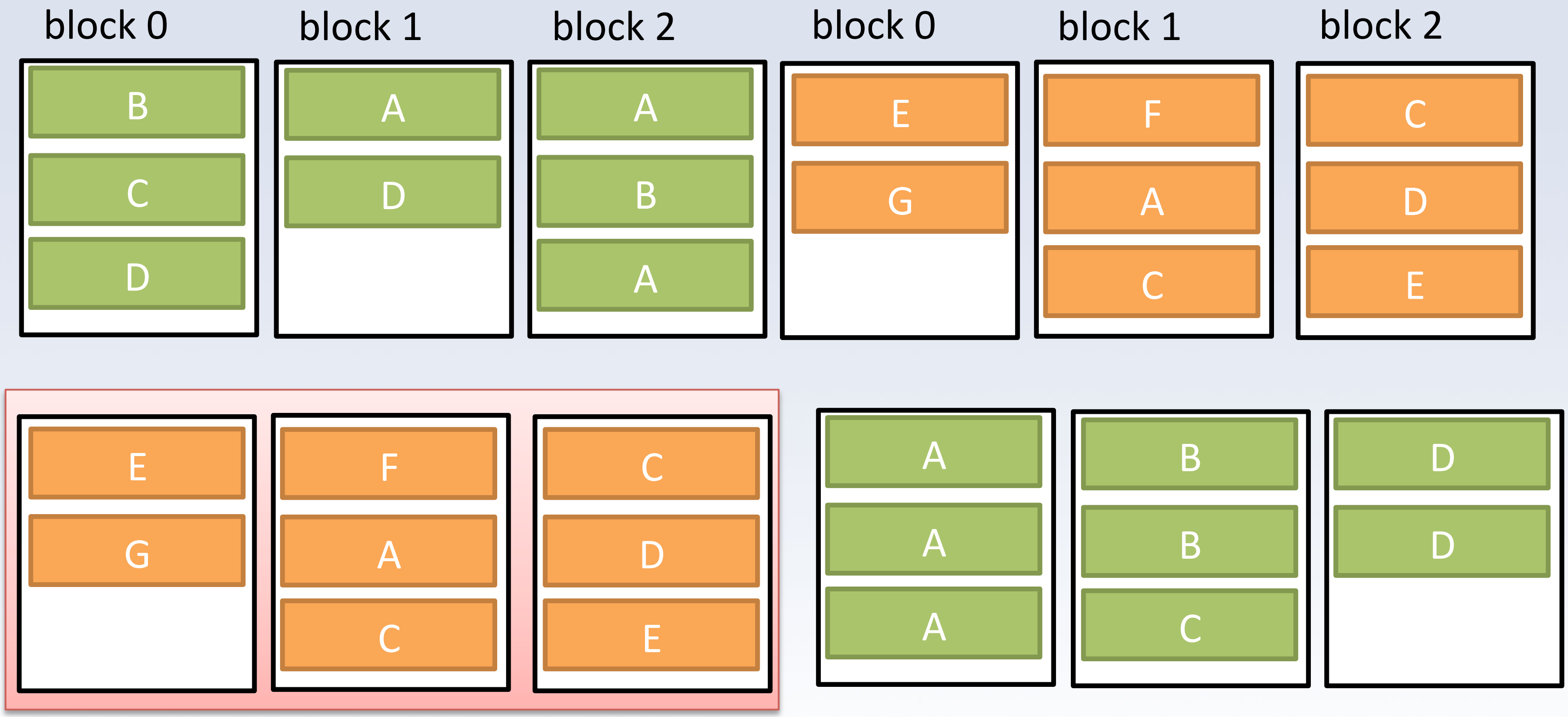
block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
A
A
A
B
B
C
D
D

Example R ∩ S



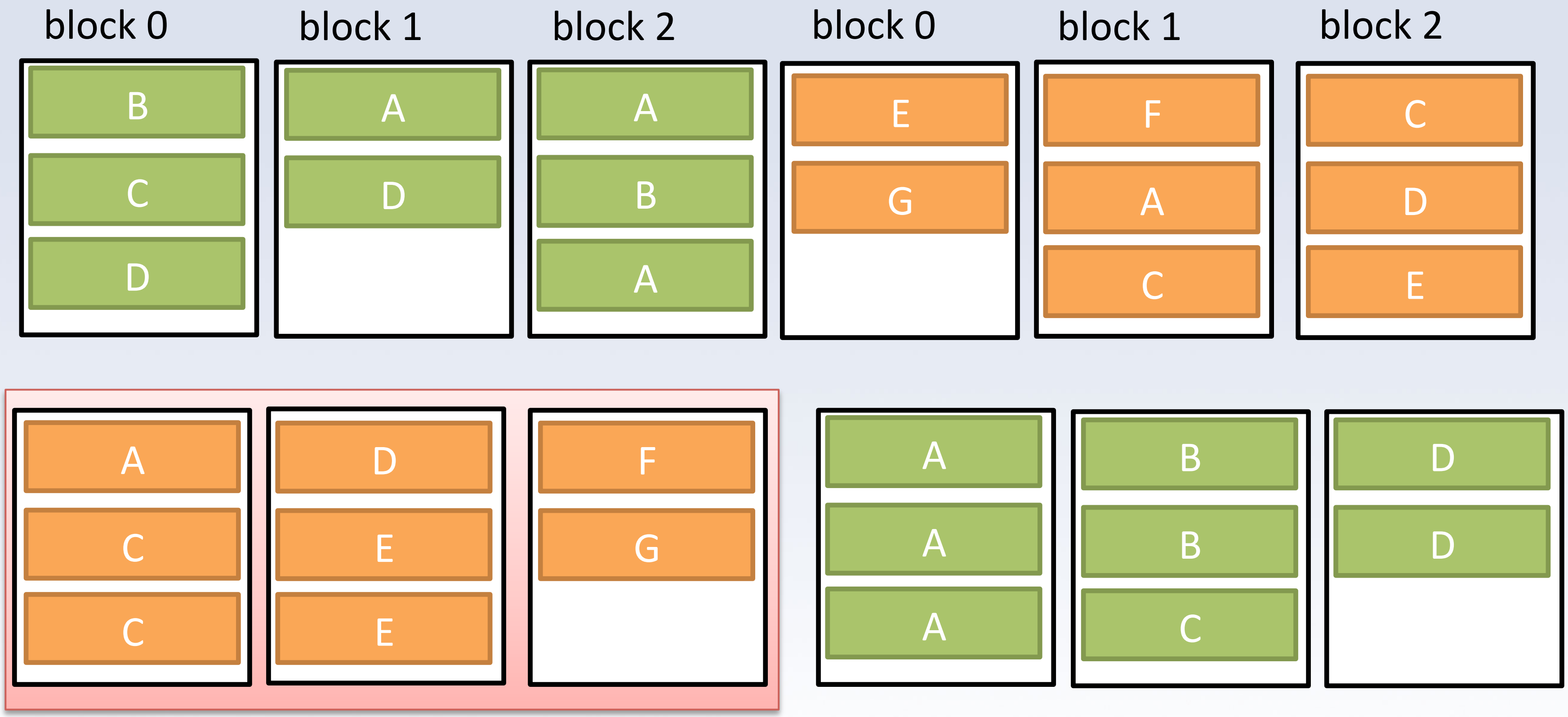
illinois.edu

Example R ∩ S

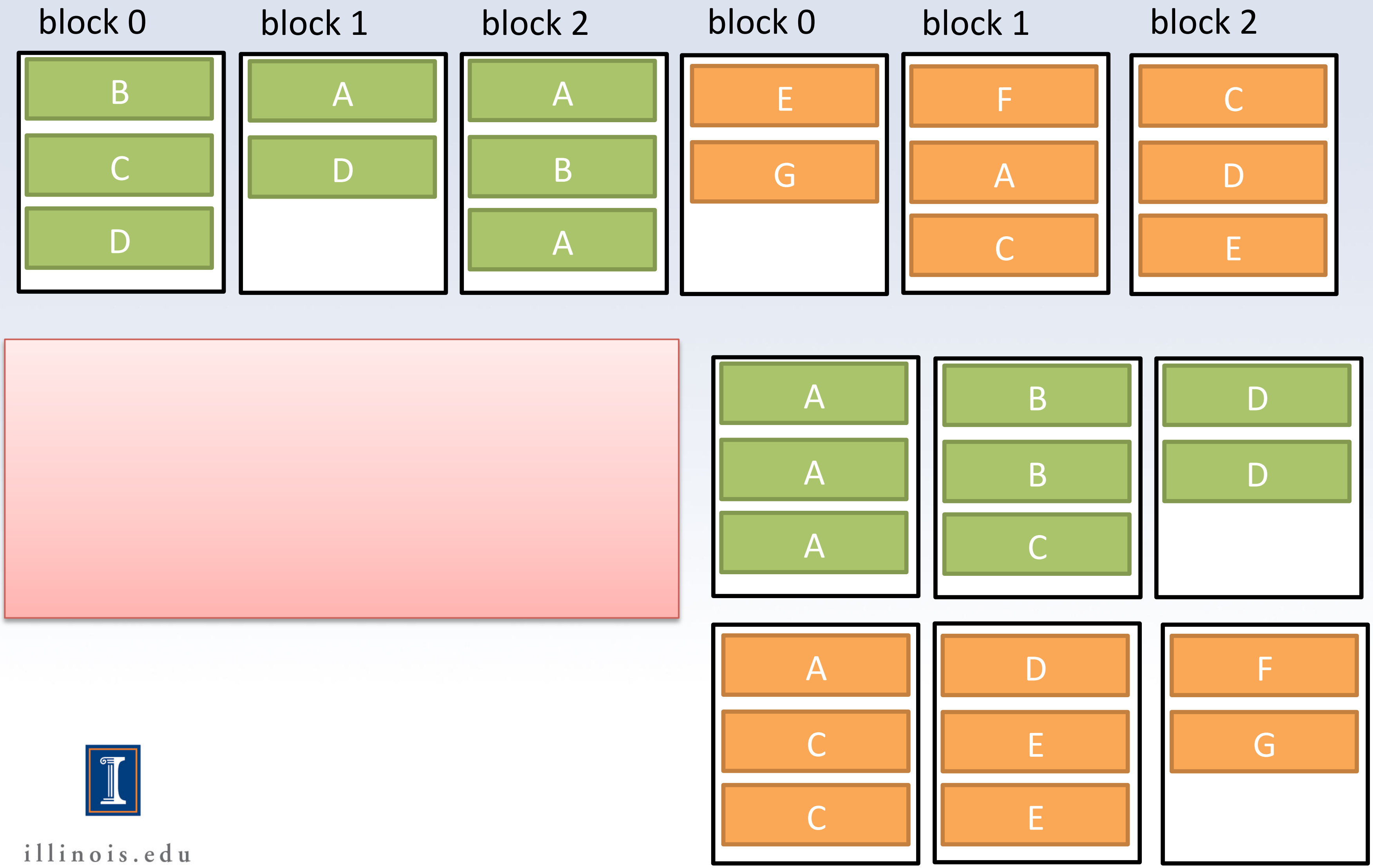


illinois.edu

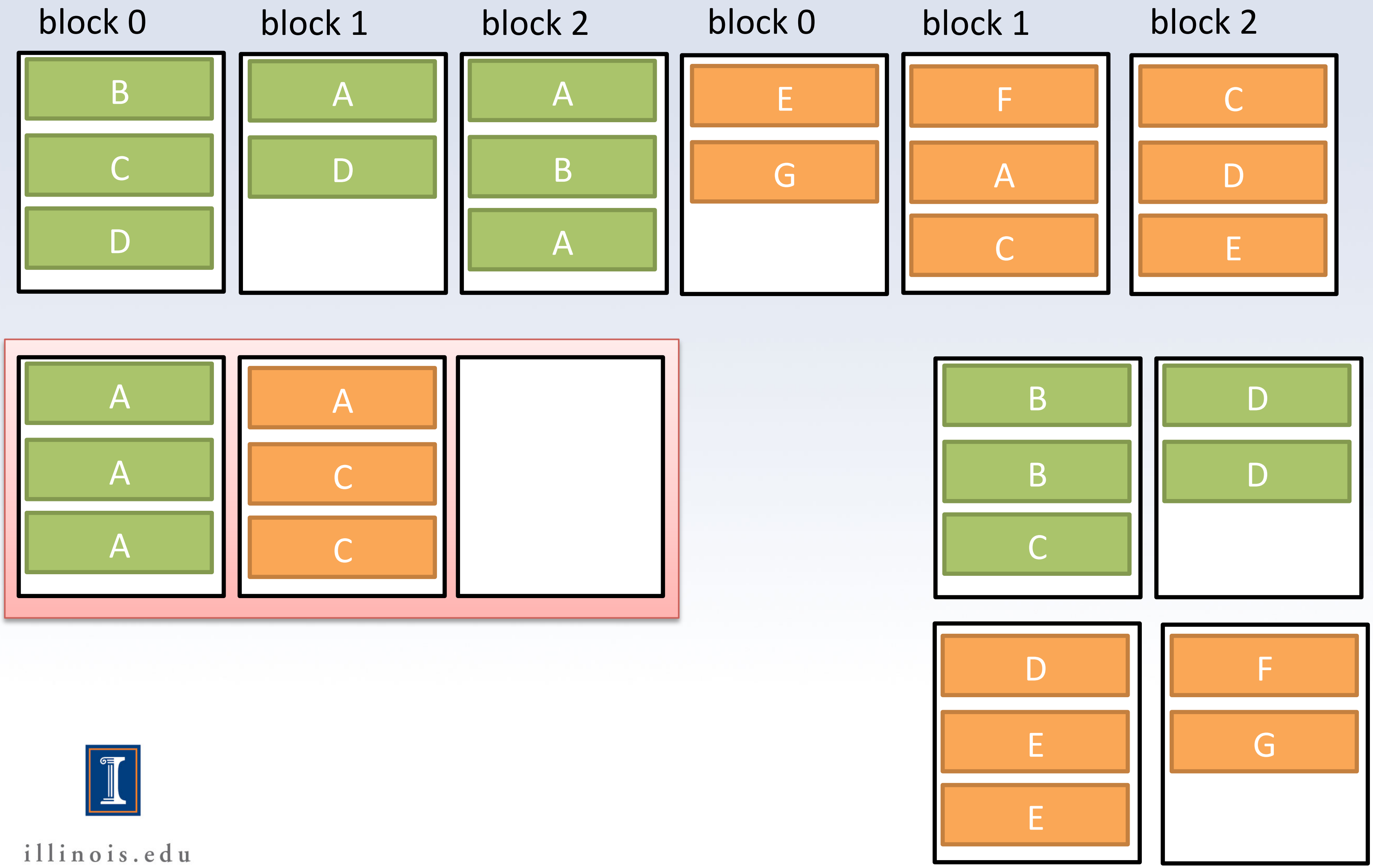
Example R ∩ S


block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
A
C
C
D
E
E
F
G
A
A
A
B
B
C
D
D

Example R ∩ S

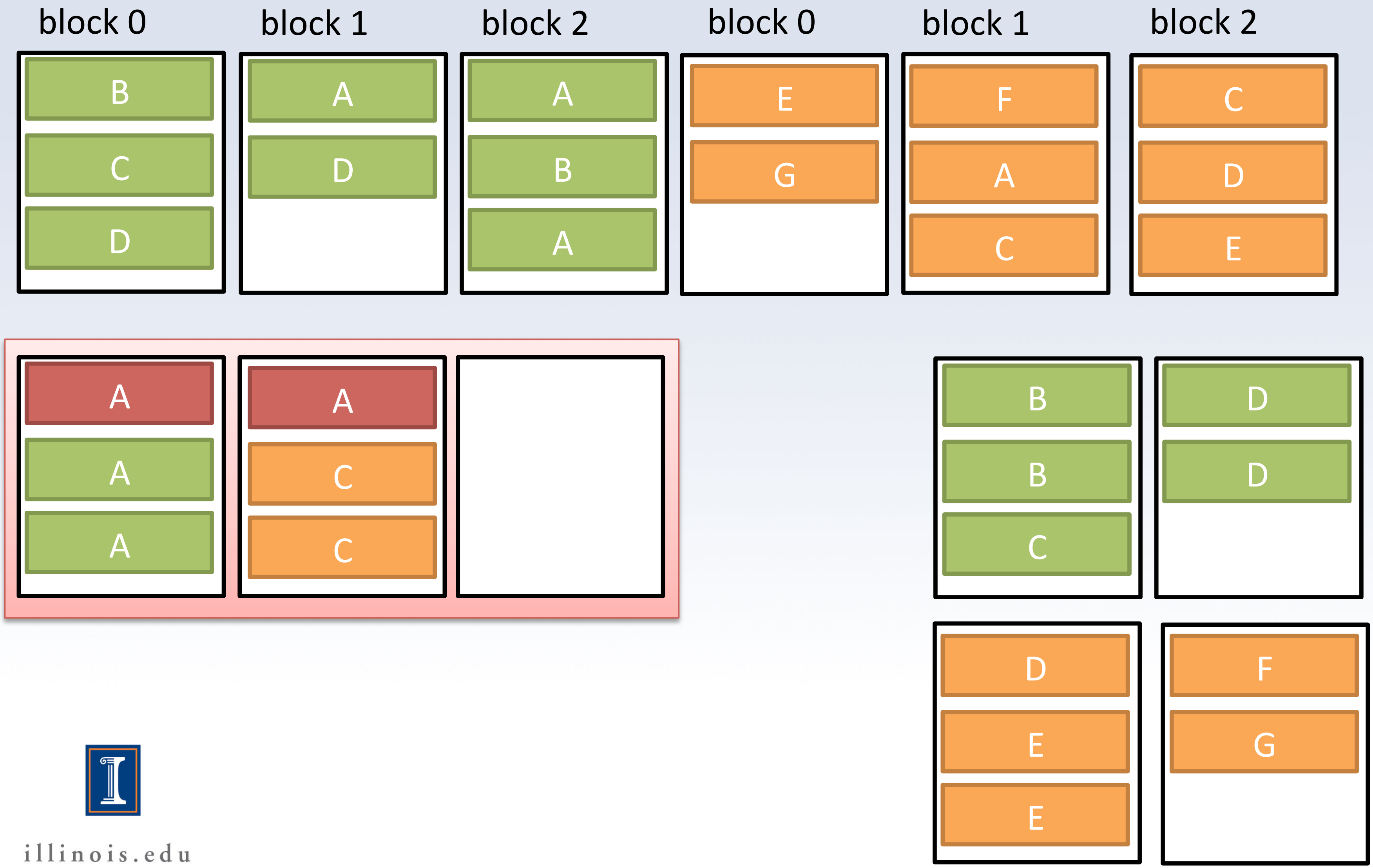

block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
A
A
A
B
B
C
D
D
A
C
C
D
E
E
F
G

Example R∩S



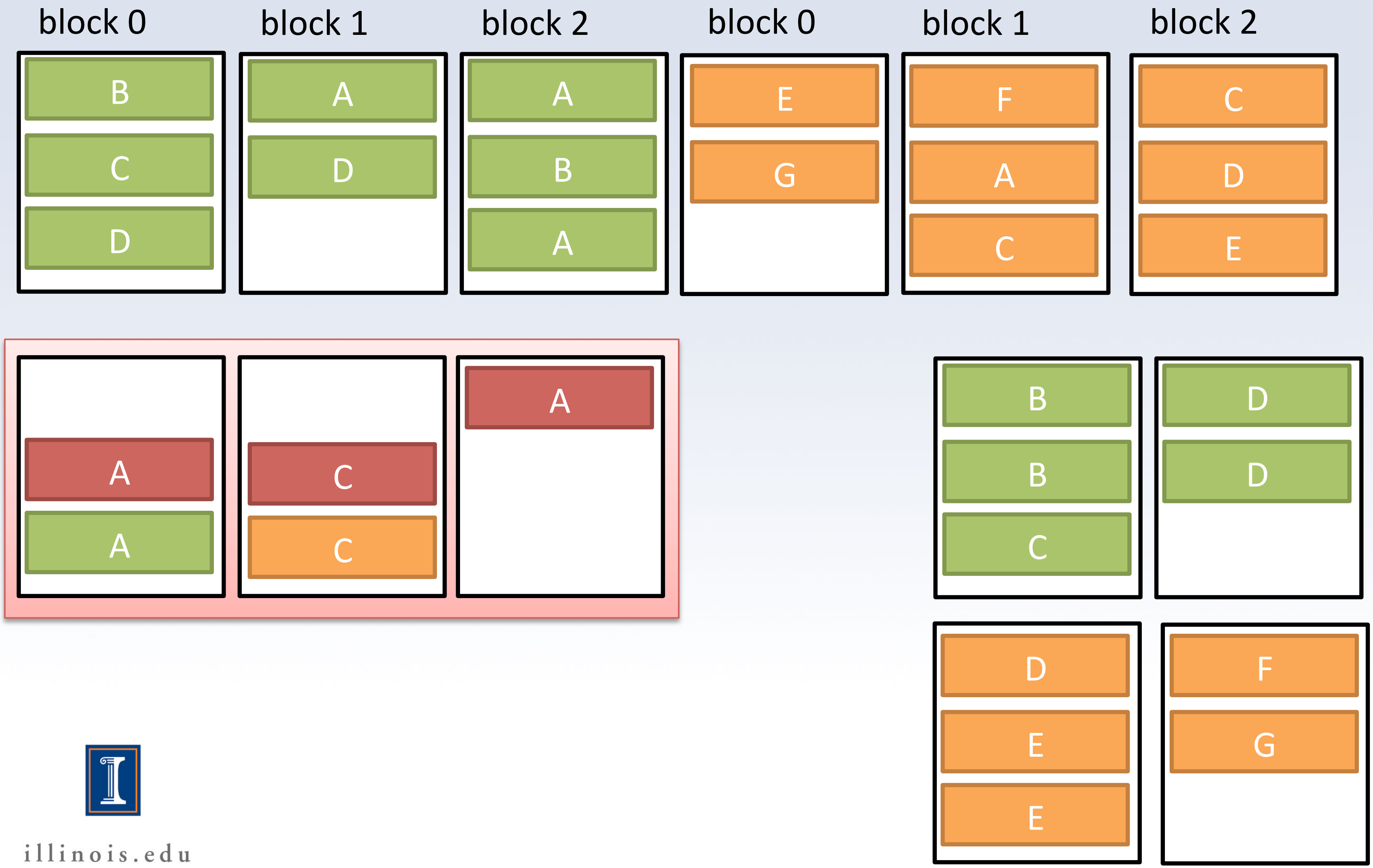
illinois.edu

Example R ∩ S

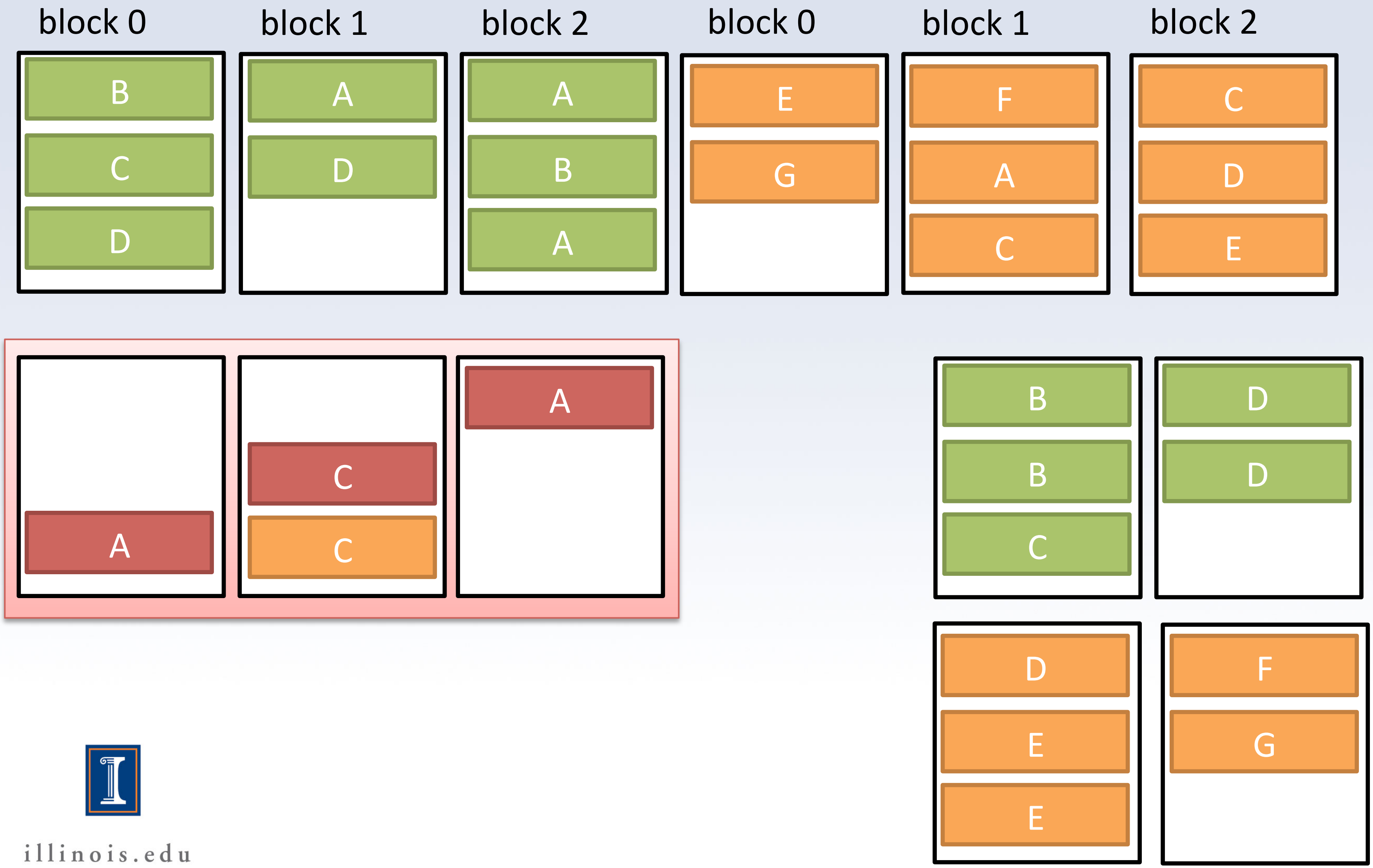


illinois.edu

Example R ∩ S

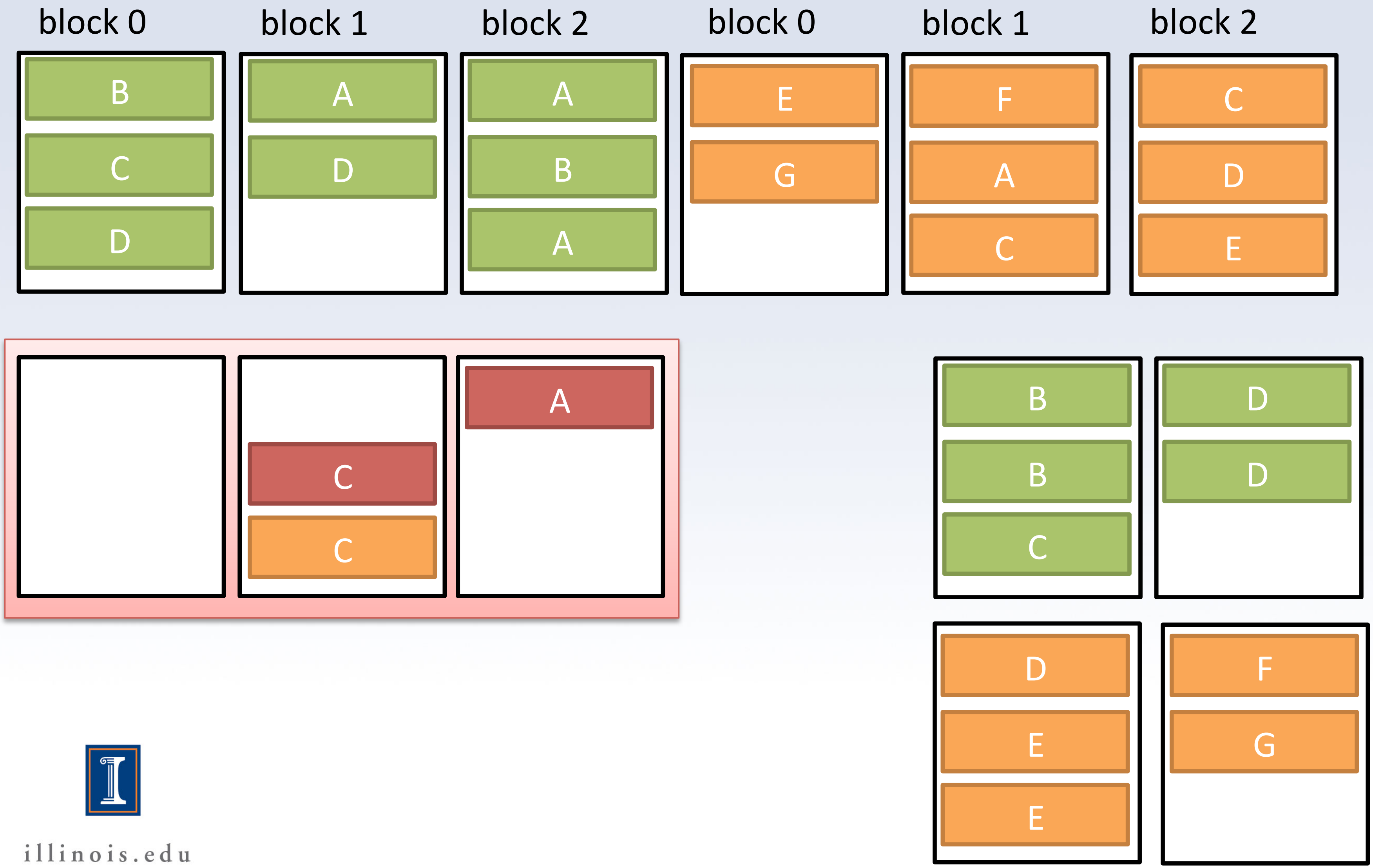

block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
A
A
C
C
A
B
B
C
D
D
D
E
E
F
G

Example R ∩ S



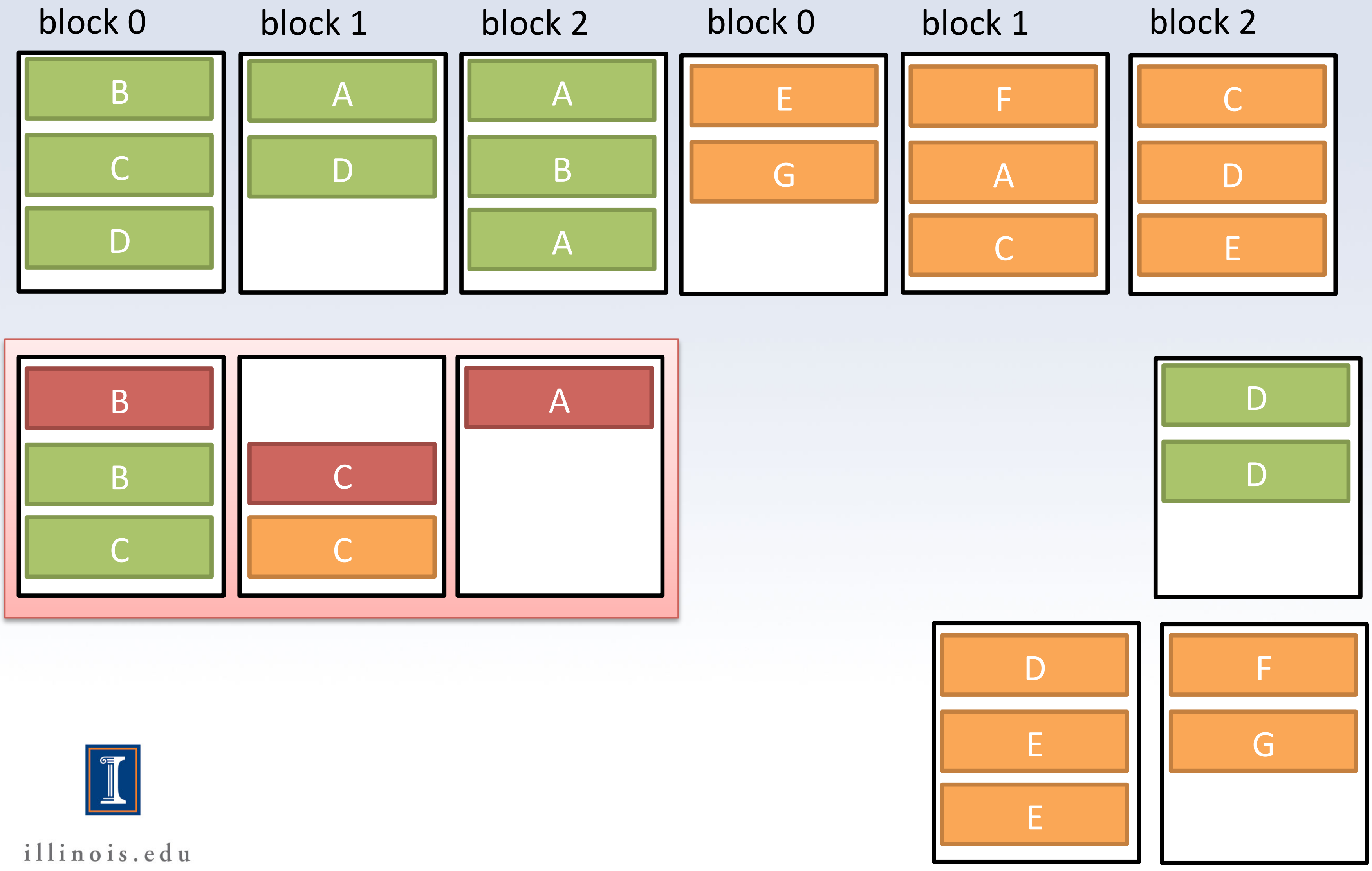
illinois.edu

Example R ∩ S

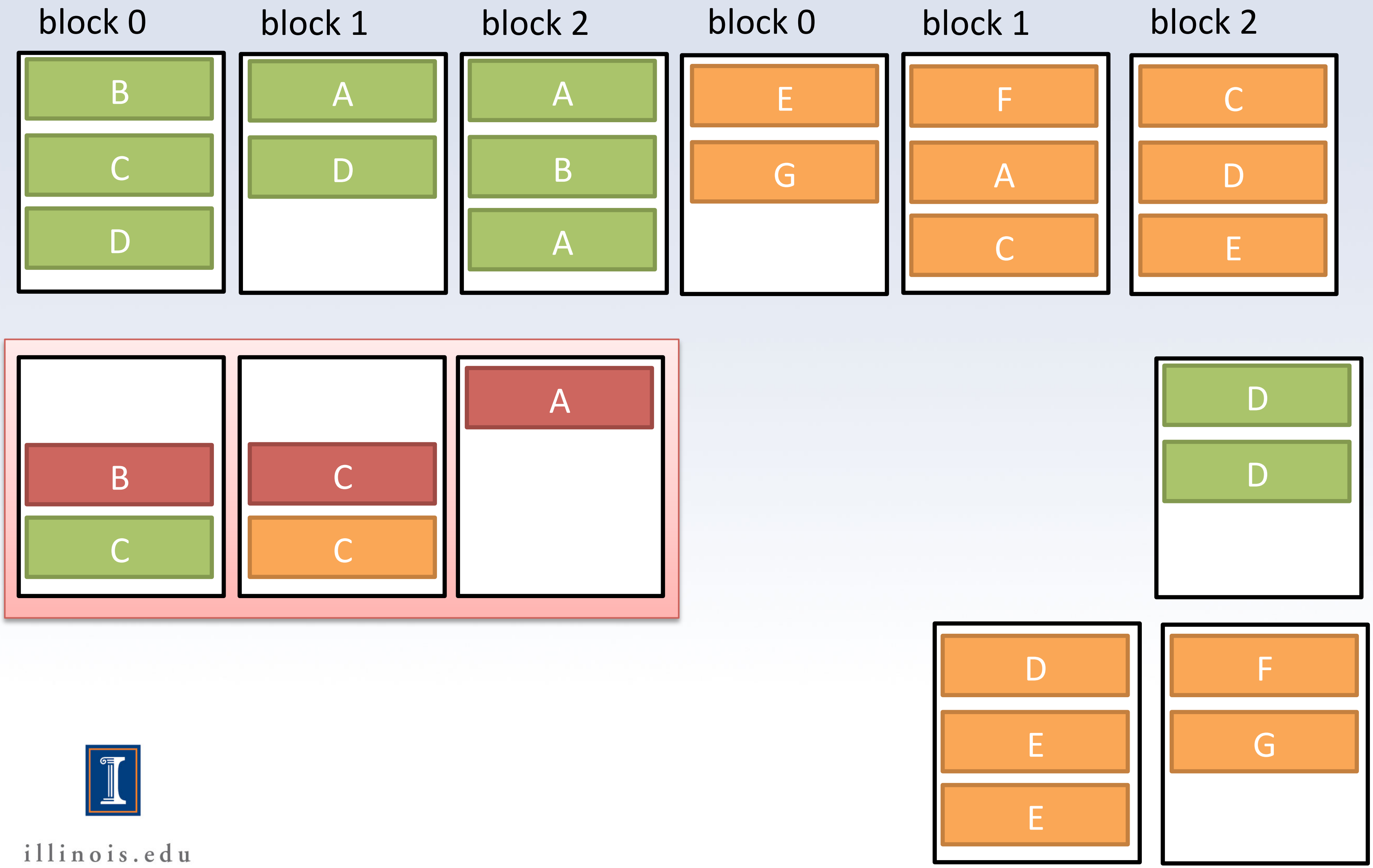


illinois.edu

Example R ∩ S

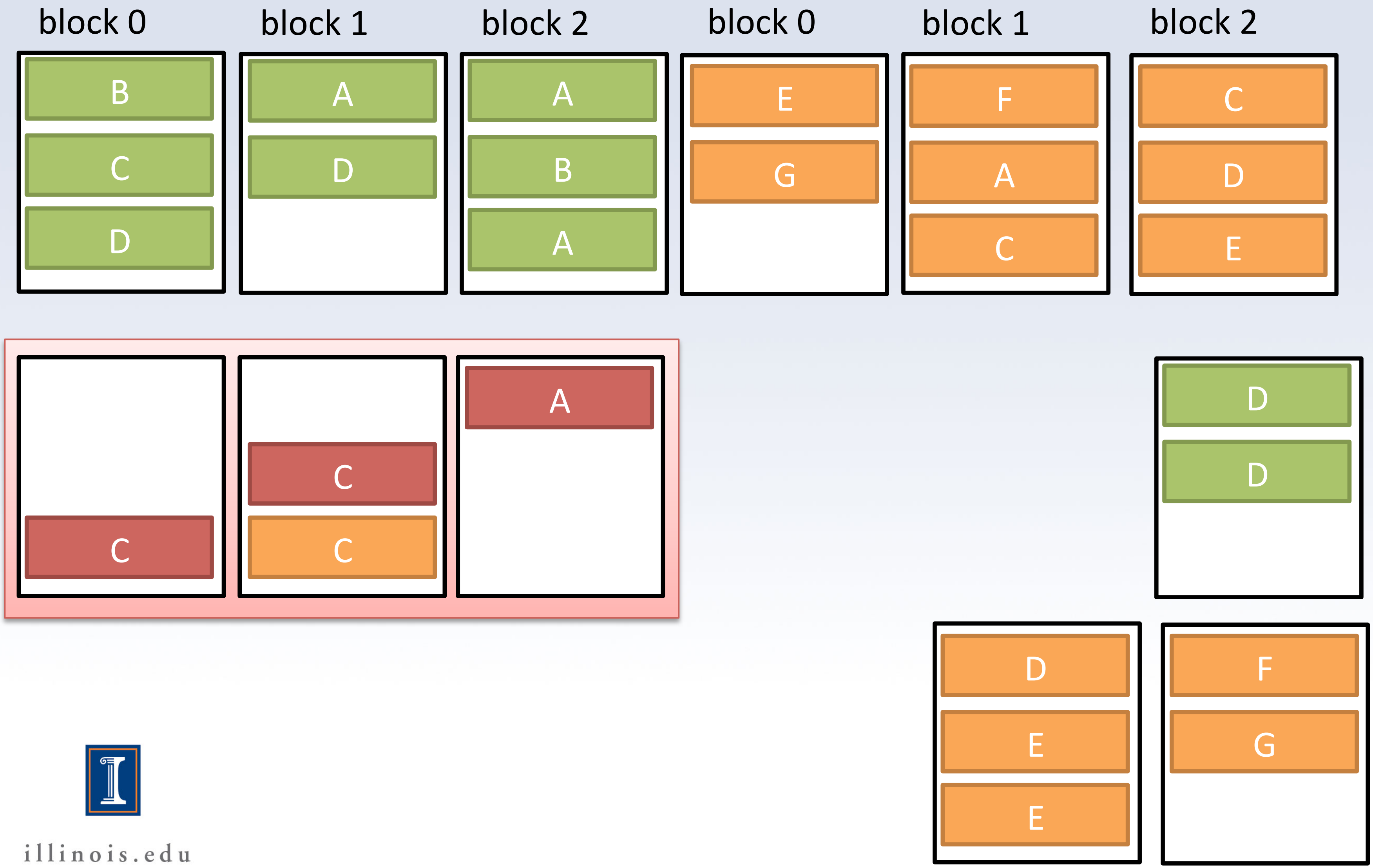

block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
B
B
C
C
C
A
D
D
D
E
E
F
G

Example R ∩ S



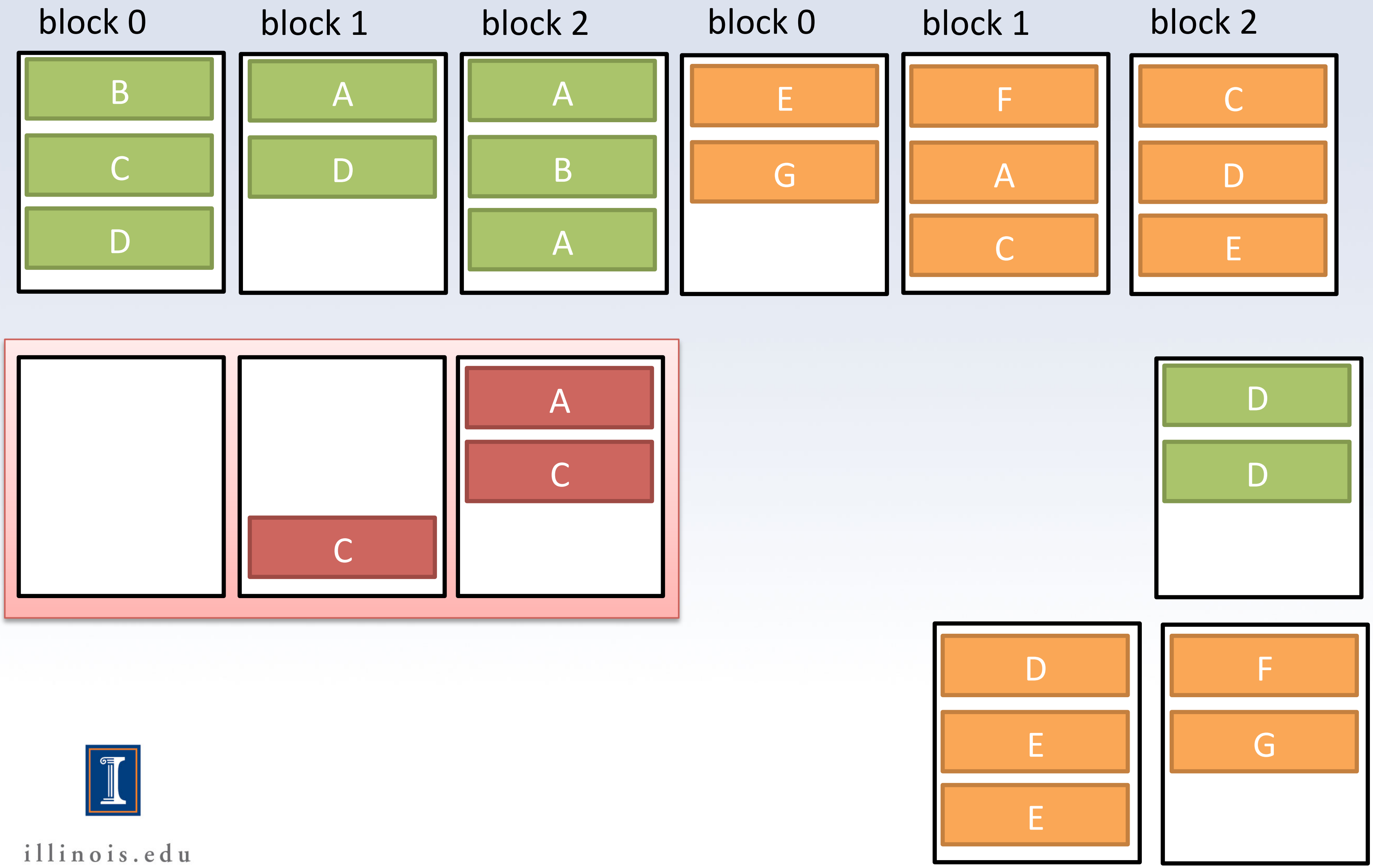
illinois.edu

Example R ∩ S

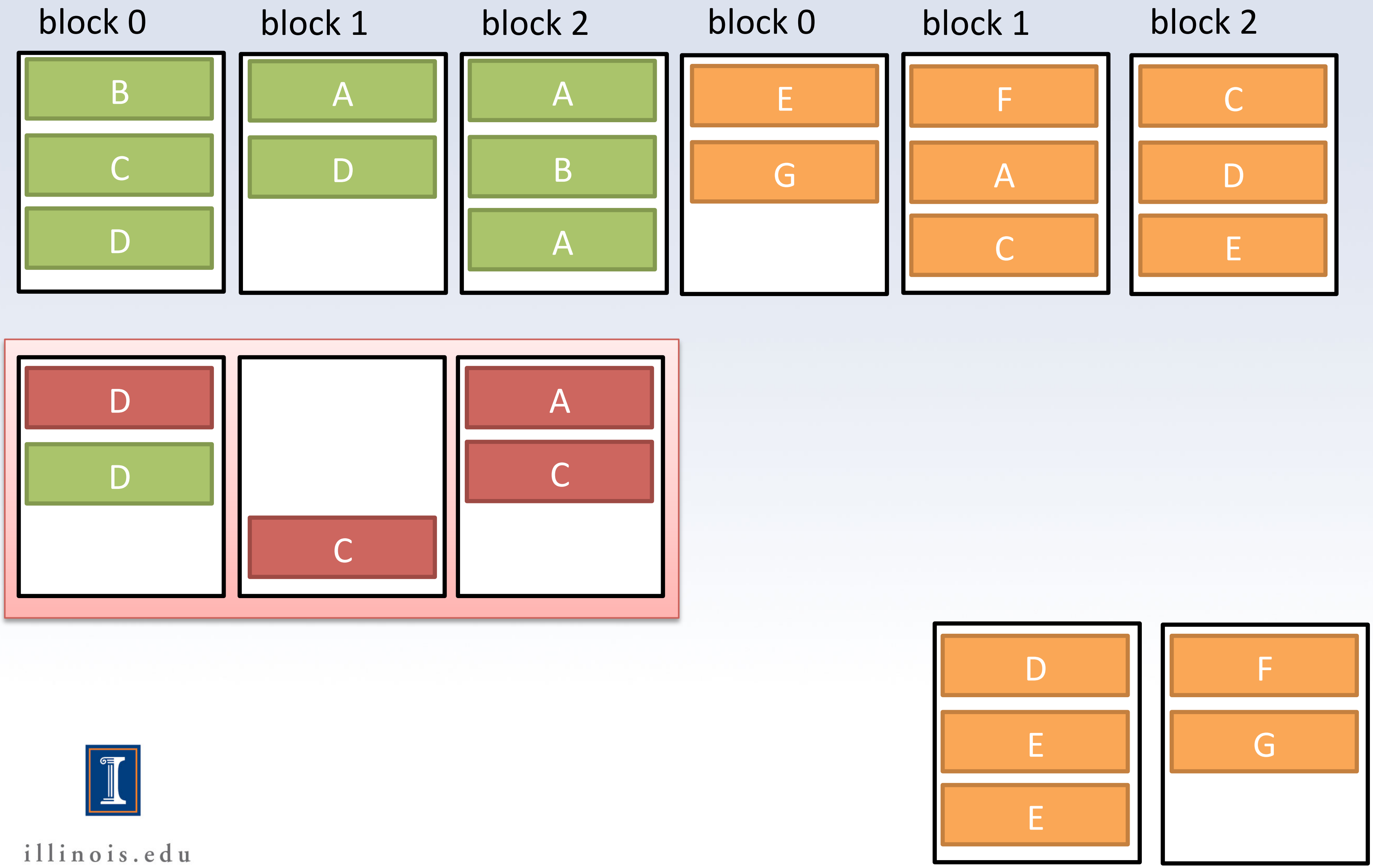


block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
C
C
C
A
D
D
D
E
E
F
G

Example R∩S

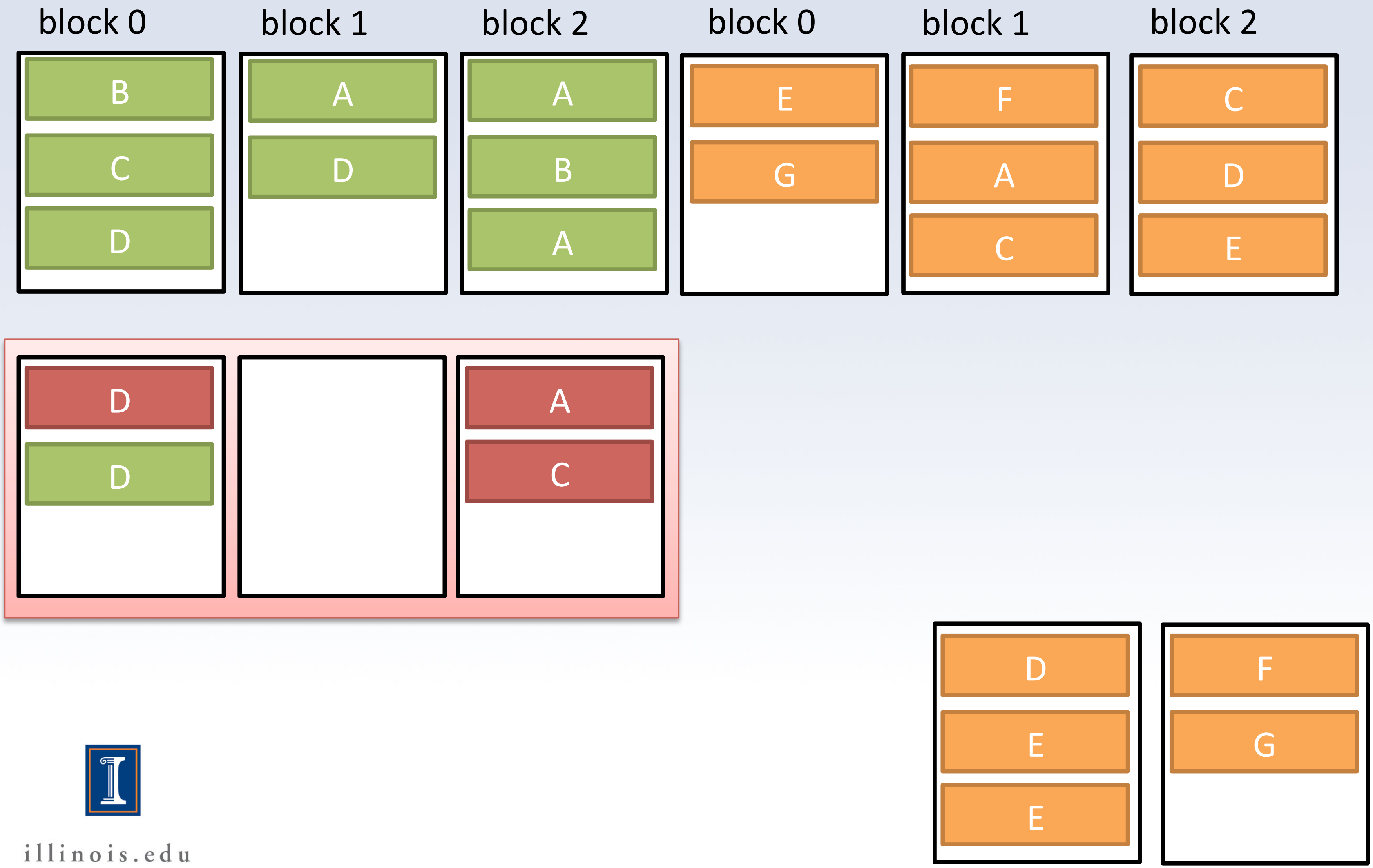

block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
A
C
C
D
D
D
E
E
F
G

Example R ∩ S



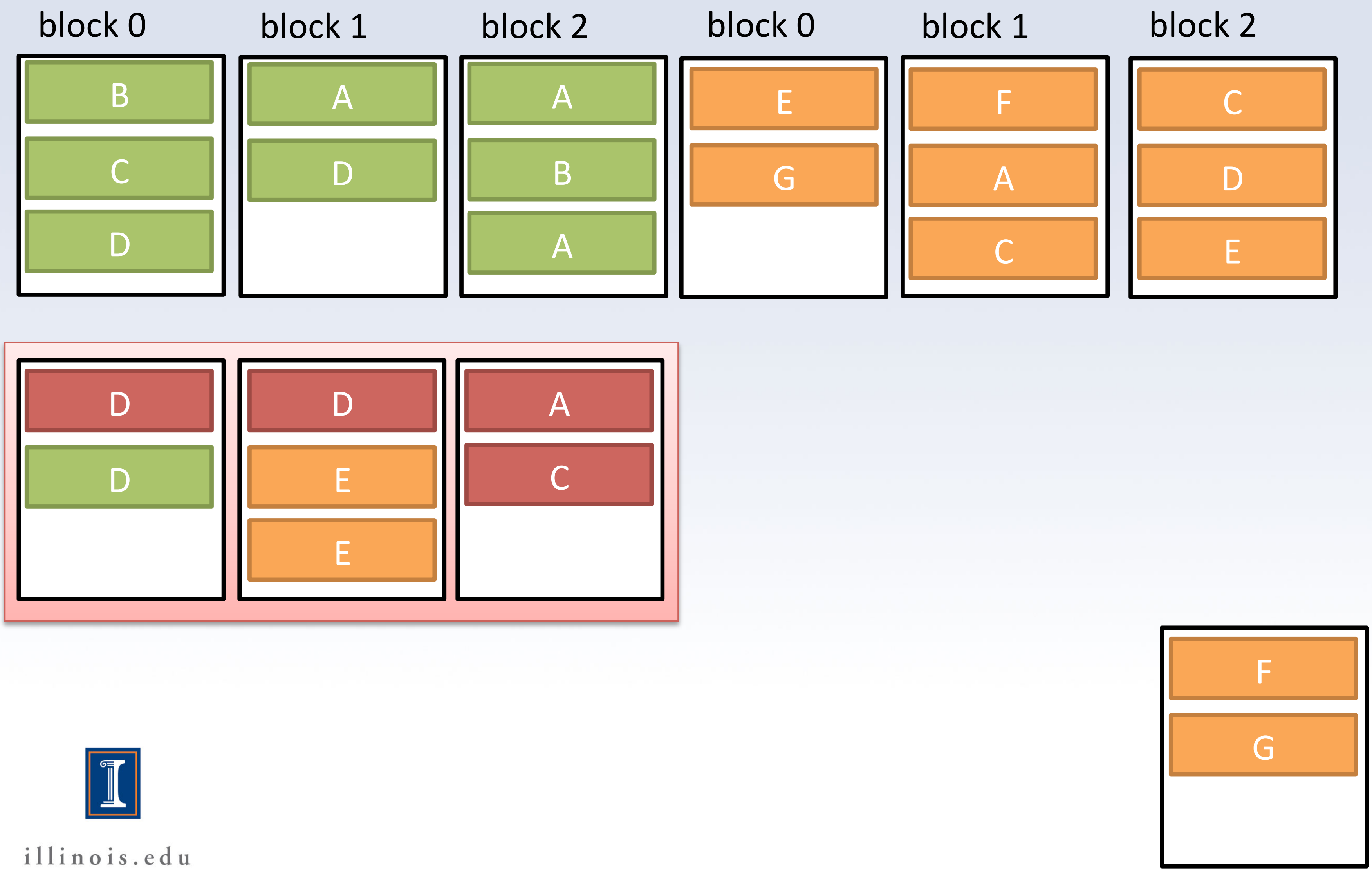
block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
D
D
C
A
C
D
E
E
F
G

Example R ∩ S

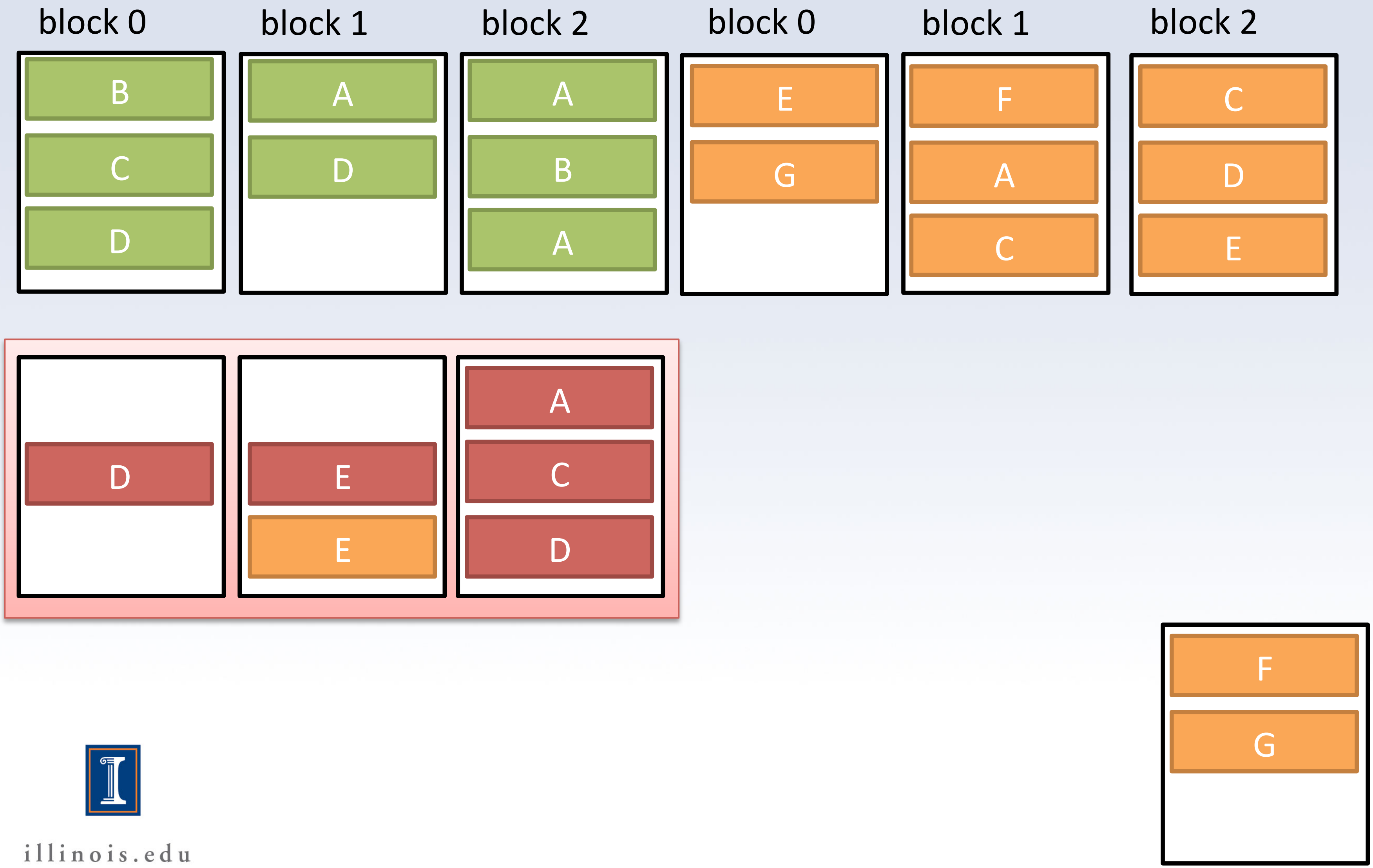


illinois.edu

Example R ∩ S

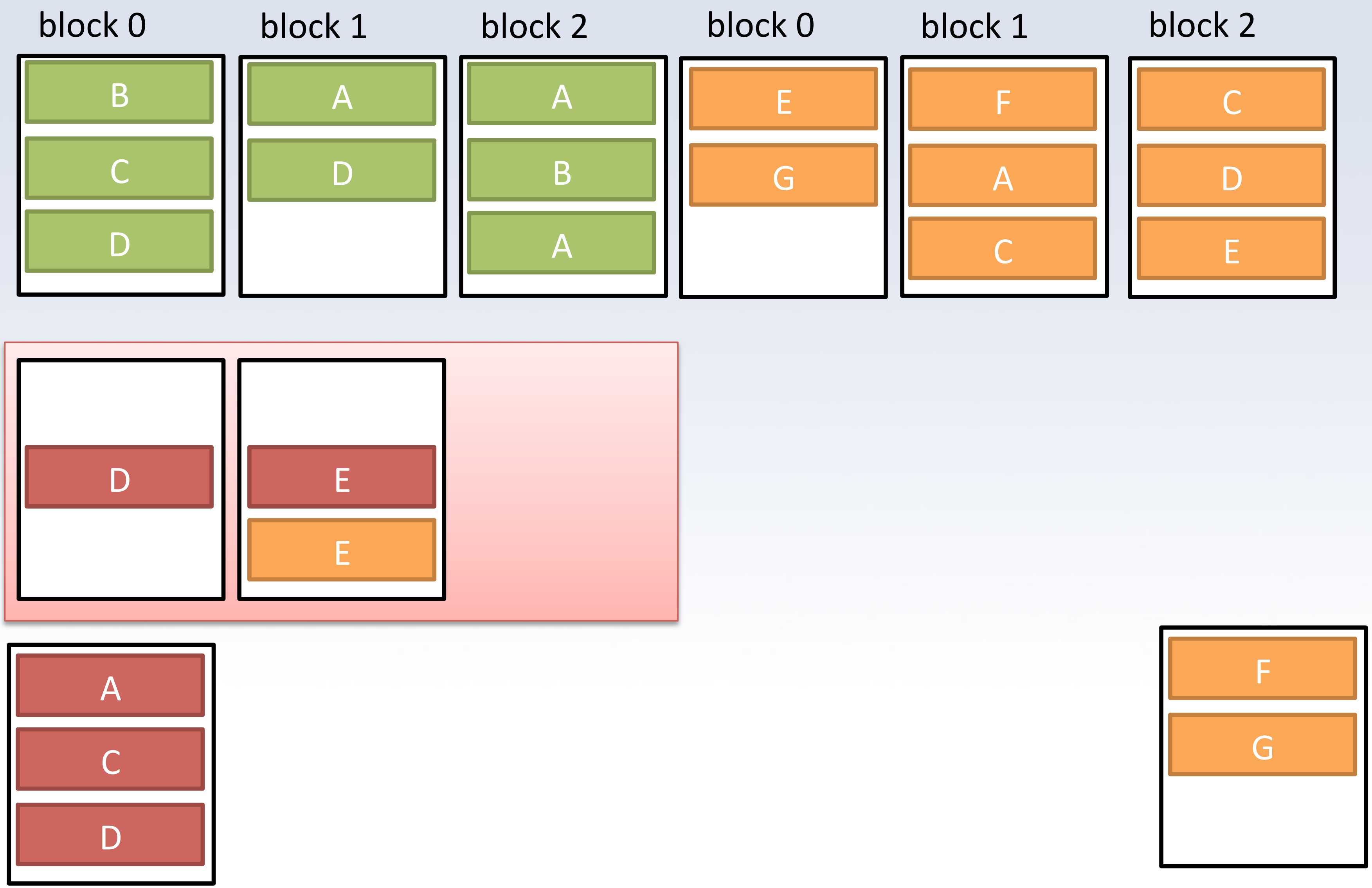

block 0
B
C
D
block 1
A
D
block 2
A
B
A
block 0
E
G
block 1
F
A
C
block 2
C
D
E
D
D
D
E
E
A
C
F
G

Example R ∩ S



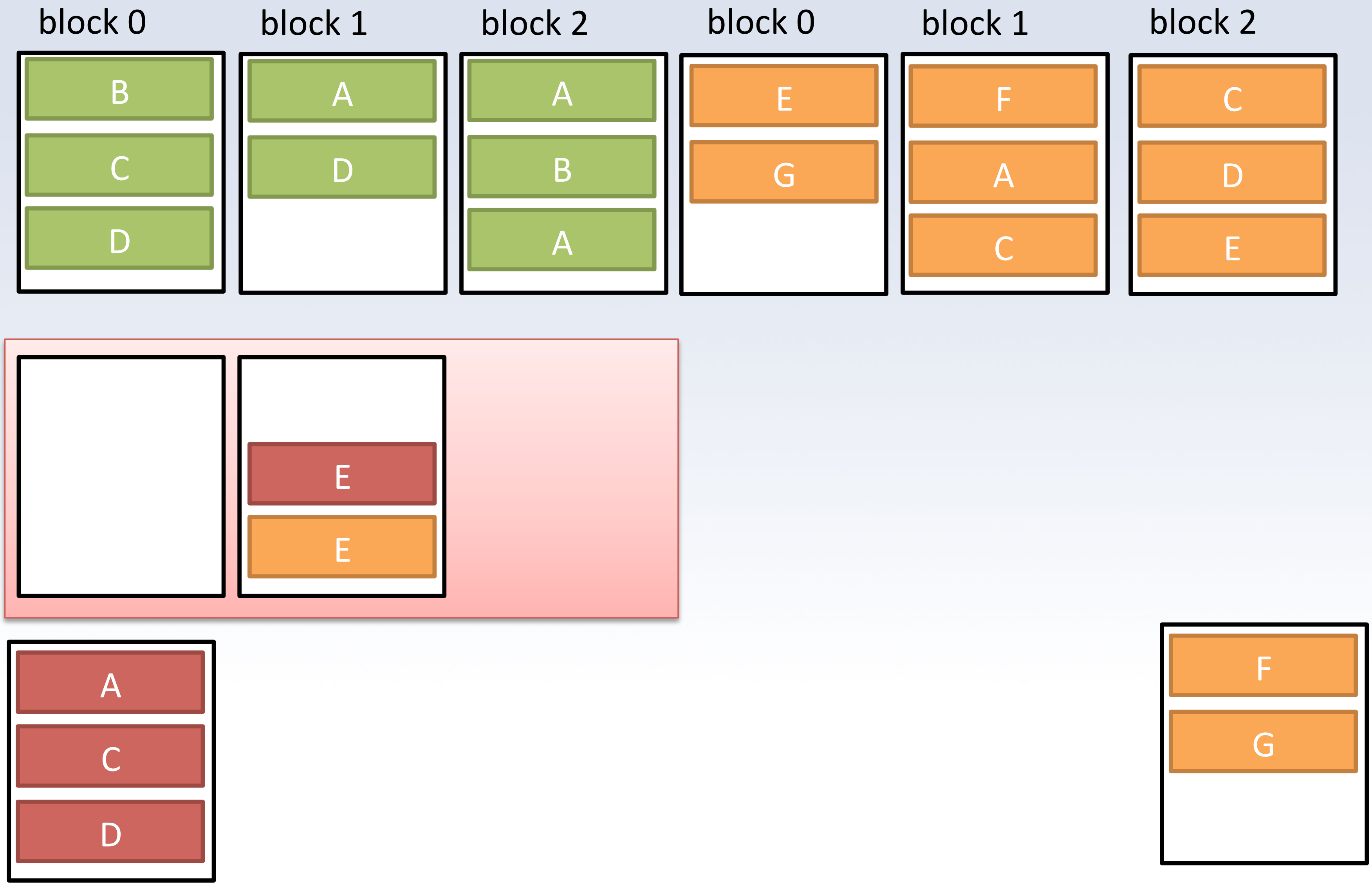
illinois.edu

Example R ∩ S

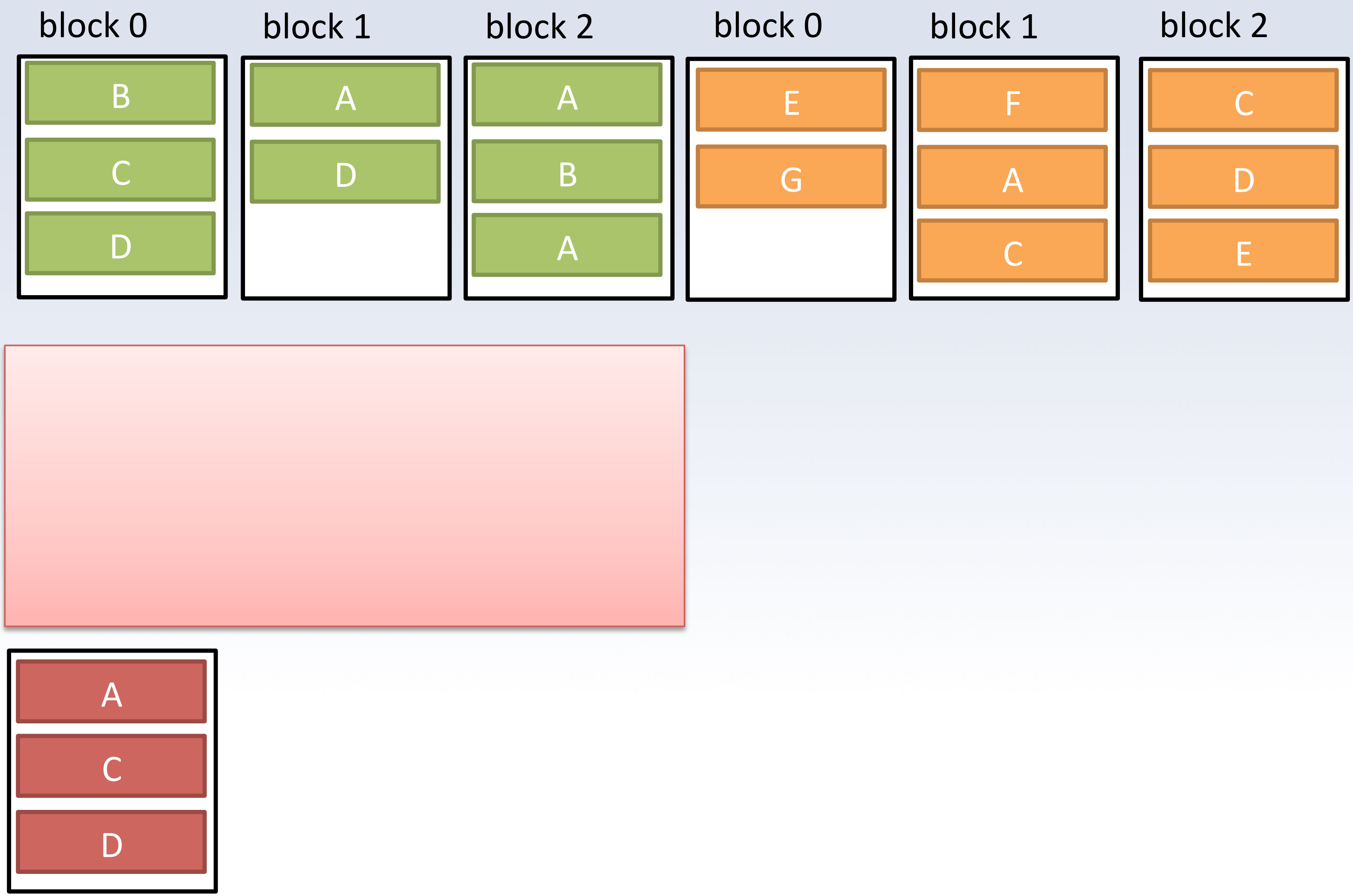


block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
D
E
E
A
C
D
F
G

Example R∩S


block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
E
E
A
C
D
F
G

Example R ∩ S



block 0
block 1
block 2
block 0
block 1
block 2
B
C
D
A
D
A
B
A
E
G
F
A
C
C
D
E
A
C
D

Sorting-based Set Difference

- We can implement R-S similar to the way we implemented $R \cap S$
- Instead of only outputting when value appears in both relations, we only output when it appears in R and not S

Simple Sort Join

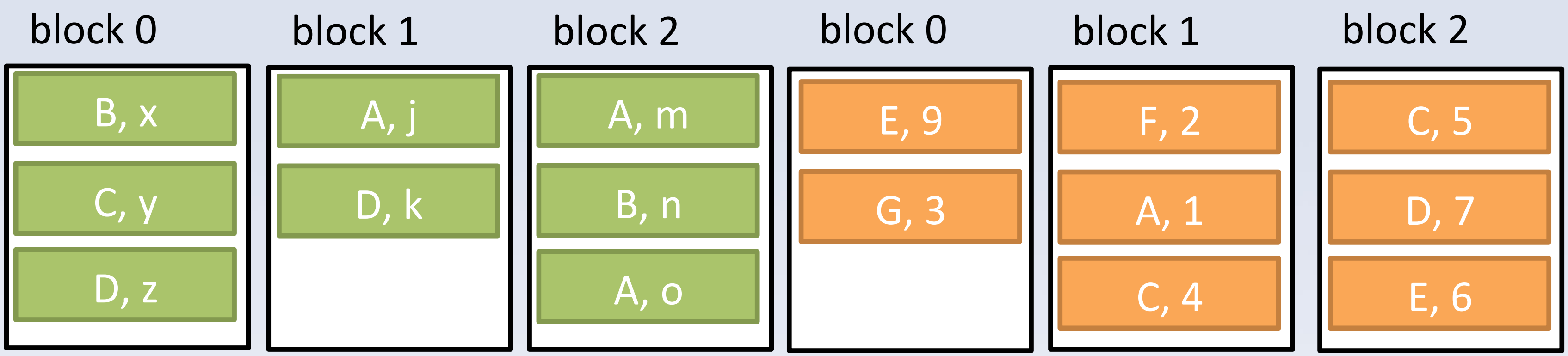
- Implementing $\bowtie$ is more complex
- Problem: we have to combine *all* of the records that share the join attributes
  - Those tuples might be bigger than memory!
  - We'd *have* to use nested loop join
- Simple sort join makes as much memory available for join as possible
- Similar to sort based intersection

Simple Sort Join

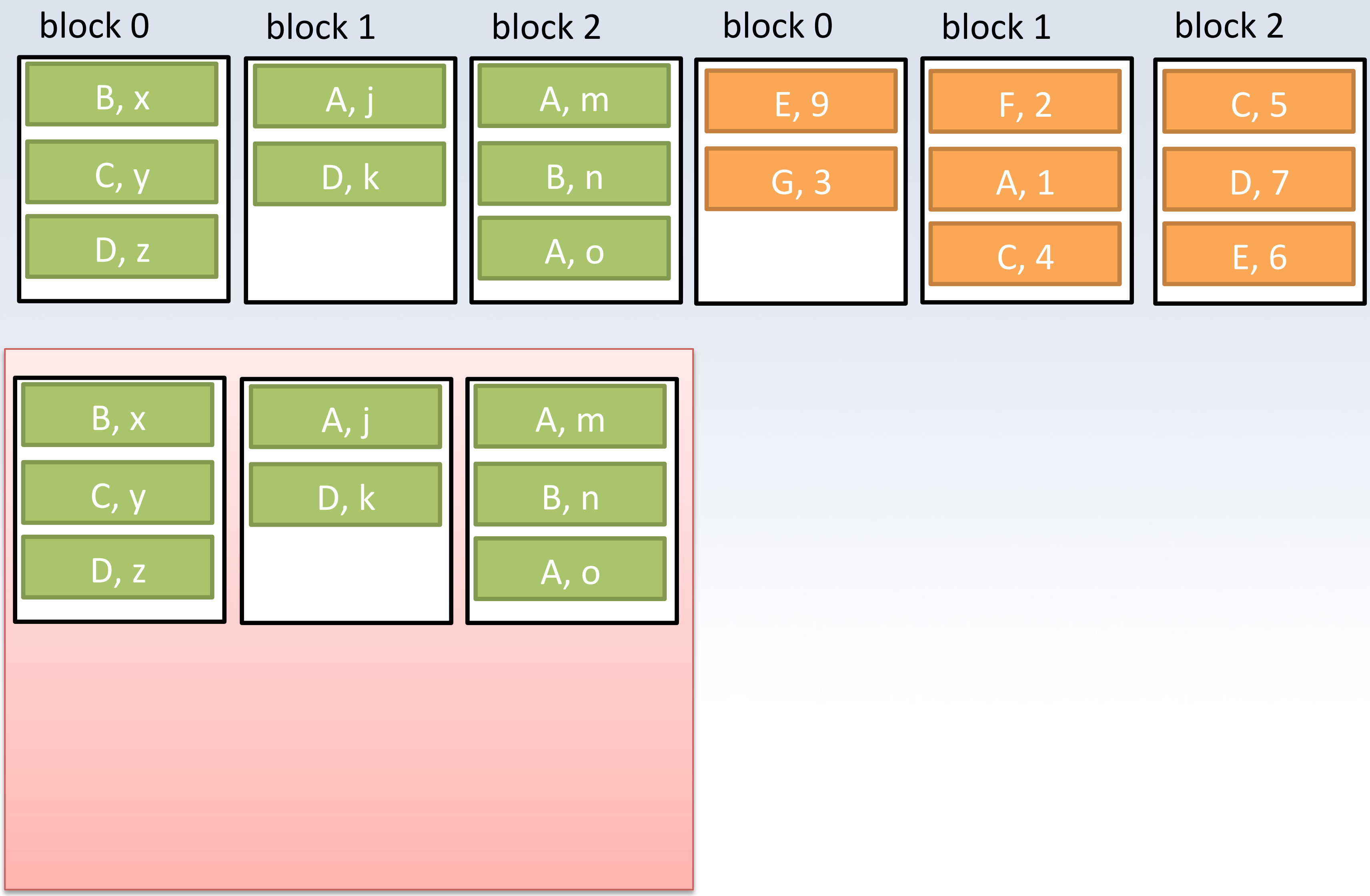
Joining R and S on attributes Y

1. Sort R using TPMMS using Y as key
2. Sort S using TPMMS using Y as key
3. Use two buffers to merge sorted lists, outputting when tuples can be joined
   - expand these two buffers until sure we have all values

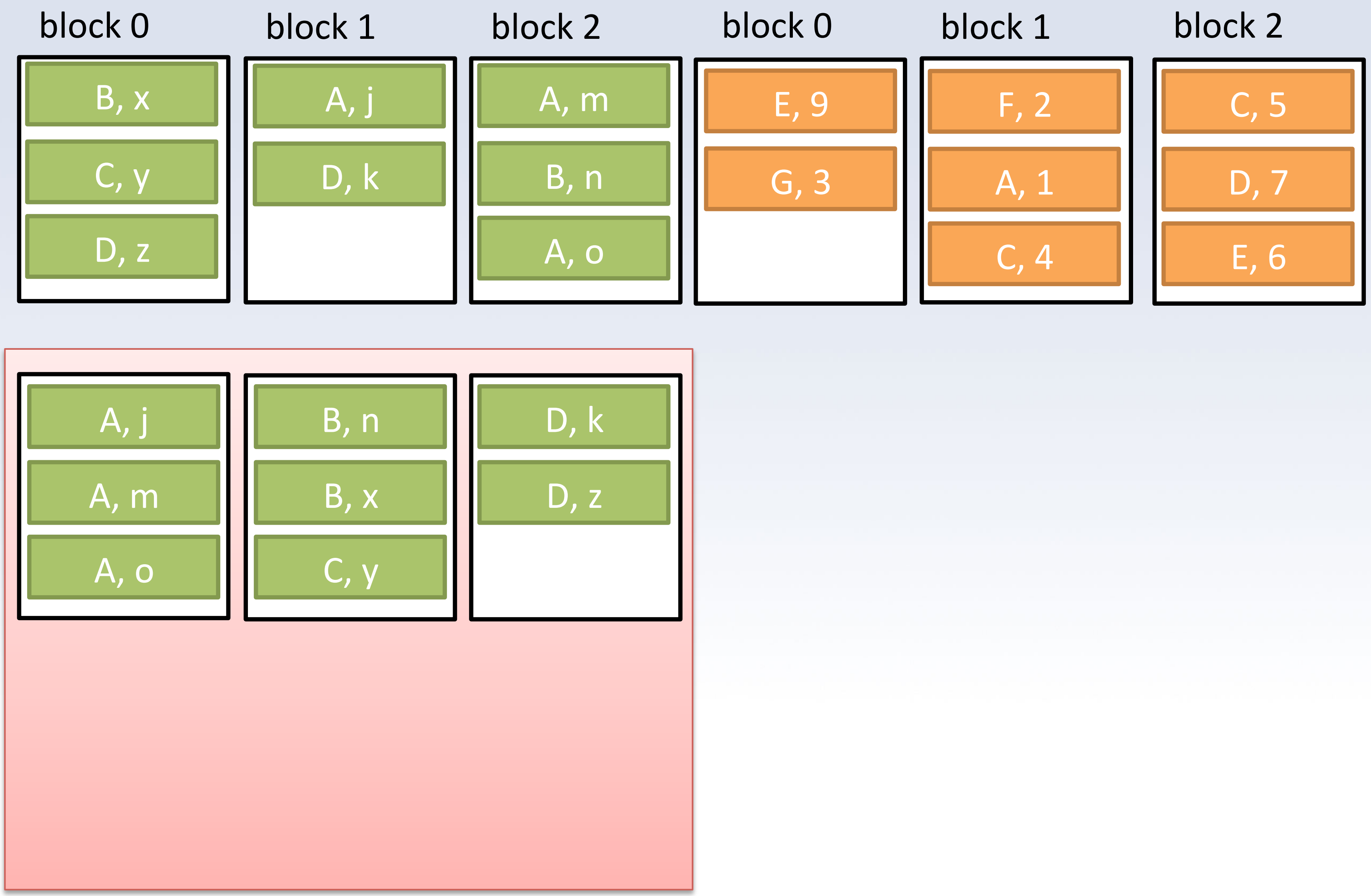
Example R⋈S


block 0
B, x
C, y
D, z
block 1
A, j
D, k
block 2
A, m
B, n
A, o
block 0
E, 9
G, 3
block 1
F, 2
A, 1
C, 4
block 2
C, 5
D, 7
E, 6

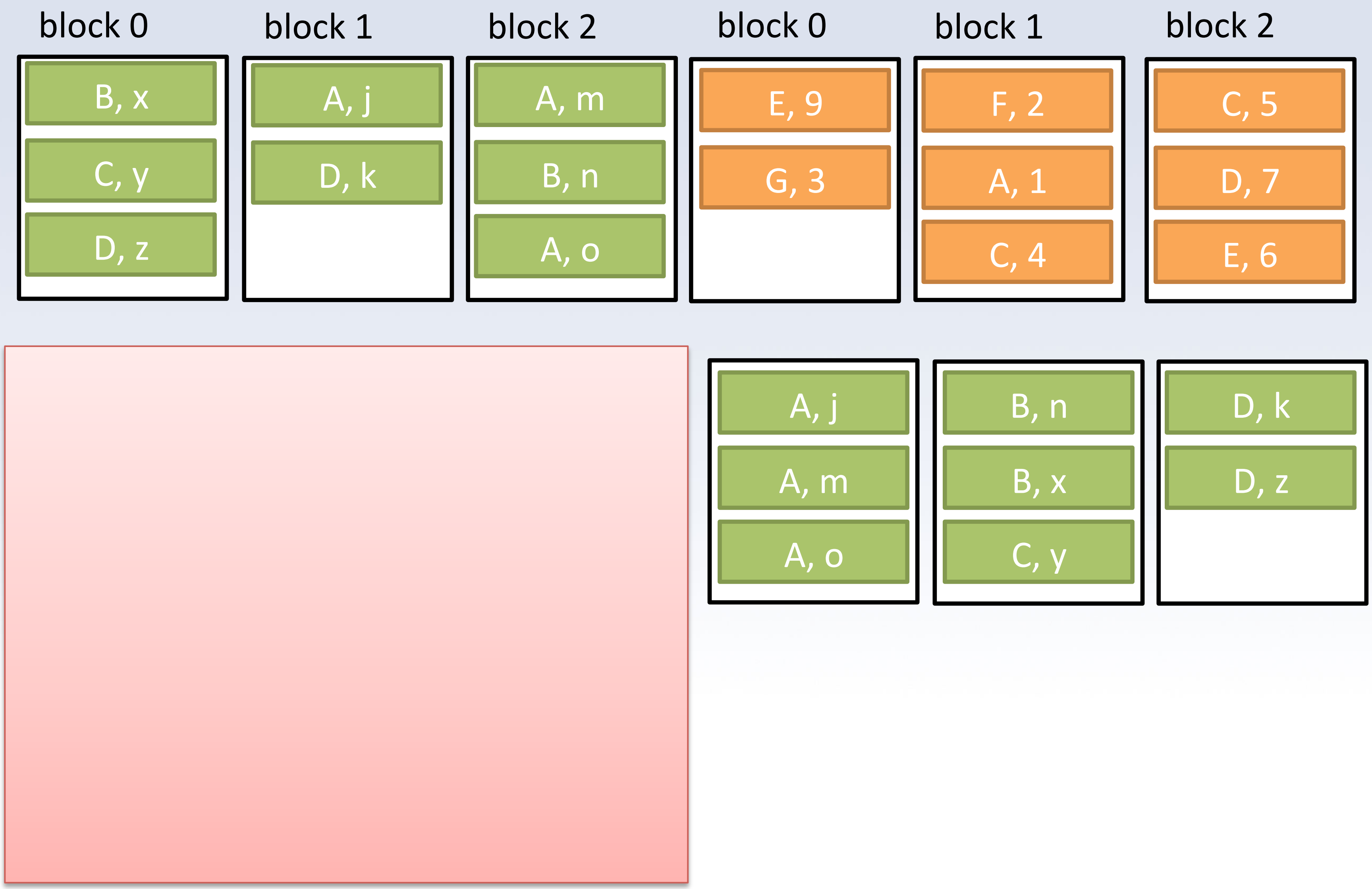
Example R⋈S


block 0
block 1
block 2
block 0
block 1
block 2
B, x
C, y
D, z
A, j
D, k
A, m
B, n
A, o
E, 9
G, 3
F, 2
A, 1
C, 4
C, 5
D, 7
E, 6
B, x
C, y
D, z
A, j
D, k
A, m
B, n
A, o

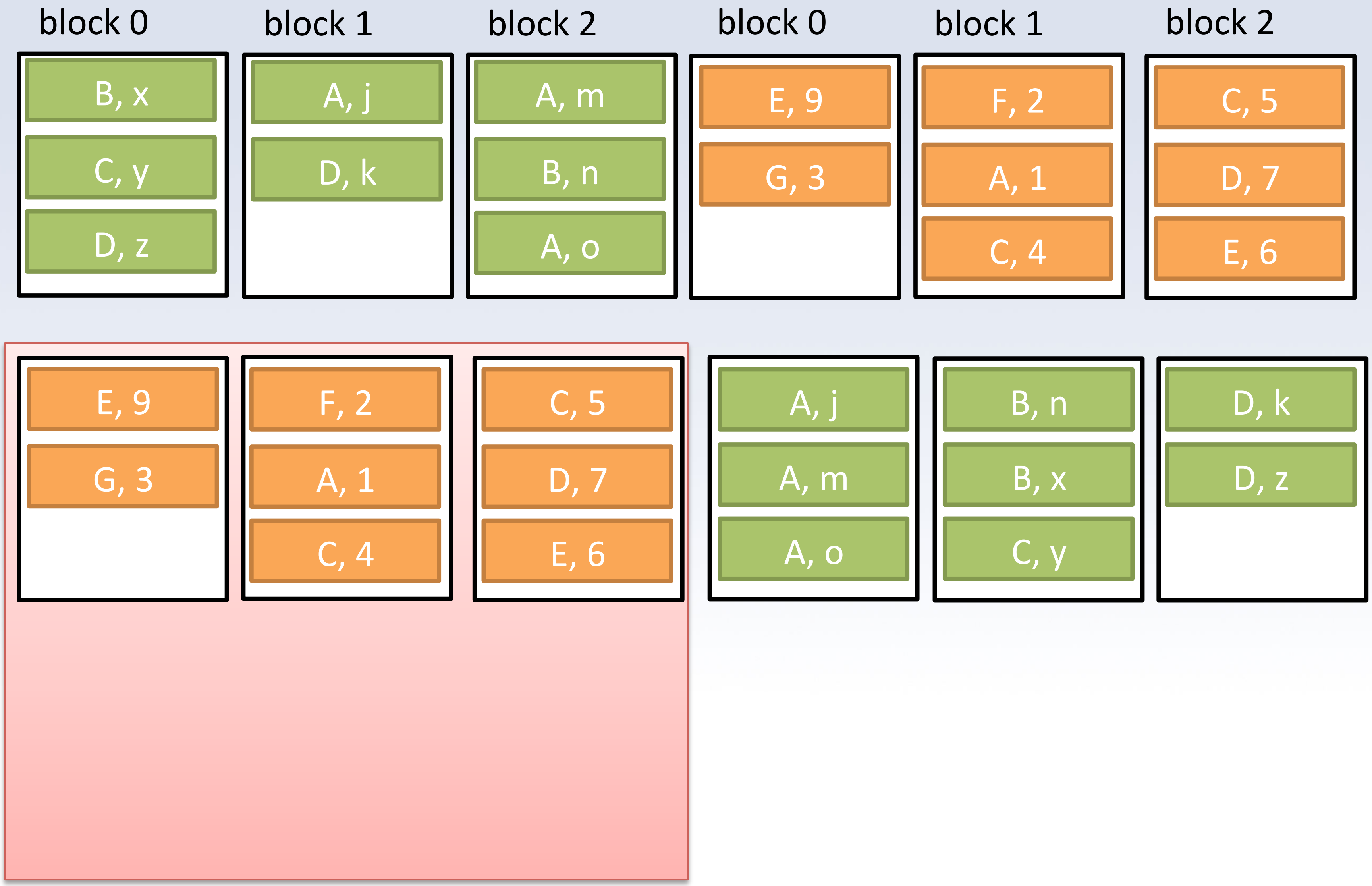
Example R⋈S


block 0
B, x
C, y
D, z
block 1
A, j
D, k
block 2
A, m
B, n
A, o
block 0
E, 9
G, 3
block 1
F, 2
A, 1
C, 4
block 2
C, 5
D, 7
E, 6
A, j
A, m
A, o
B, n
B, x
C, y
D, k
D, z

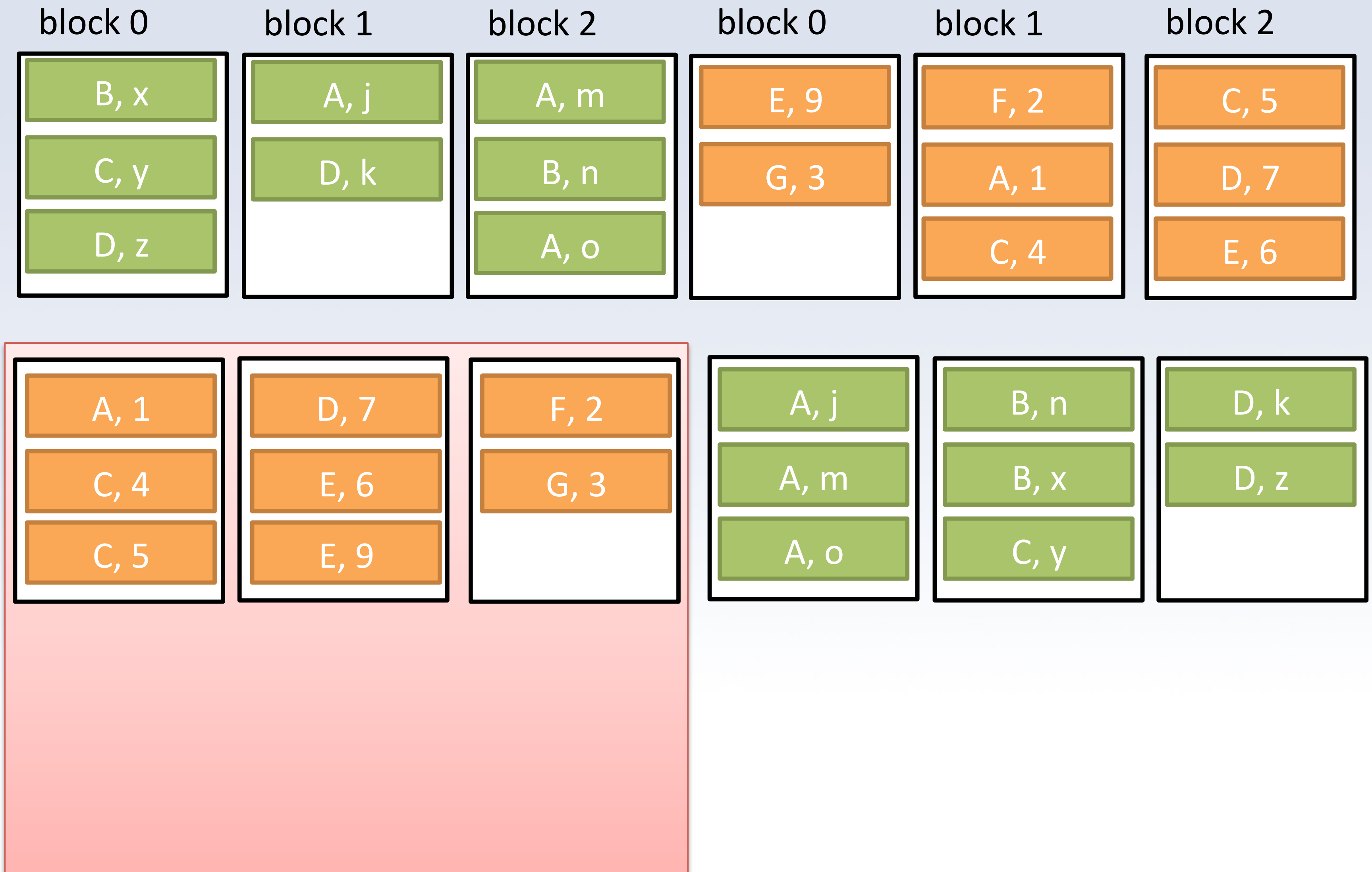
Example R⋈S


block 0
B, x
C, y
D, z
block 1
A, j
D, k
block 2
A, m
B, n
A, o
block 0
E, 9
G, 3
block 1
F, 2
A, 1
C, 4
block 2
C, 5
D, 7
E, 6
A, j
A, m
A, o
B, n
B, x
C, y
D, k
D, z

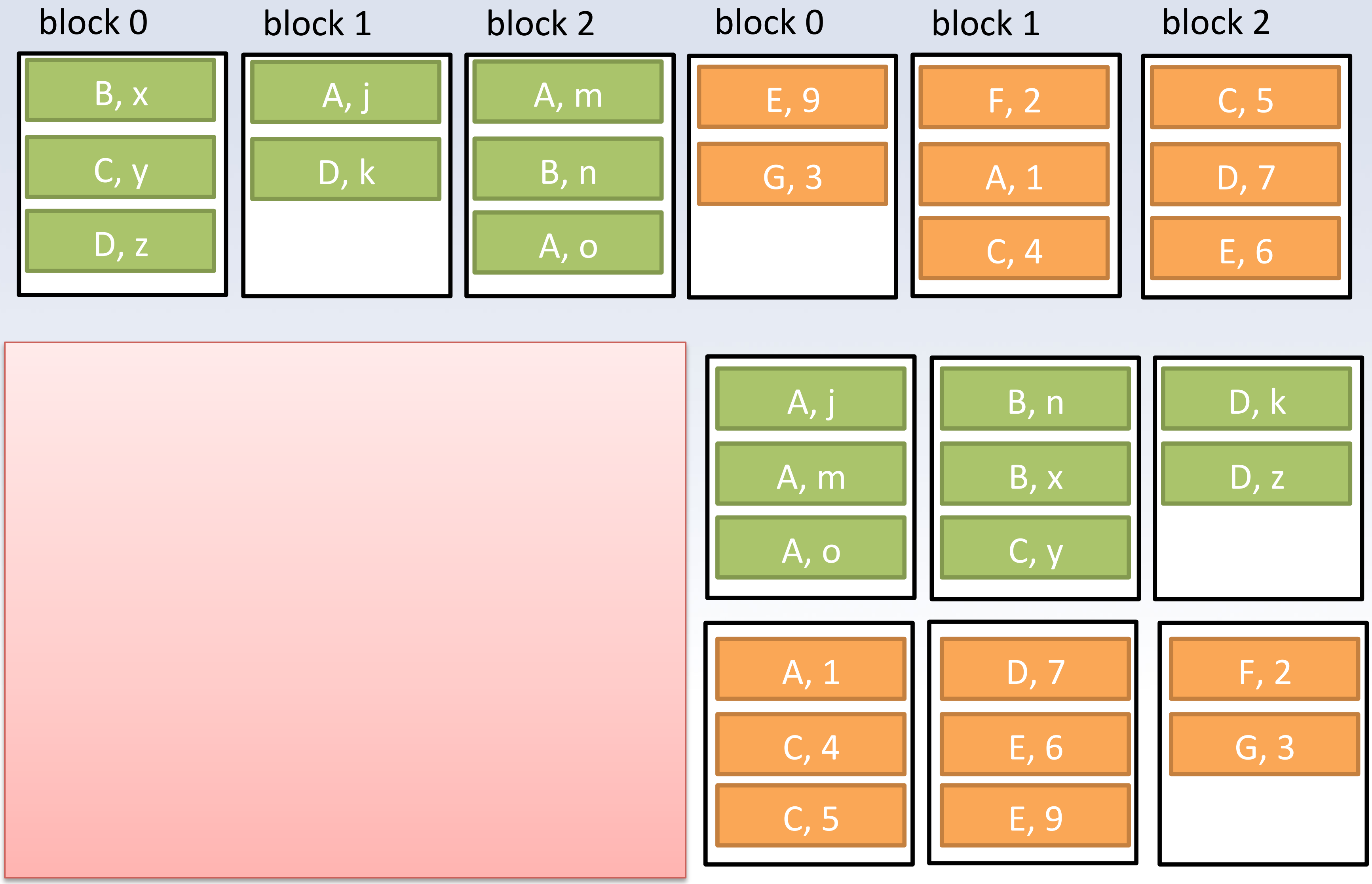
Example R⋈S


block 0
block 1
block 2
block 0
block 1
block 2
B, x
C, y
D, z
A, j
D, k
A, m
B, n
A, o
E, 9
G, 3
F, 2
A, 1
C, 4
C, 5
D, 7
E, 6
E, 9
G, 3
F, 2
A, 1
C, 4
C, 5
D, 7
E, 6
A, j
A, m
A, o
B, n
B, x
C, y
D, k
D, z

Example R⋈S

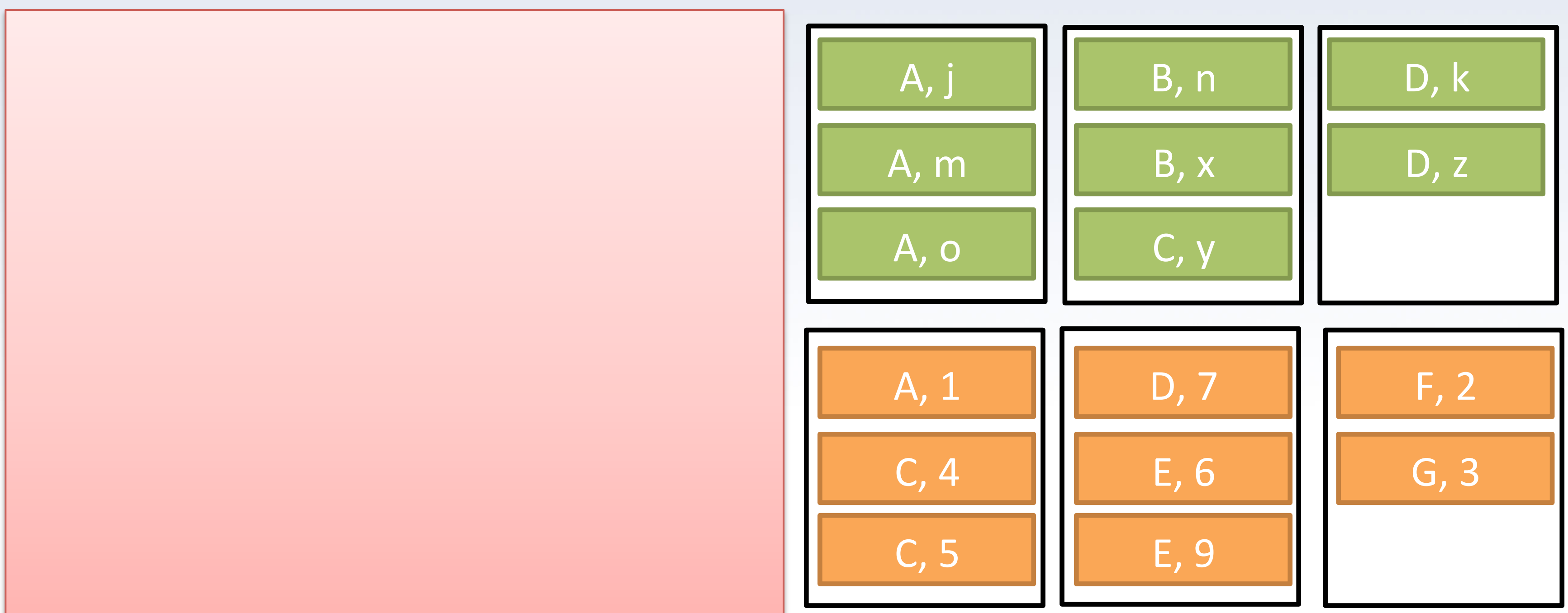

block 0
block 1
block 2
block 0
block 1
block 2
B, x
C, y
D, z
A, j
D, k
A, m
B, n
A, o
E, 9
G, 3
F, 2
A, 1
C, 4
C, 5
D, 7
E, 6
A, 1
C, 4
C, 5
D, 7
E, 6
E, 9
F, 2
G, 3
A, j
A, m
A, o
B, n
B, x
C, y
D, k
D, z

Example R⋈S

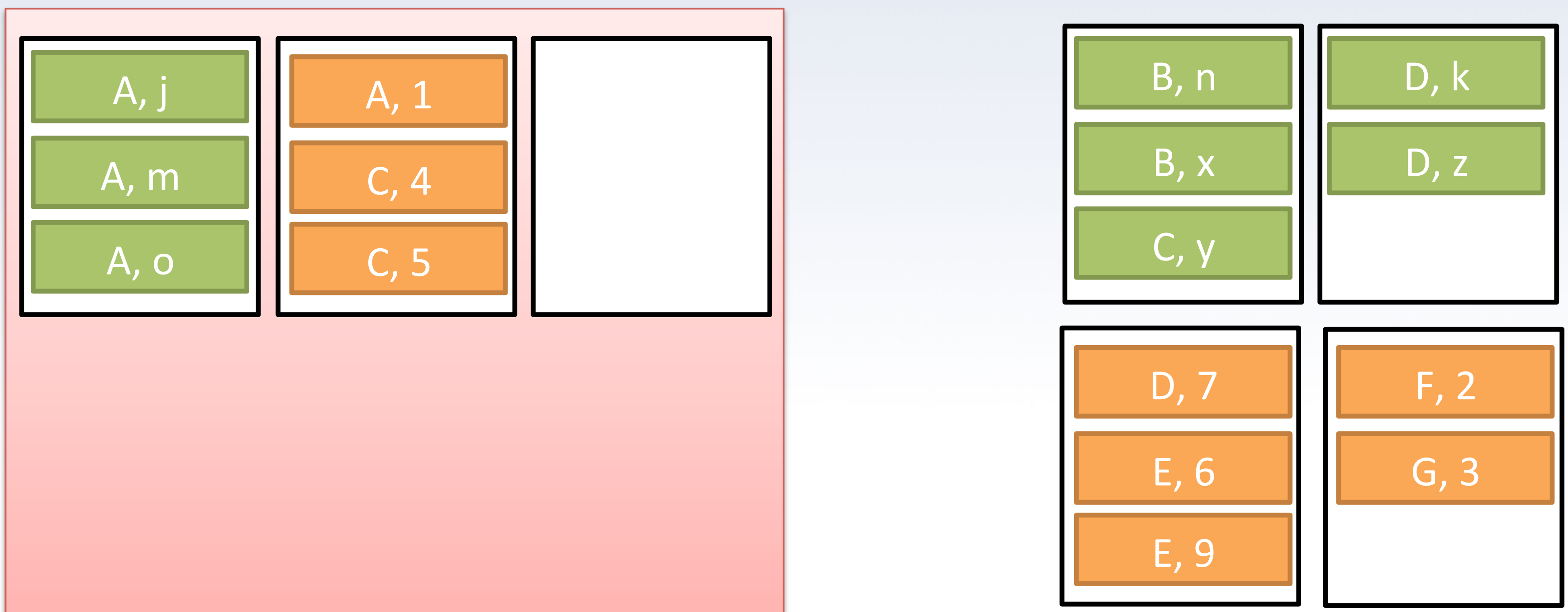


block 0
block 1
block 2
block 0
block 1
block 2
B, x
C, y
D, z
A, j
D, k
A, m
B, n
A, o
E, 9
G, 3
F, 2
A, 1
C, 4
C, 5
D, 7
E, 6
A, j
A, m
A, o
B, n
B, x
C, y
D, k
D, z
A, 1
C, 4
C, 5
D, 7
E, 6
E, 9
F, 2
G, 3

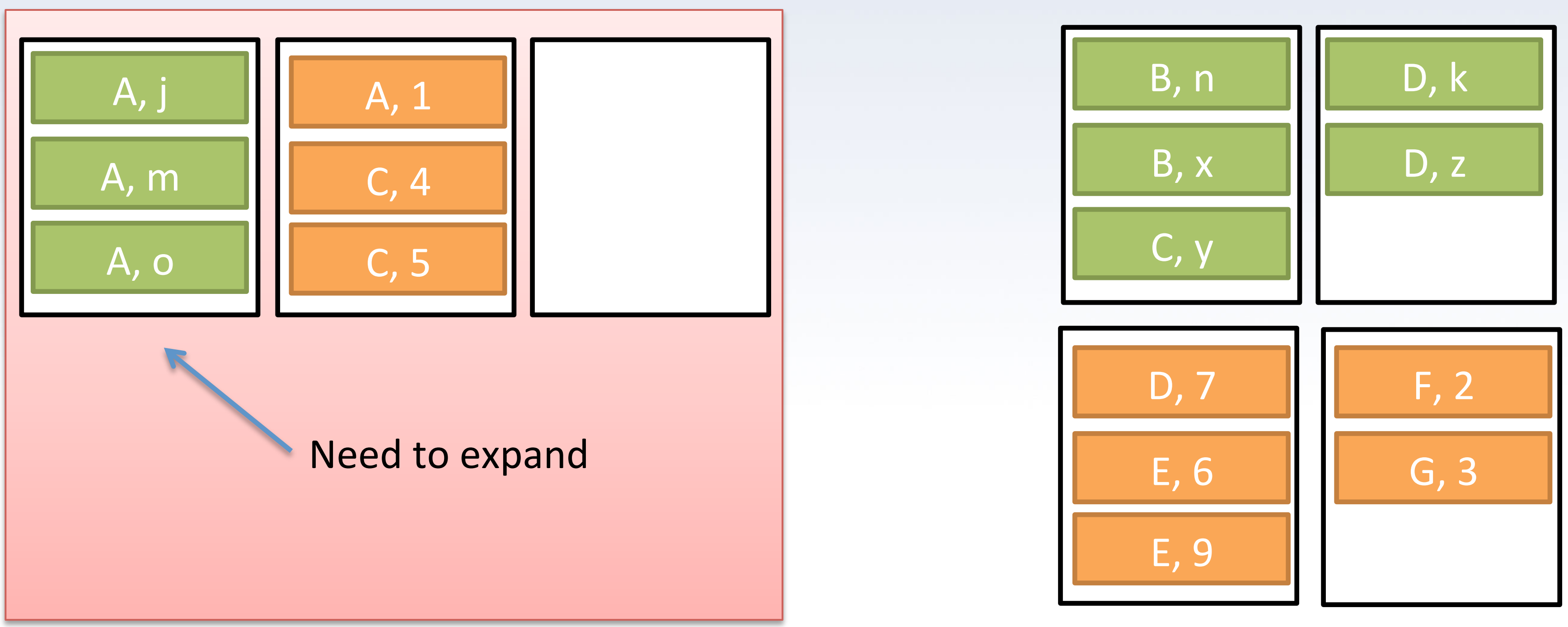
Example R⋈S



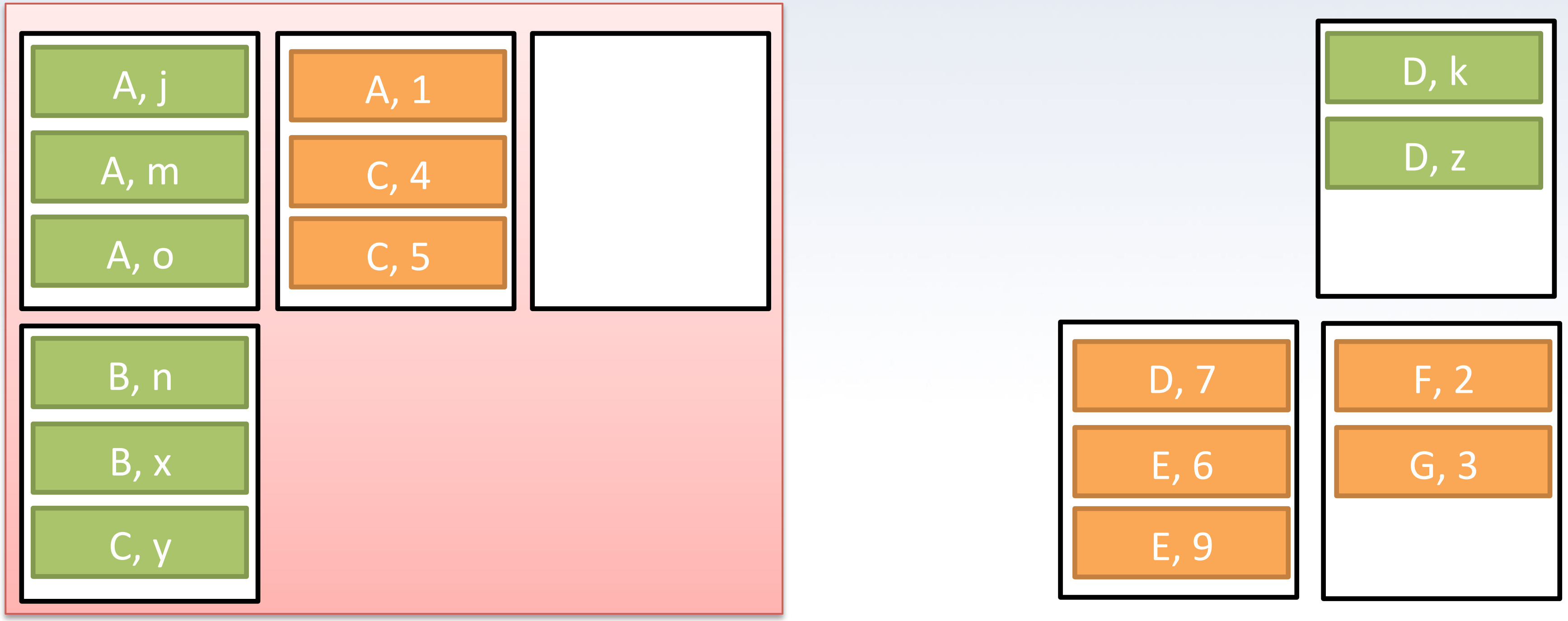
Example R⋈S



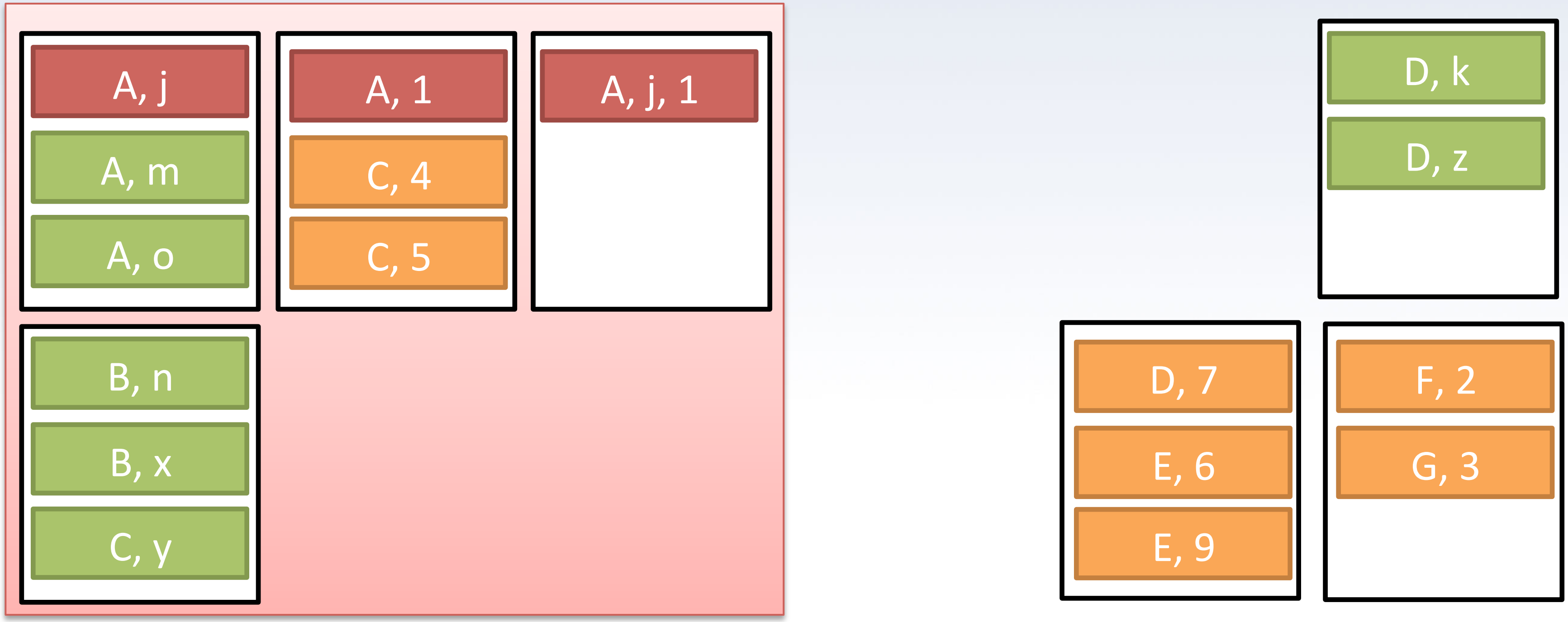
Example R⋈S



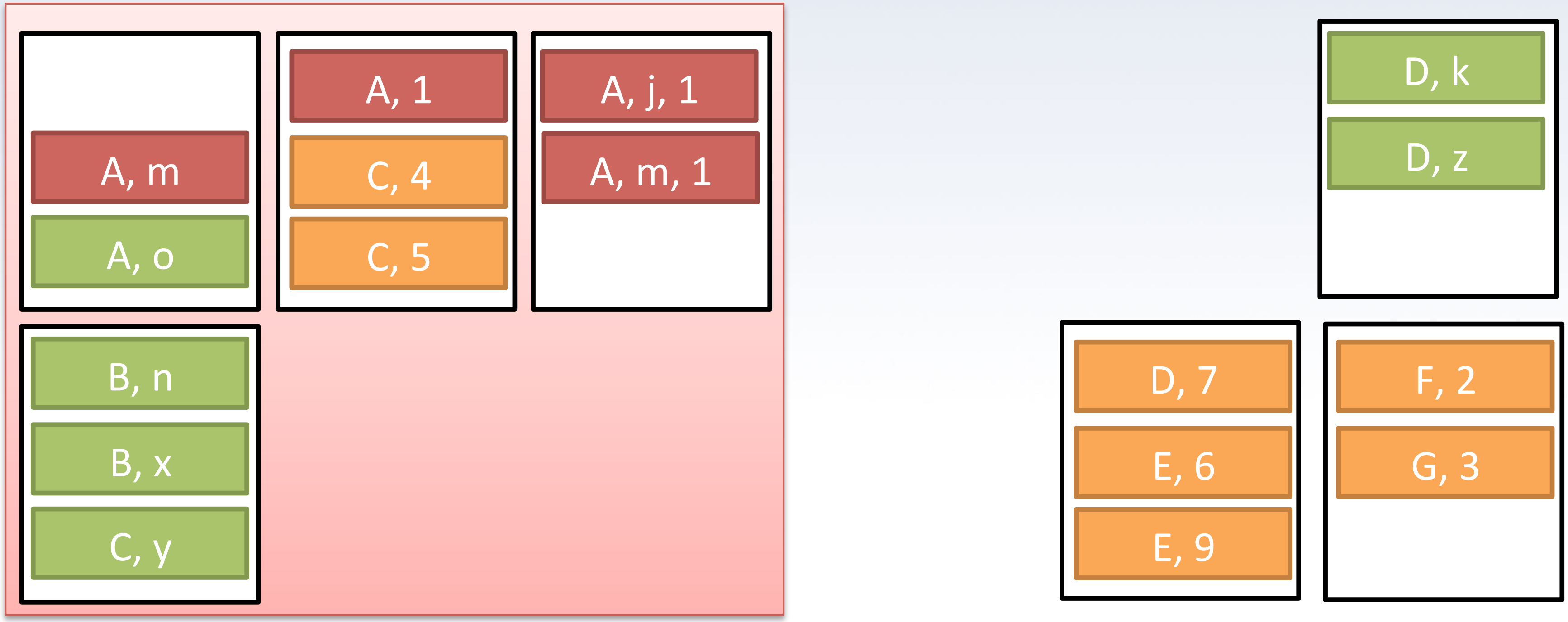
Example R⋈S



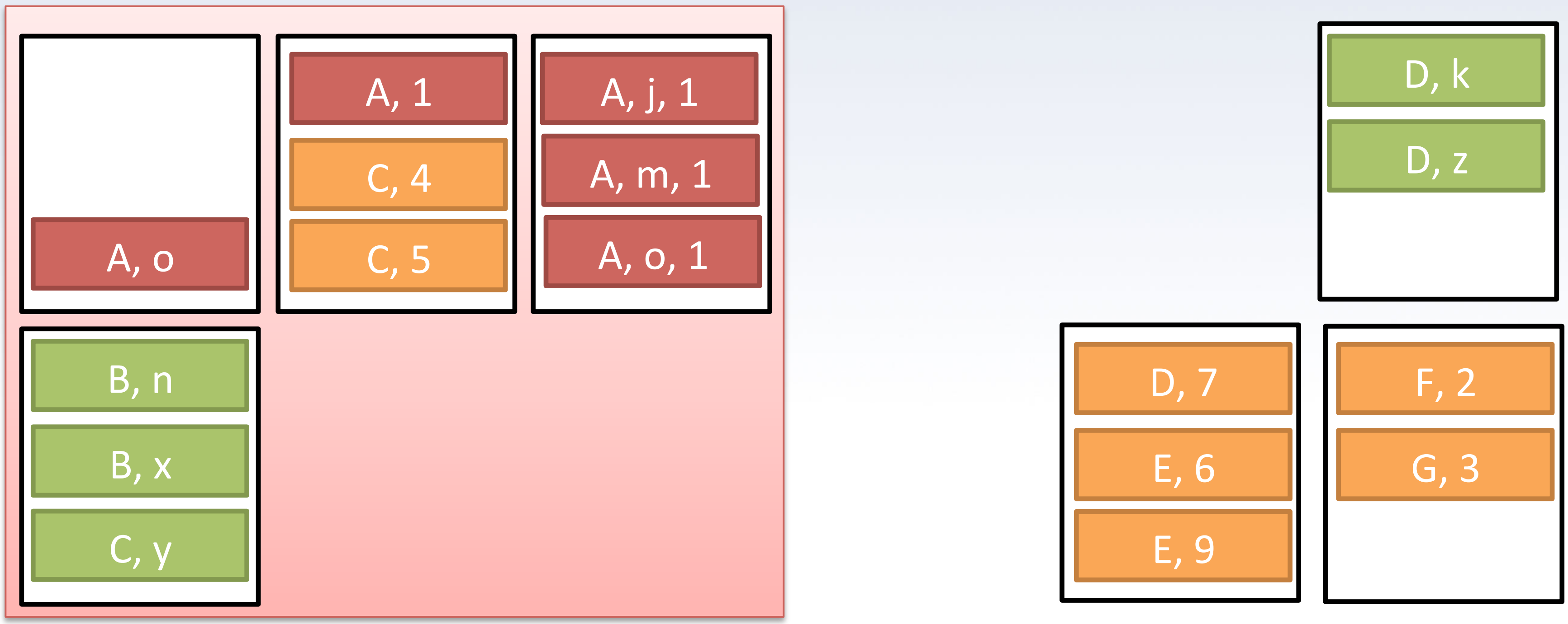
Example R⋈S



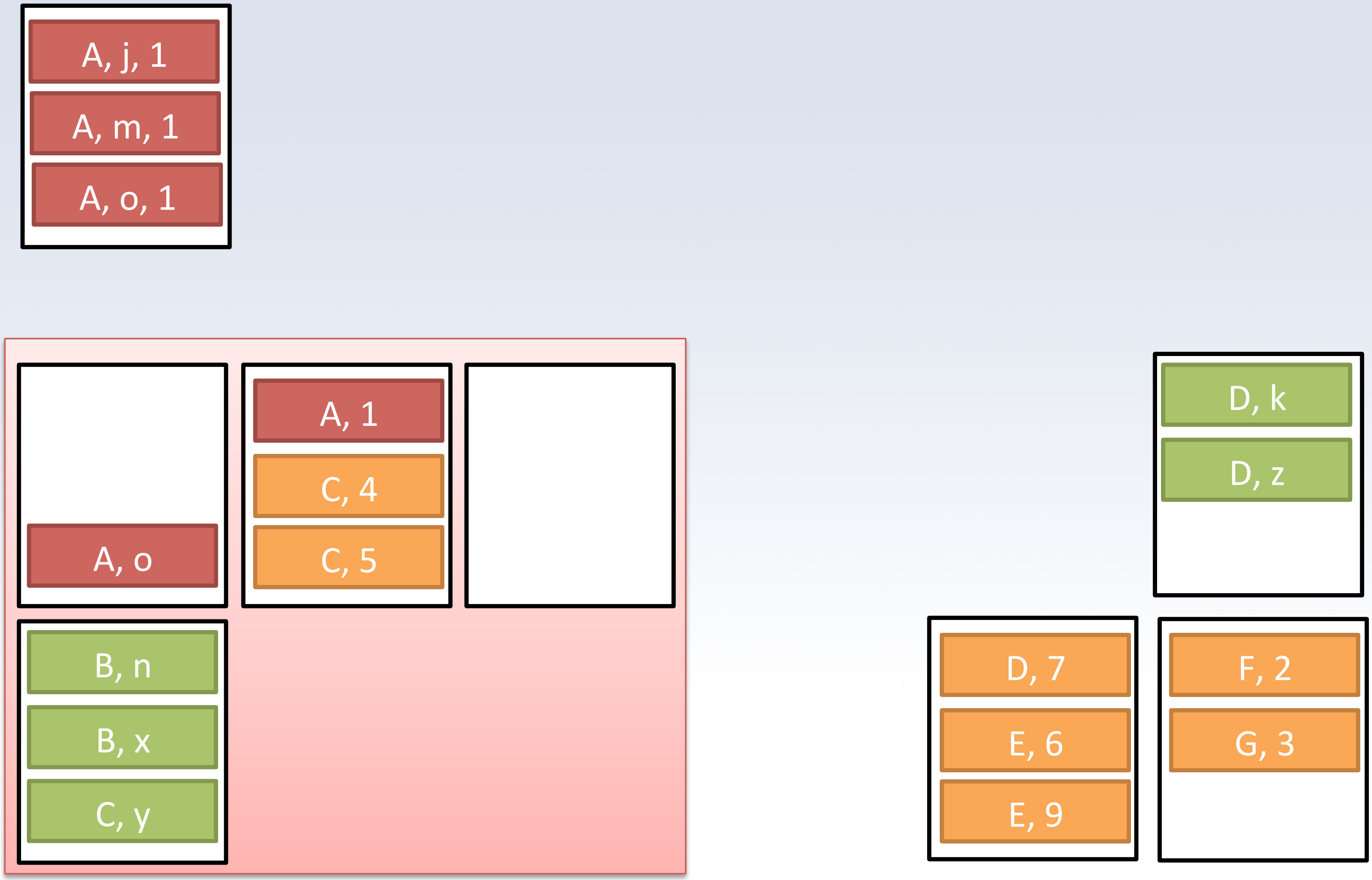
Example R⋈S



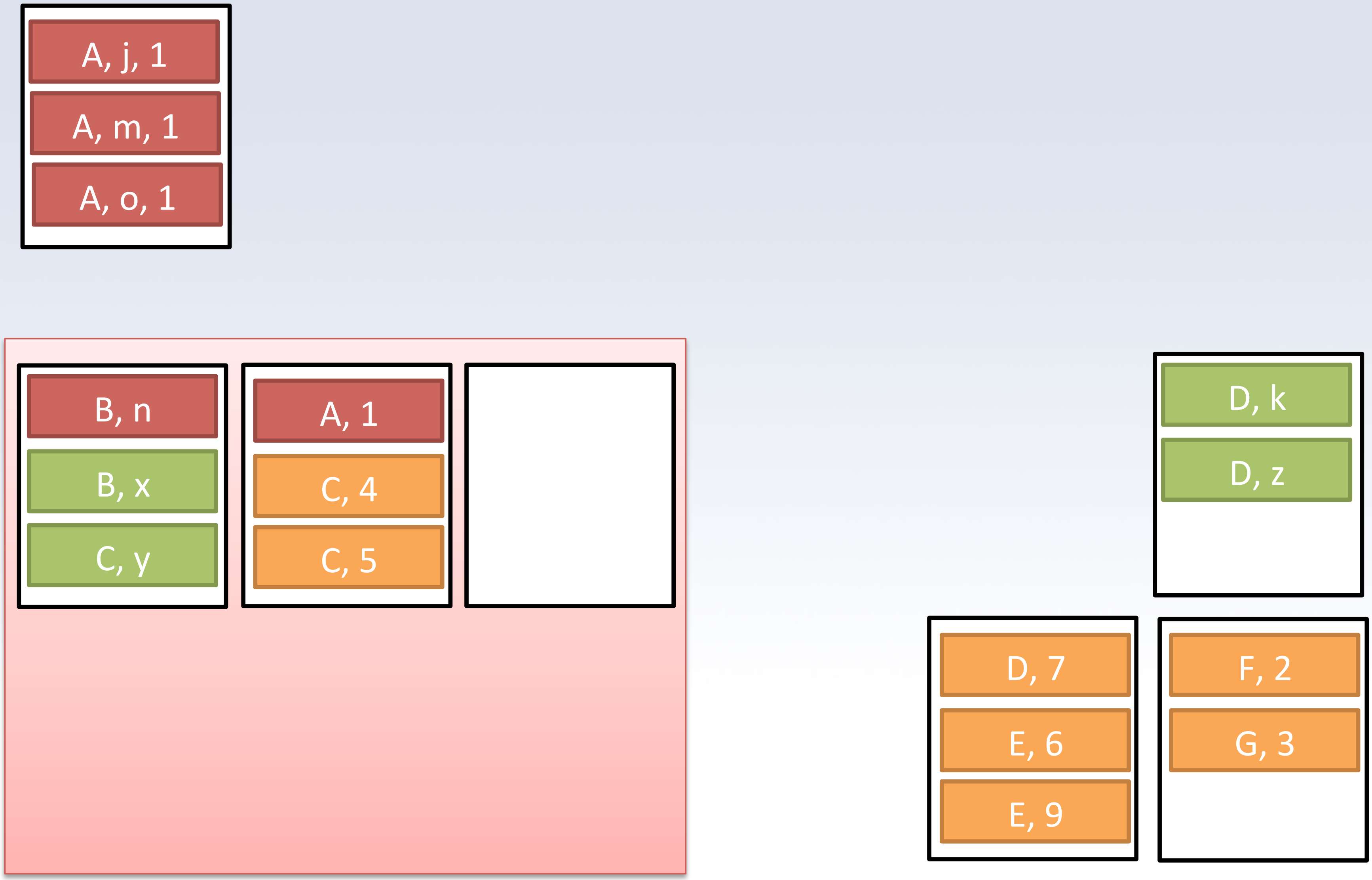
Example R⋈S



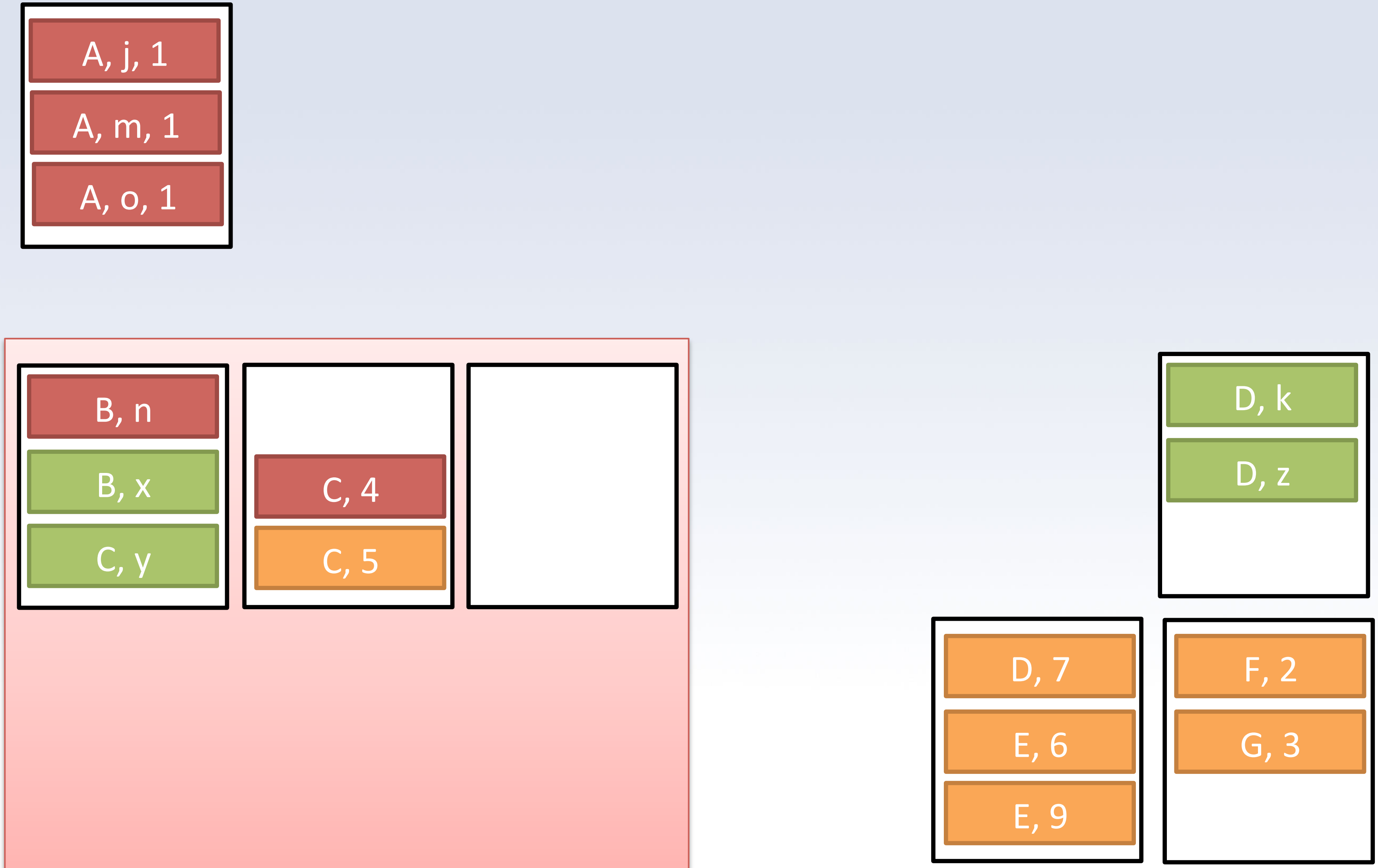
Example R⋈S



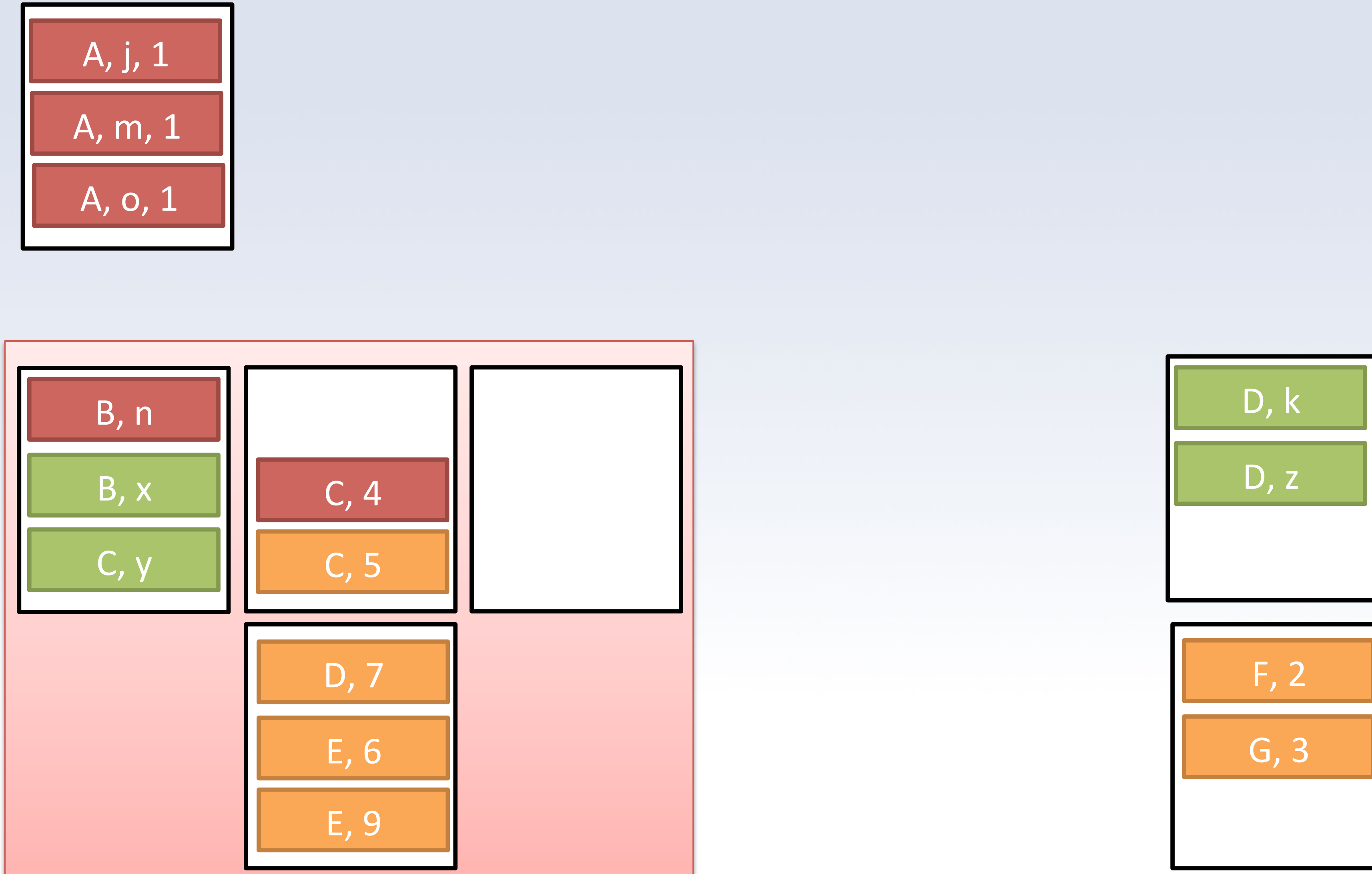
Example R⋈S



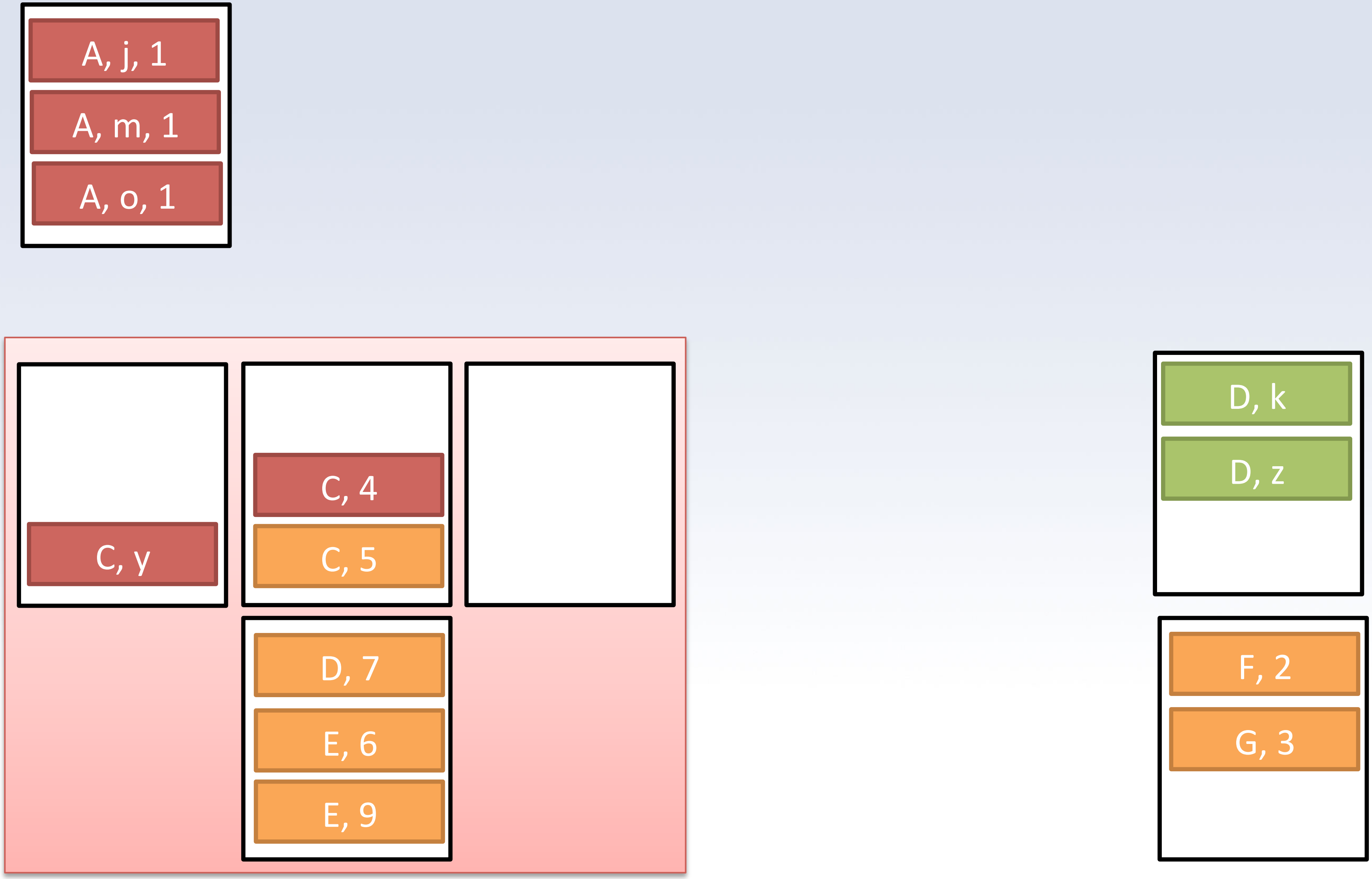
Example R⋈S



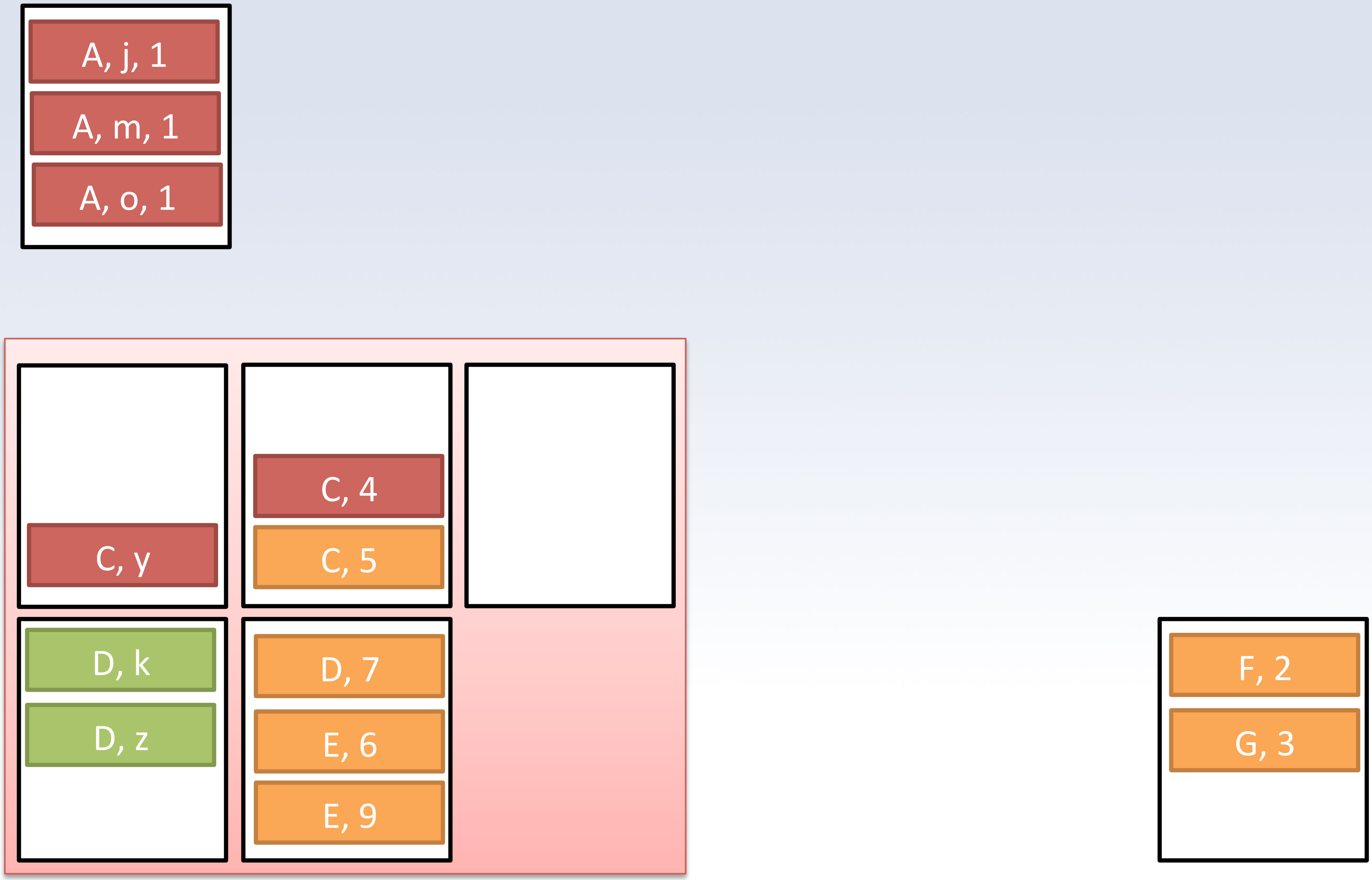
Example R⋈S



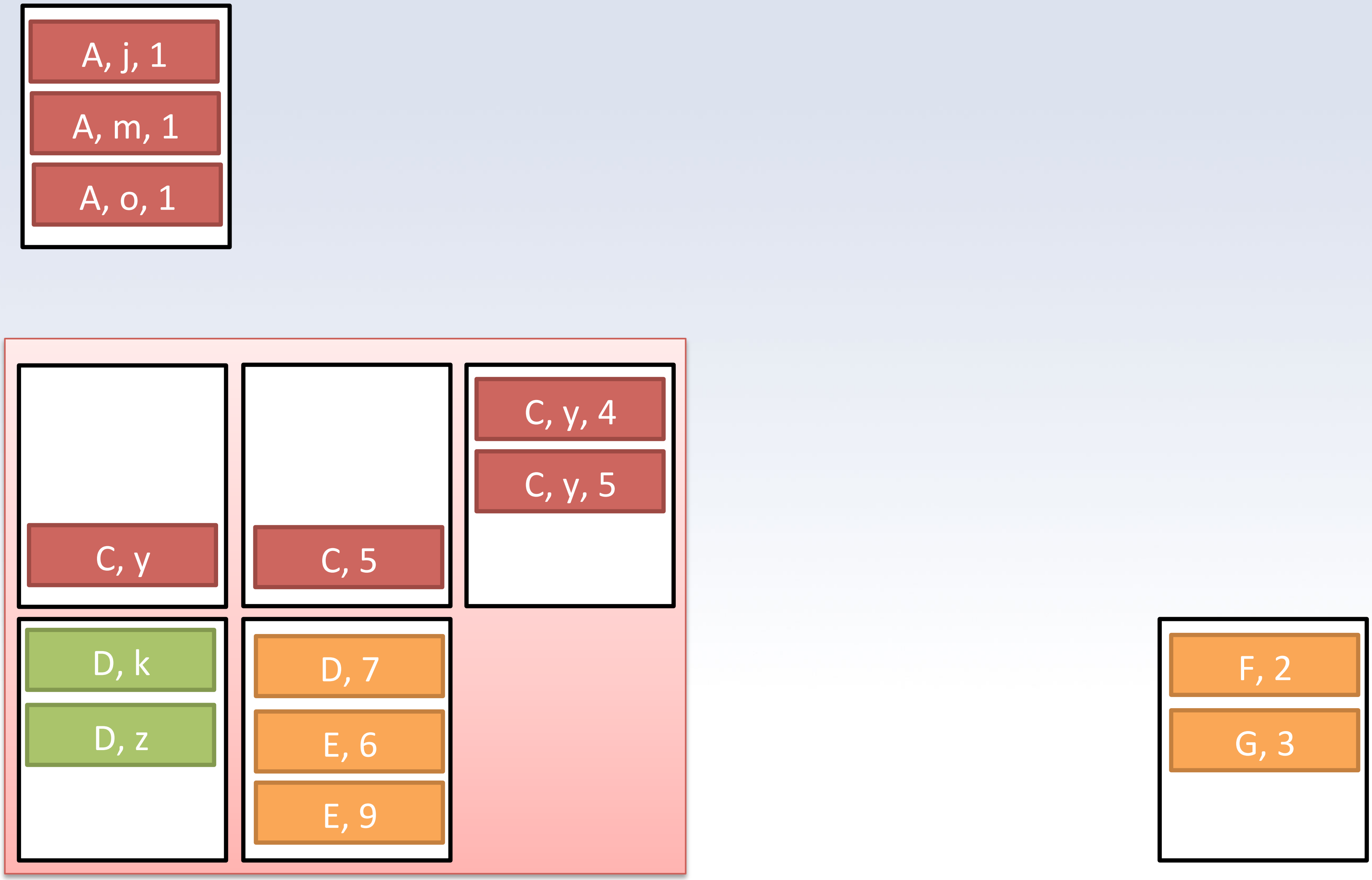
Example R⋈S



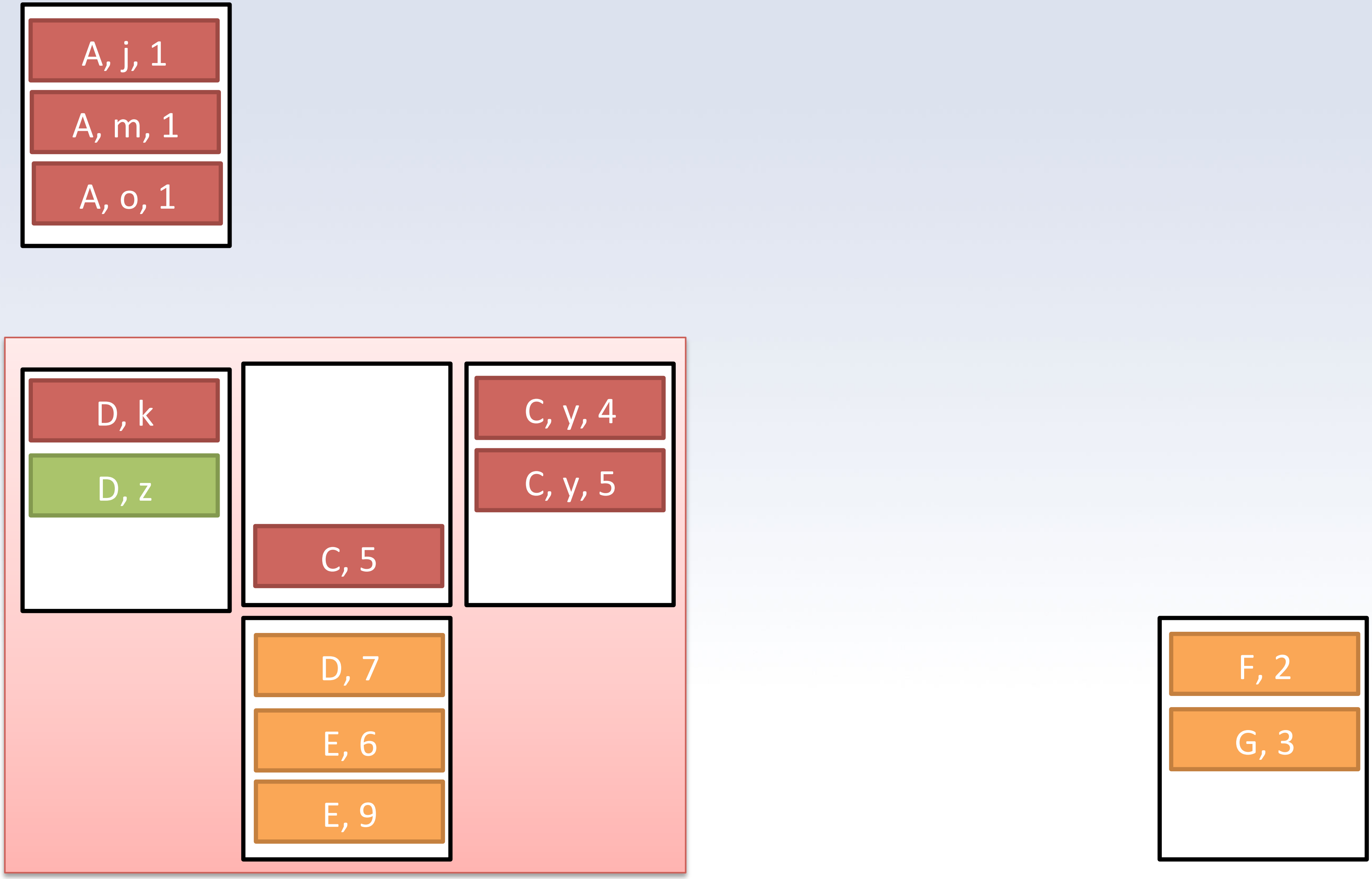
Example R⋈S



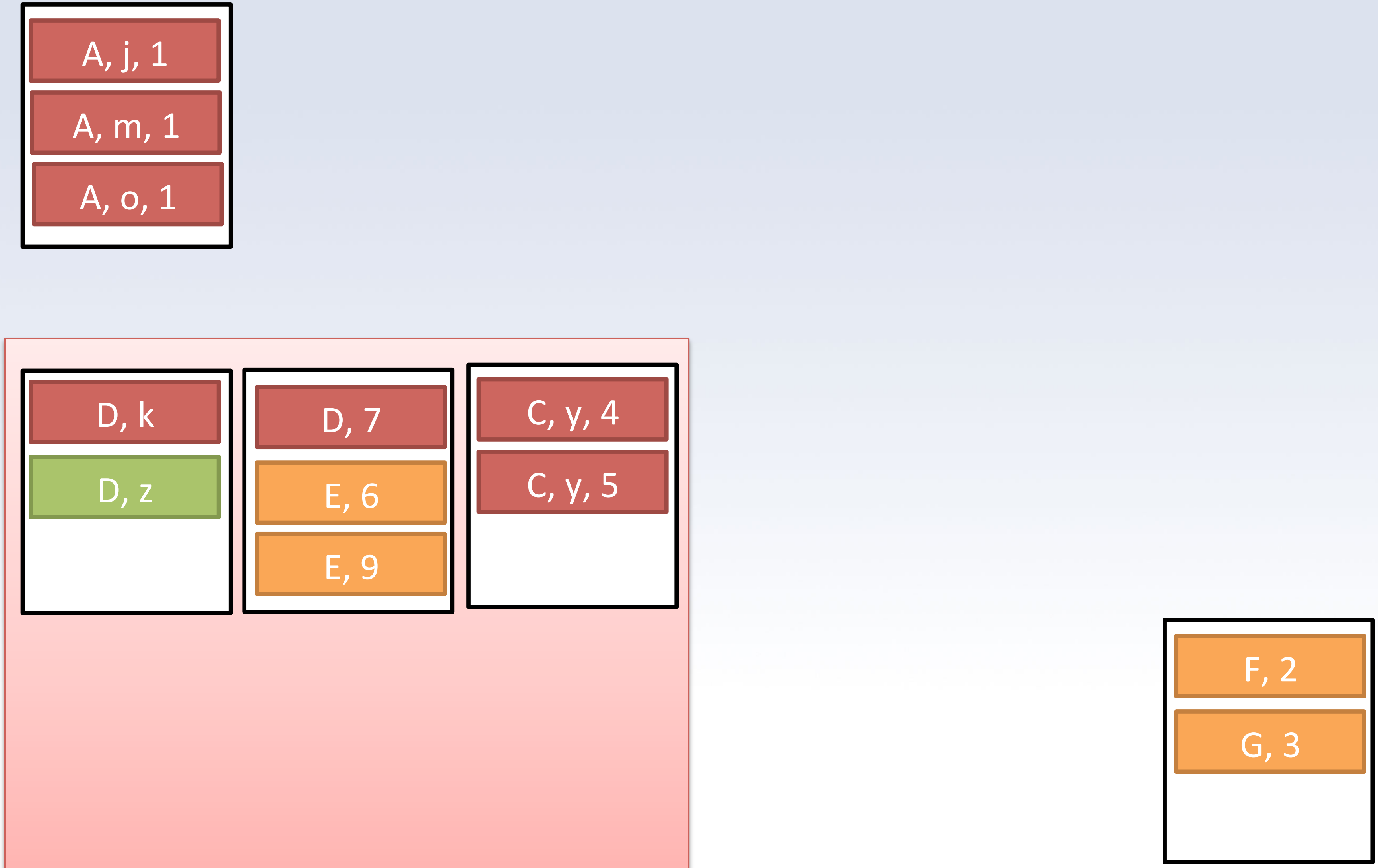
Example R⋈S



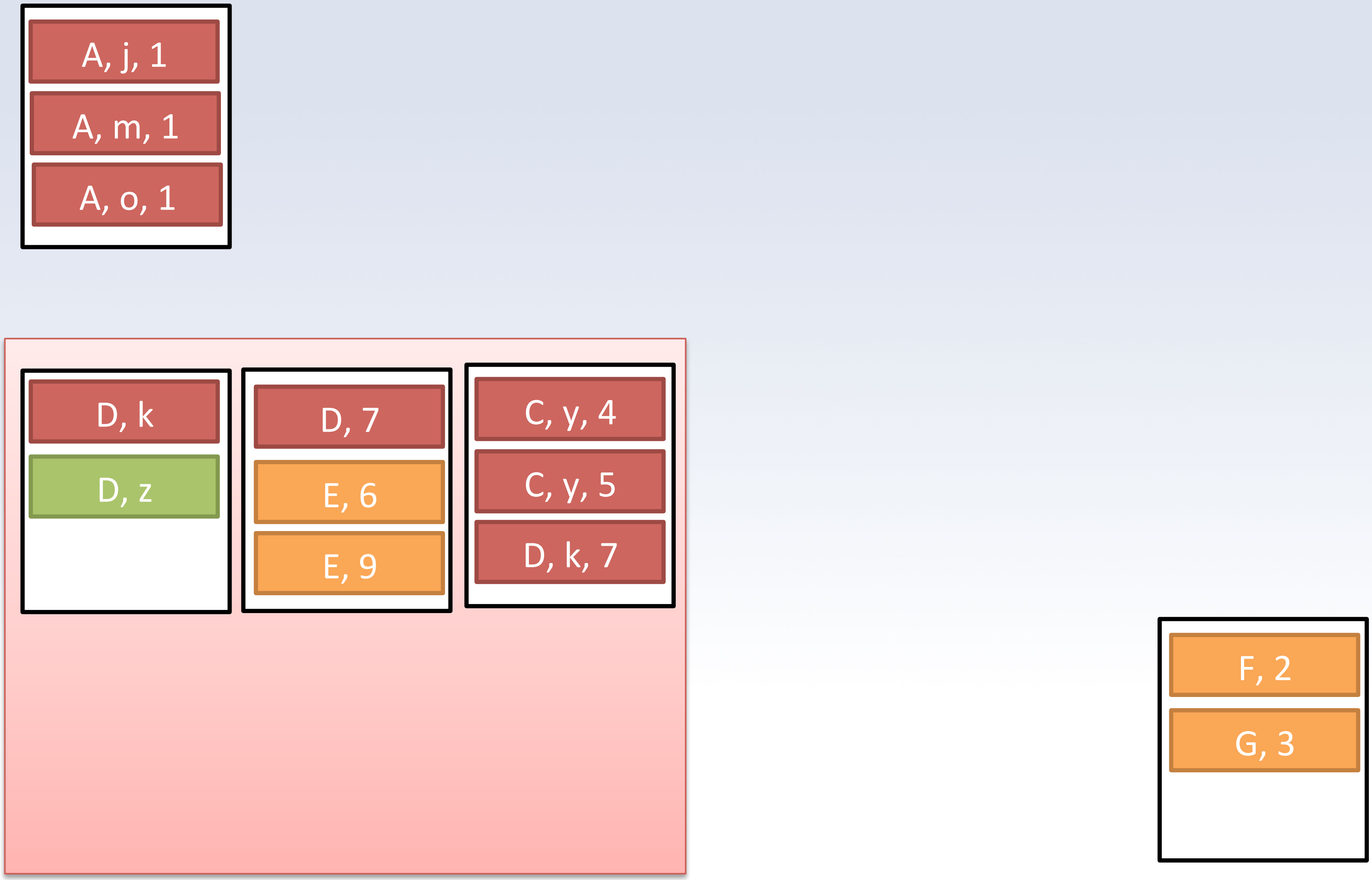
Example R⋉S



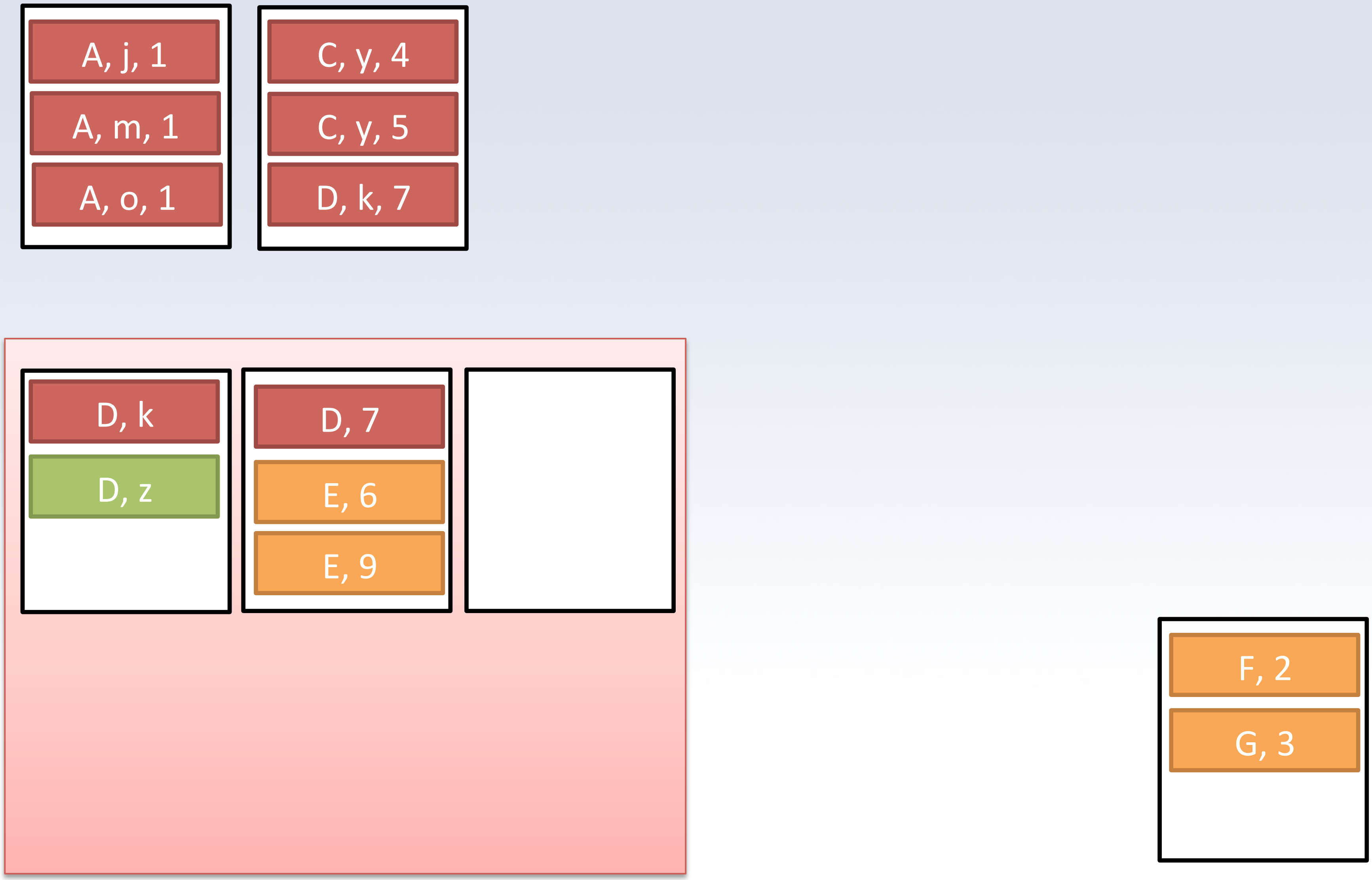
Example R⋈S



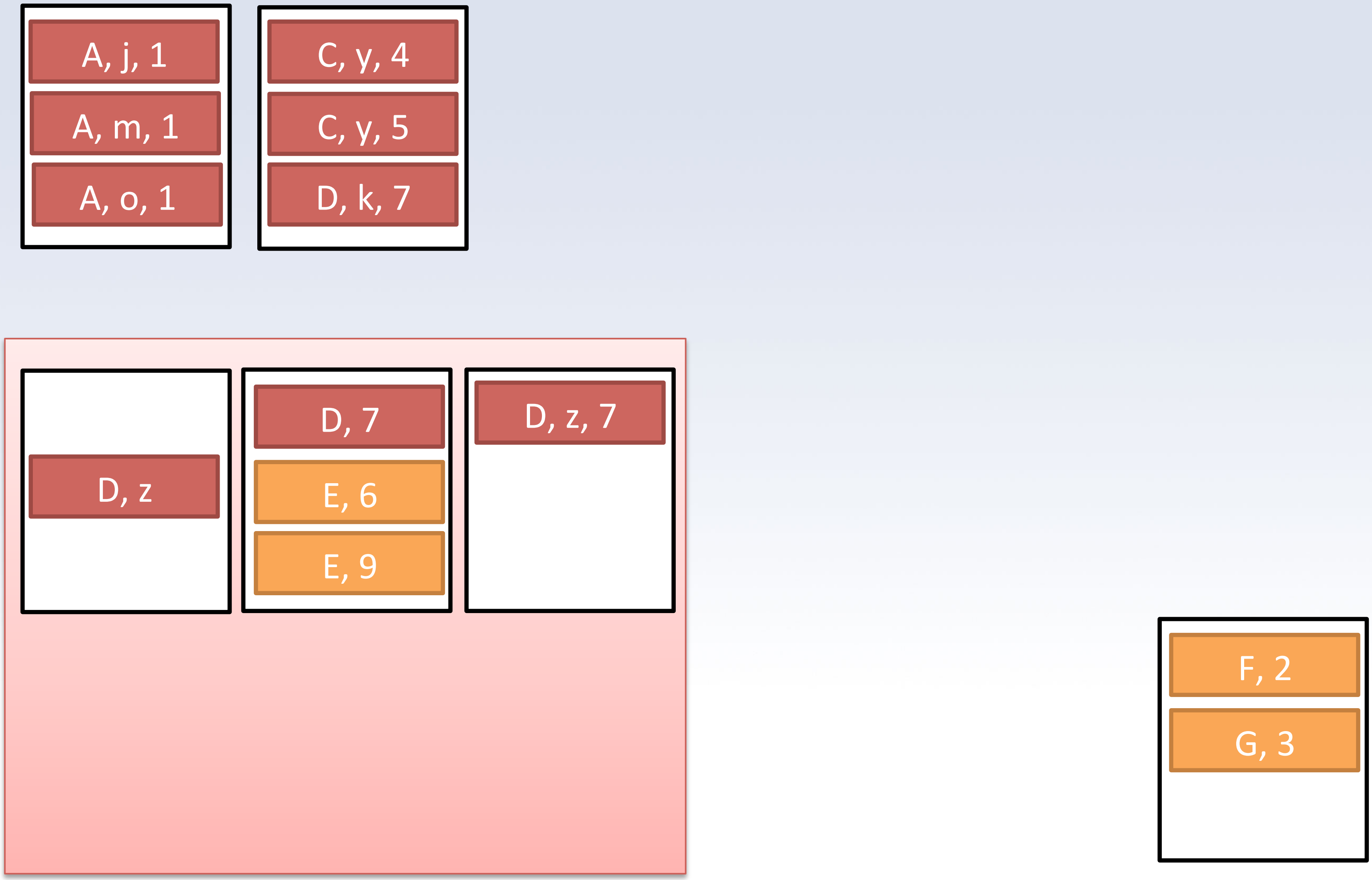
Example R⋈S



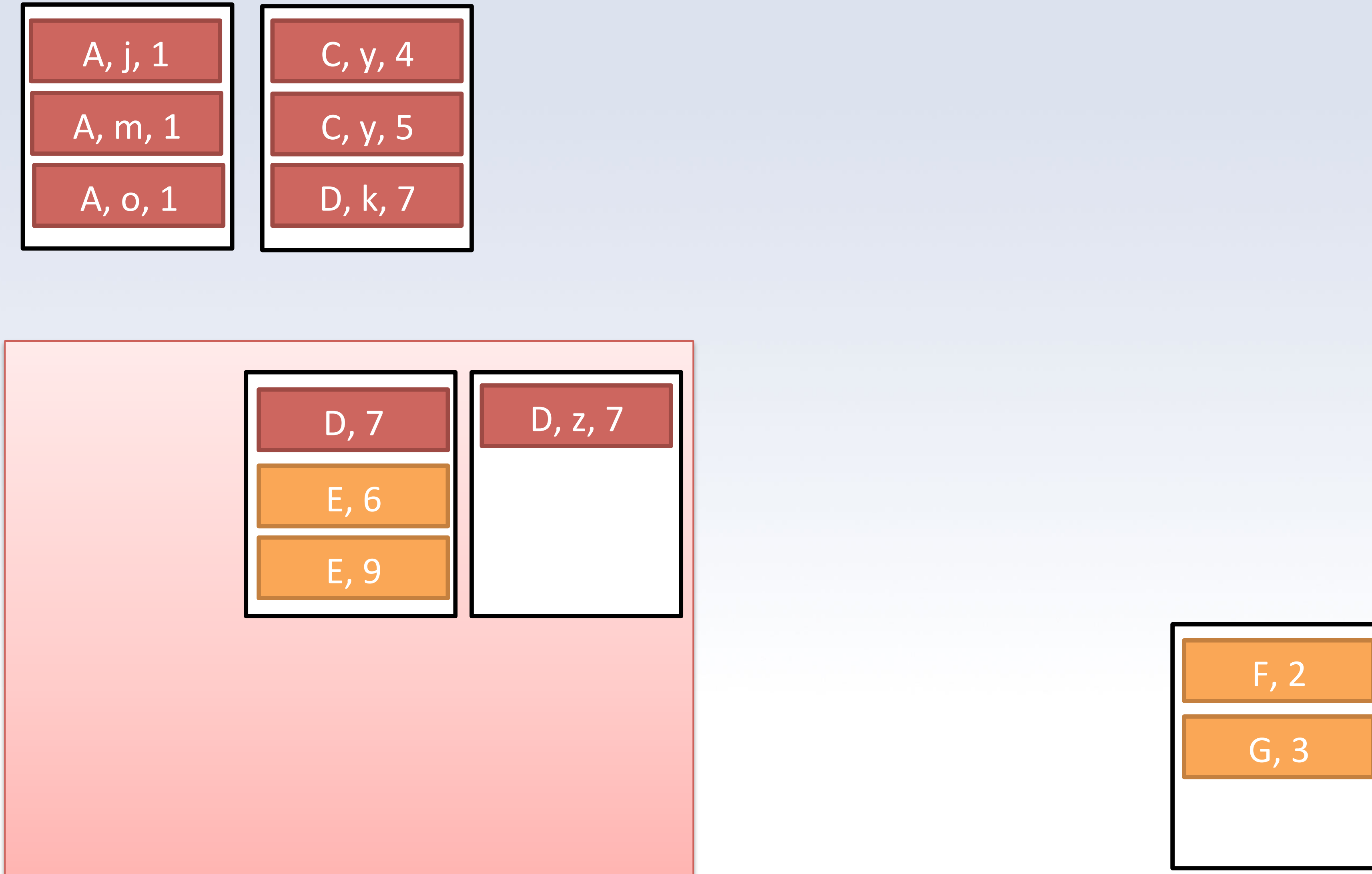
Example R⋈S



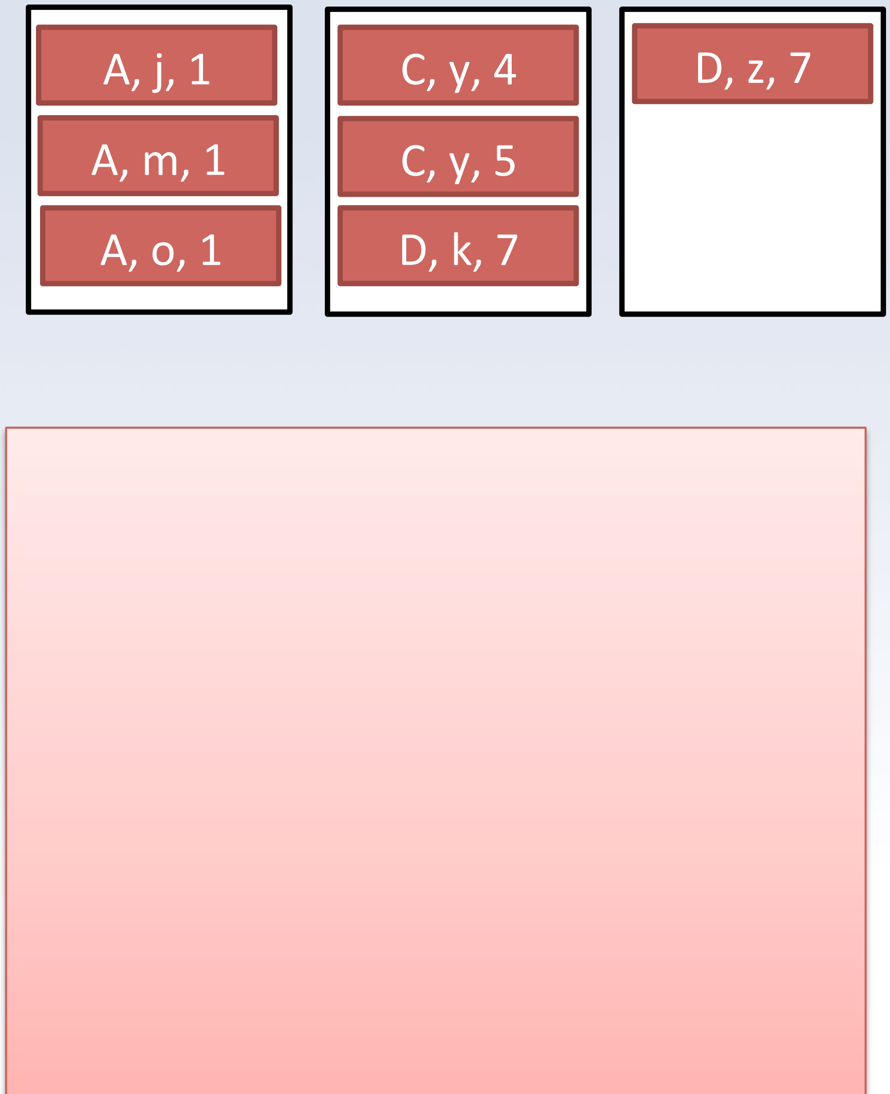
Example R⋈S



Example R⋈S



Example R⋈S



Analysis

- Requires both $B(R)<M^2$ and $B(S)<M^2$
- Shared join attributes must fit in memory, so $\max(B(R),B(S))<M^2$
- Requires $5(B(R)+B(S))$ I/O operations
  - Two reads and writes for first pass
  - Two reads and writes for second pass
  - One read and write for joining relations

Sort Join

- We can do better:
  - Assume all records with common join attributes fit in memory
  - Build the join into the second phase
  - Create sorted sublists using join attributes
  - Bring smallest block for each sublist into memory and perform joins

Sort Join

- Requires all records with same join attribute fit in memory
  - $(B(R)+B(S))<M^2$
- Only requires two passes:
  - $3(B(R)+B(S))$ I/O operations

Sorting-Based Summary

Operator	M required	I/O Cost
δ, γ	√B	3B
∪ , ∩ , -	√(B(R)+B(S))	3(B(R)+B(S))
⋈	√(max(B(R),B(S))	5(B(R)+B(S))
⋈	√(B(R)+B(S))	3(B(R)+B(S))

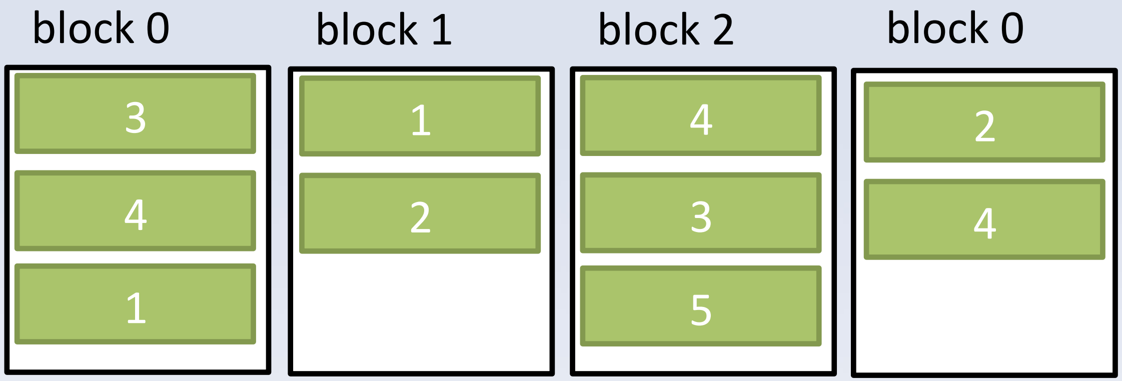
2 Pass Hash Based Algorithms

- We can also design two pass algorithms based on hashing
- First pass: hash the relation into M-1 buckets
- Second pass: process each bucket using one pass algorithms
  - requires that each bucket fits in memory

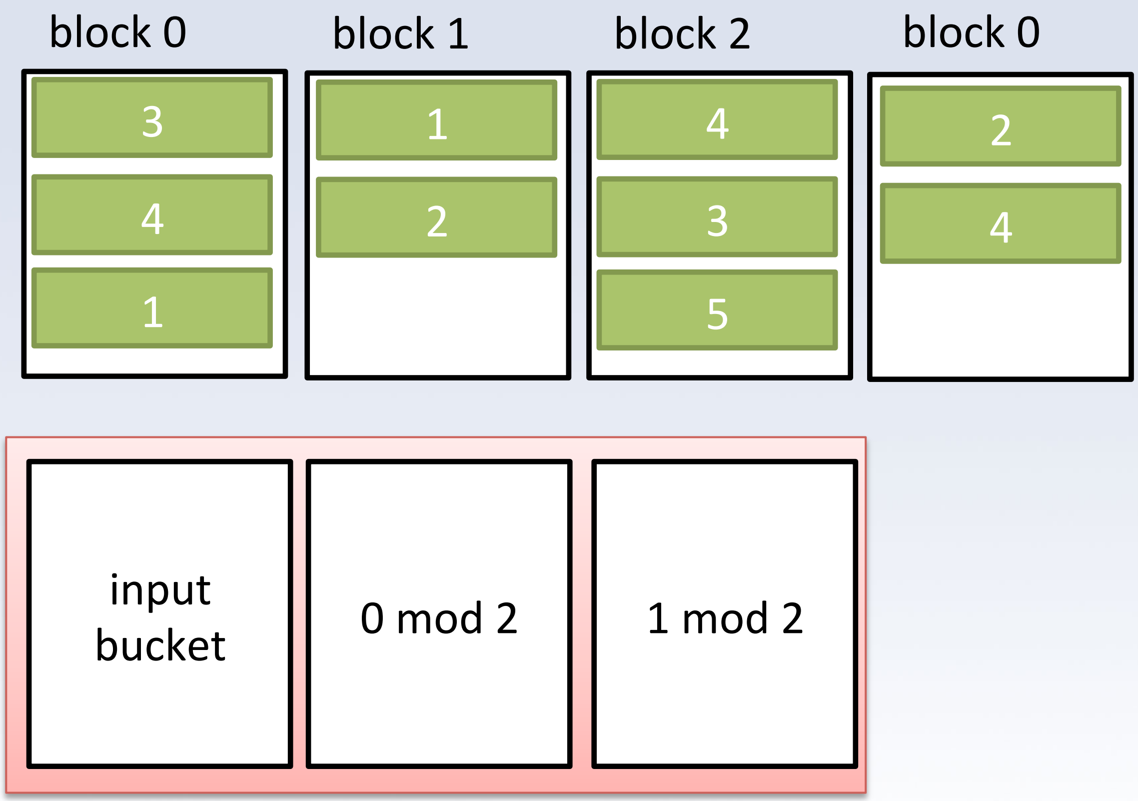
Two Pass Hashing for δ

- Hash the relation into M-1 buckets
- Identical records should hash to the same bucket
- Bring each bucket into memory and output each tuple only once

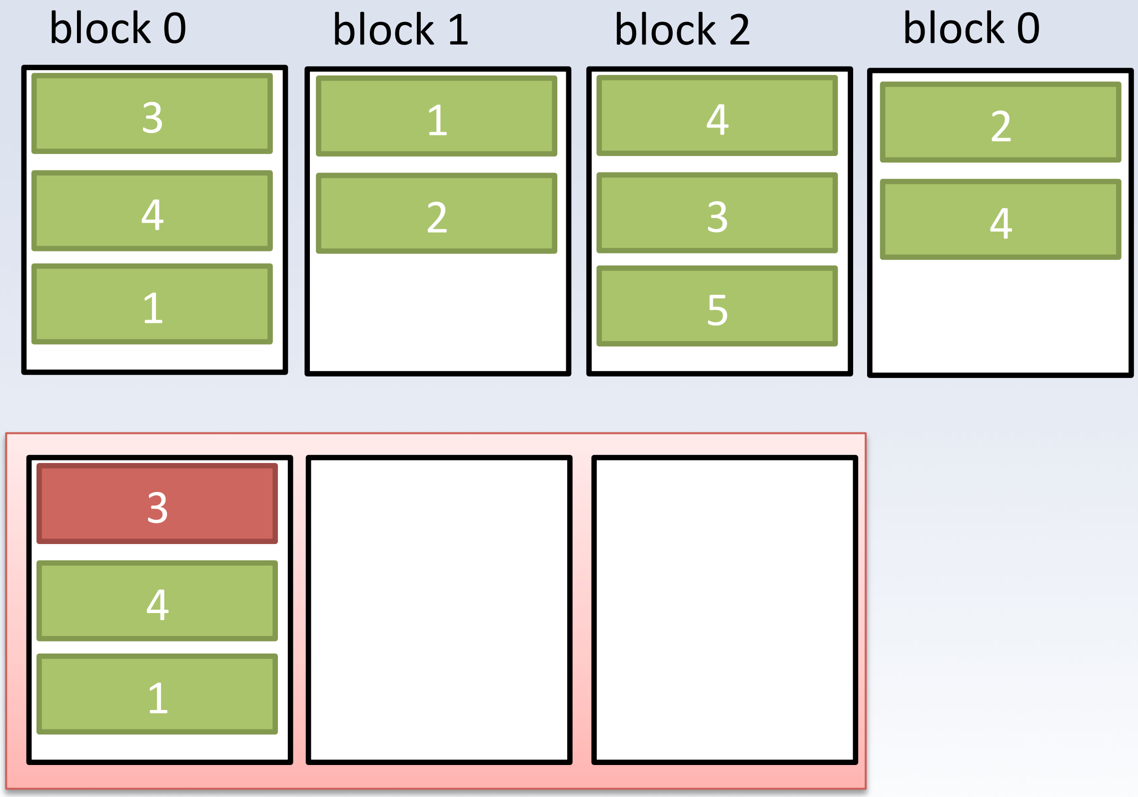
Two Pass Hashing for δ



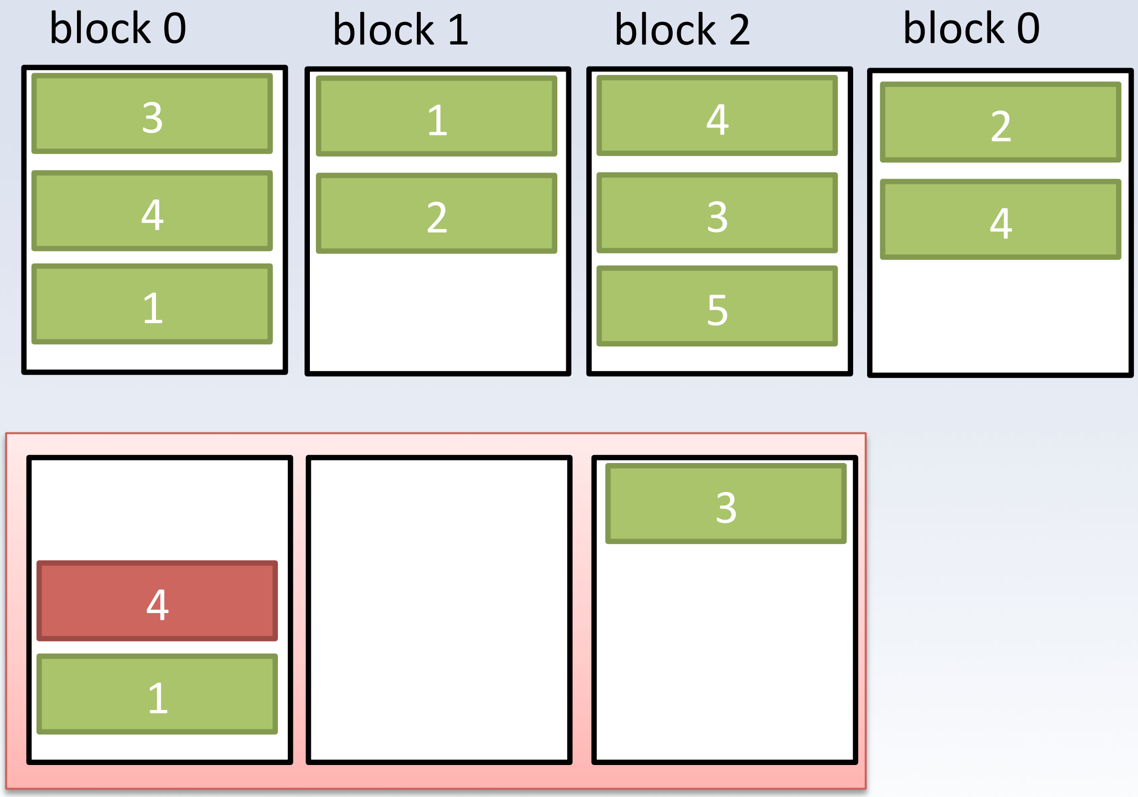
Two Pass Hashing for δ



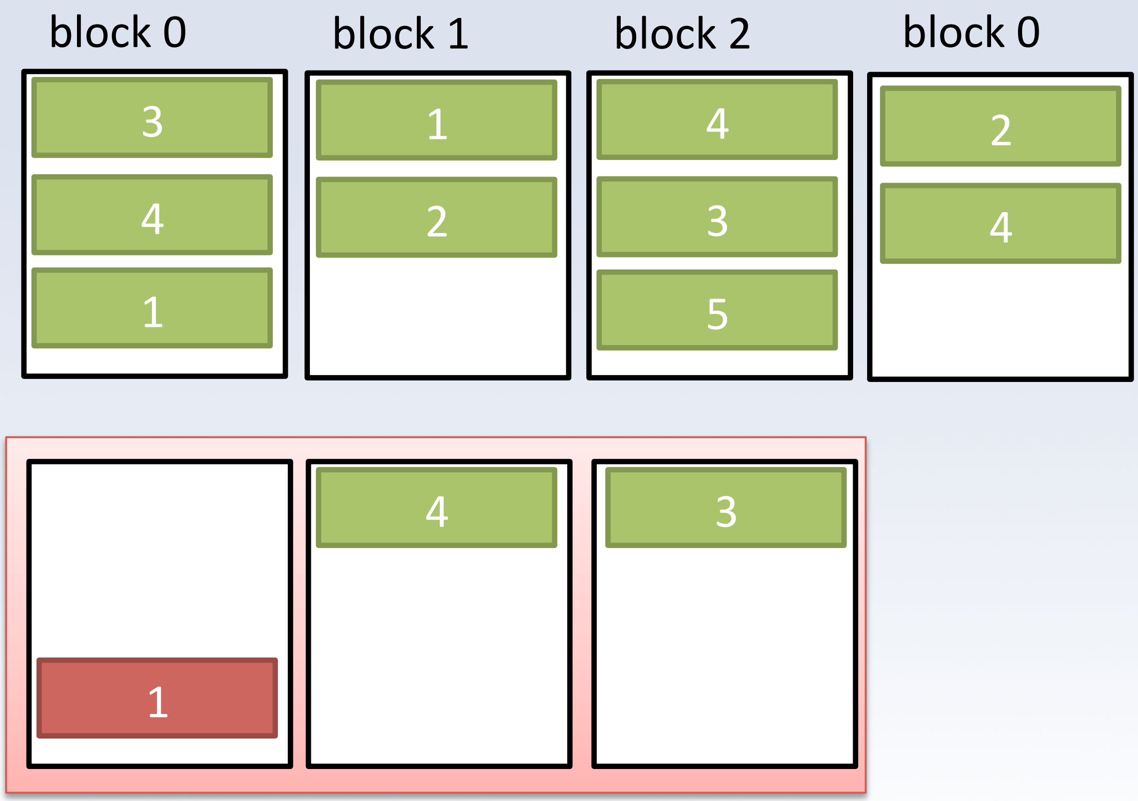
Two Pass Hashing for δ



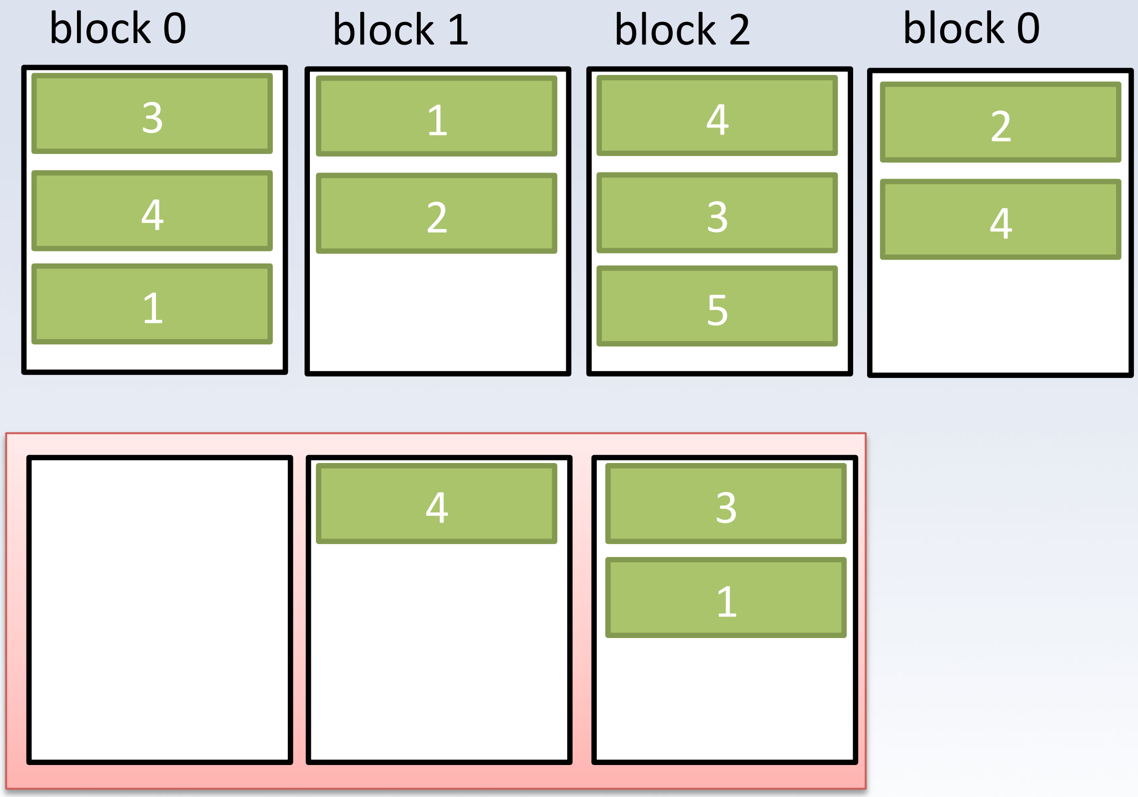
Two Pass Hashing for δ



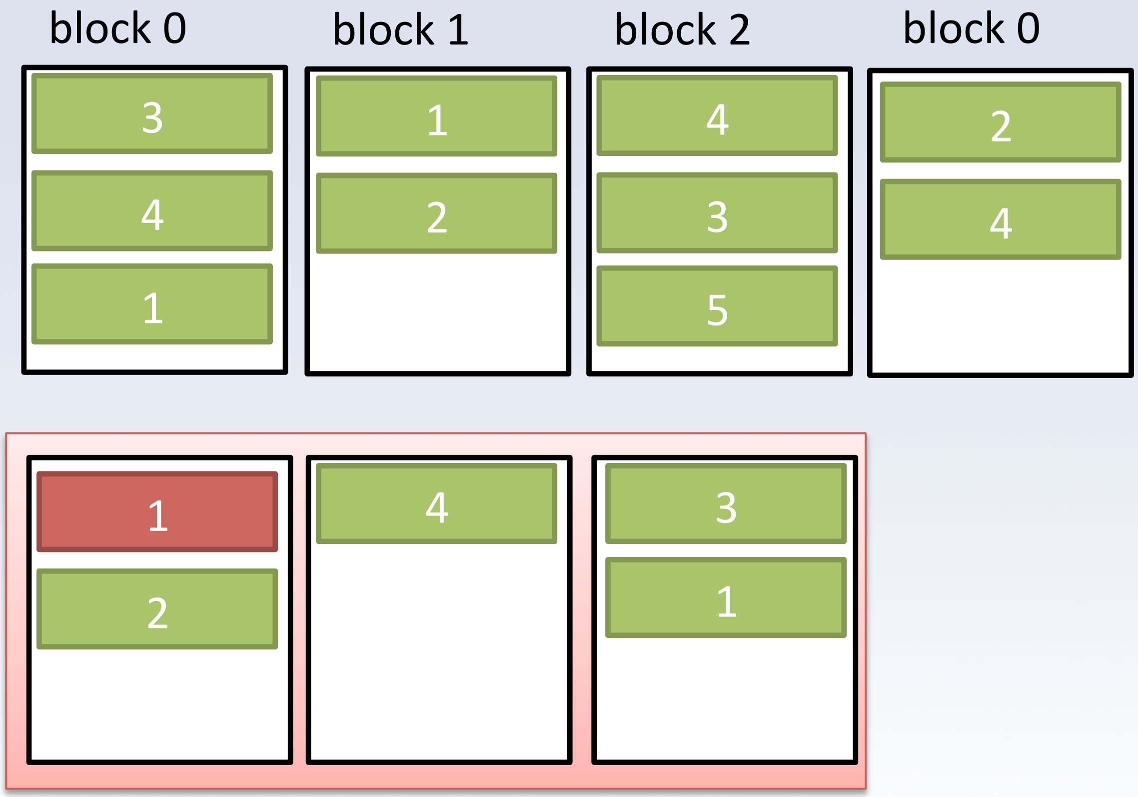
Two Pass Hashing for δ



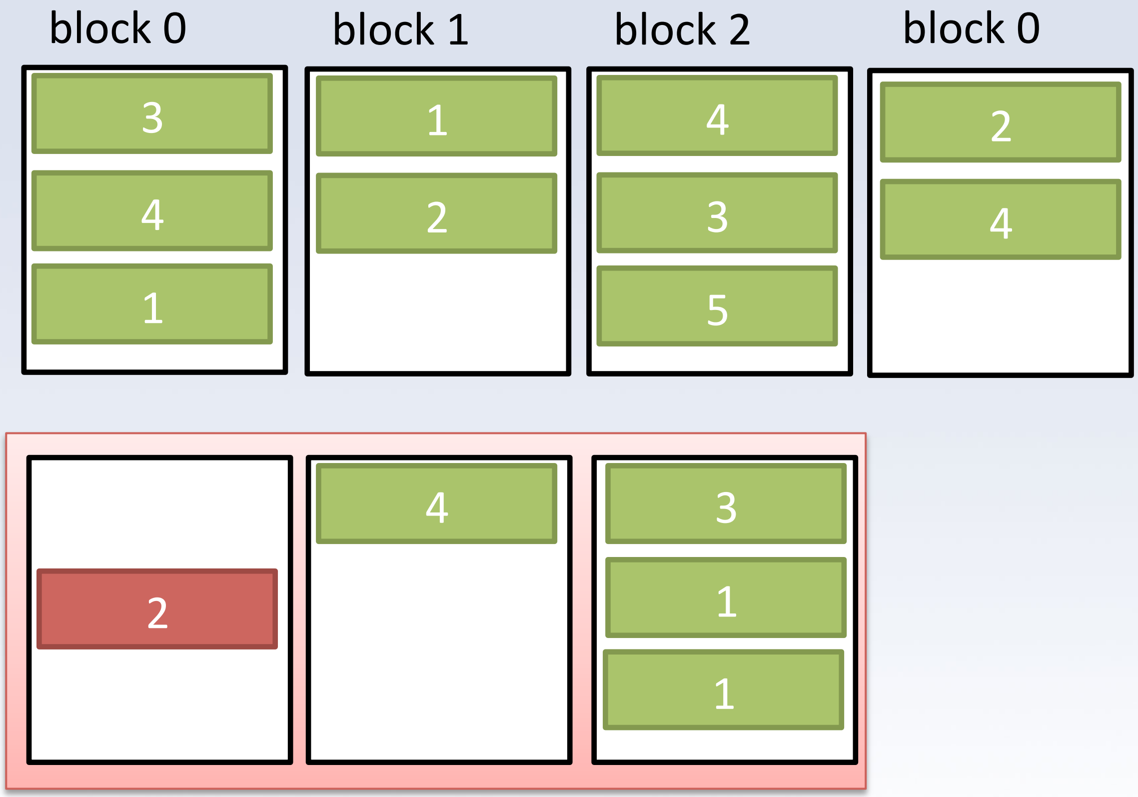
Two Pass Hashing for δ



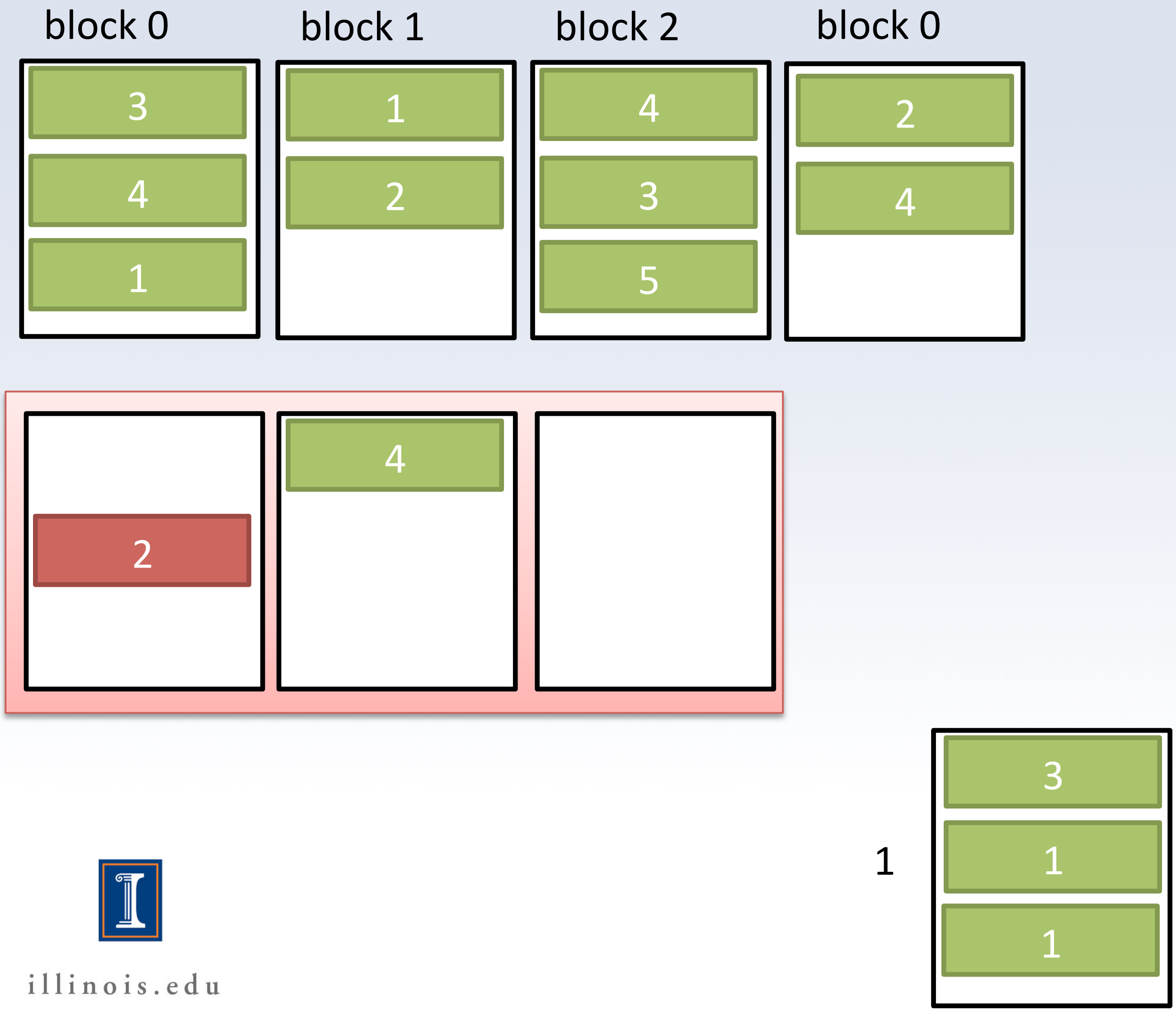
Two Pass Hashing for δ



Two Pass Hashing for δ

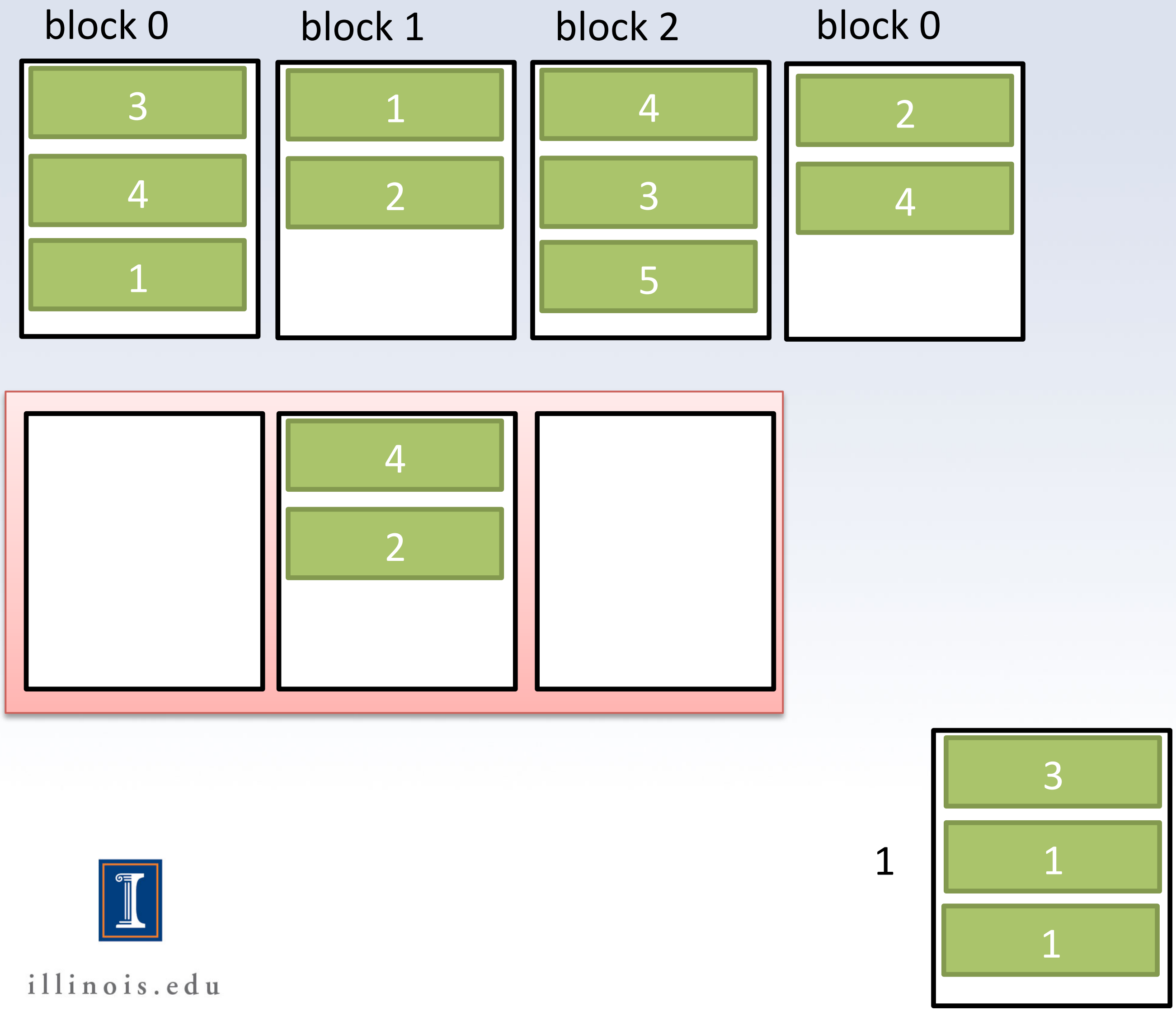


Two Pass Hashing for δ



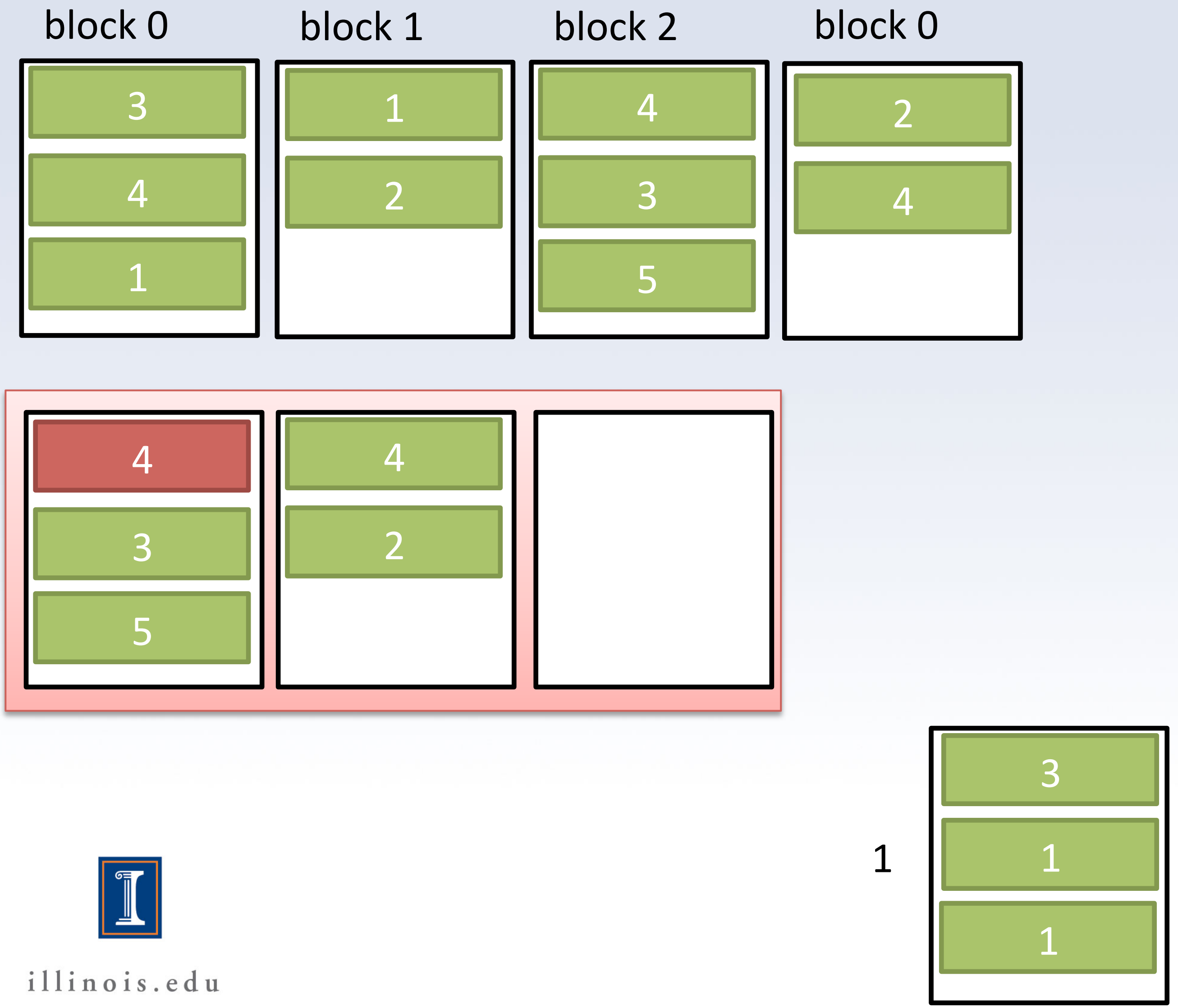
illinois.edu

Two Pass Hashing for δ



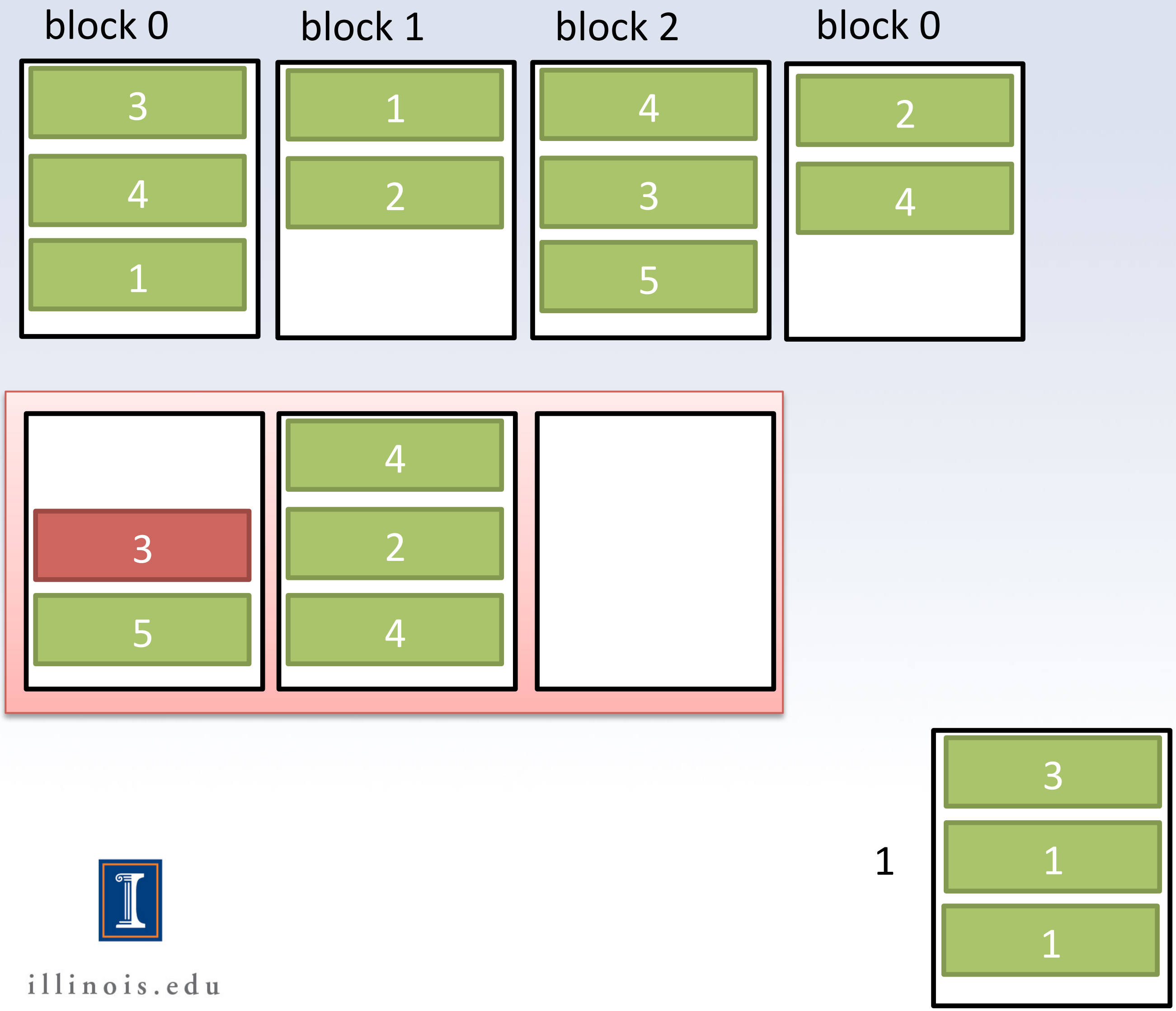
illinois.edu

Two Pass Hashing for δ

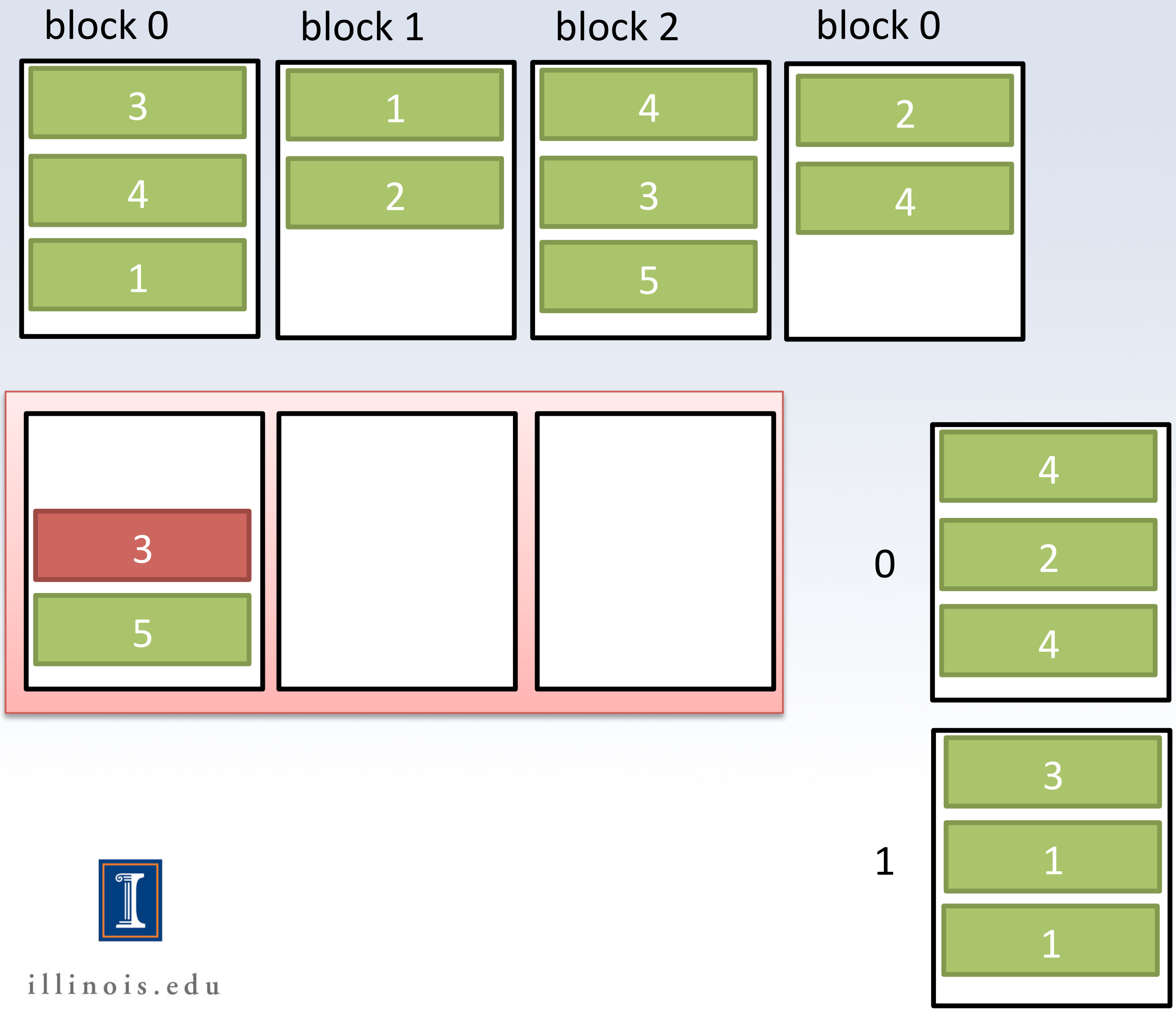


illinois.edu

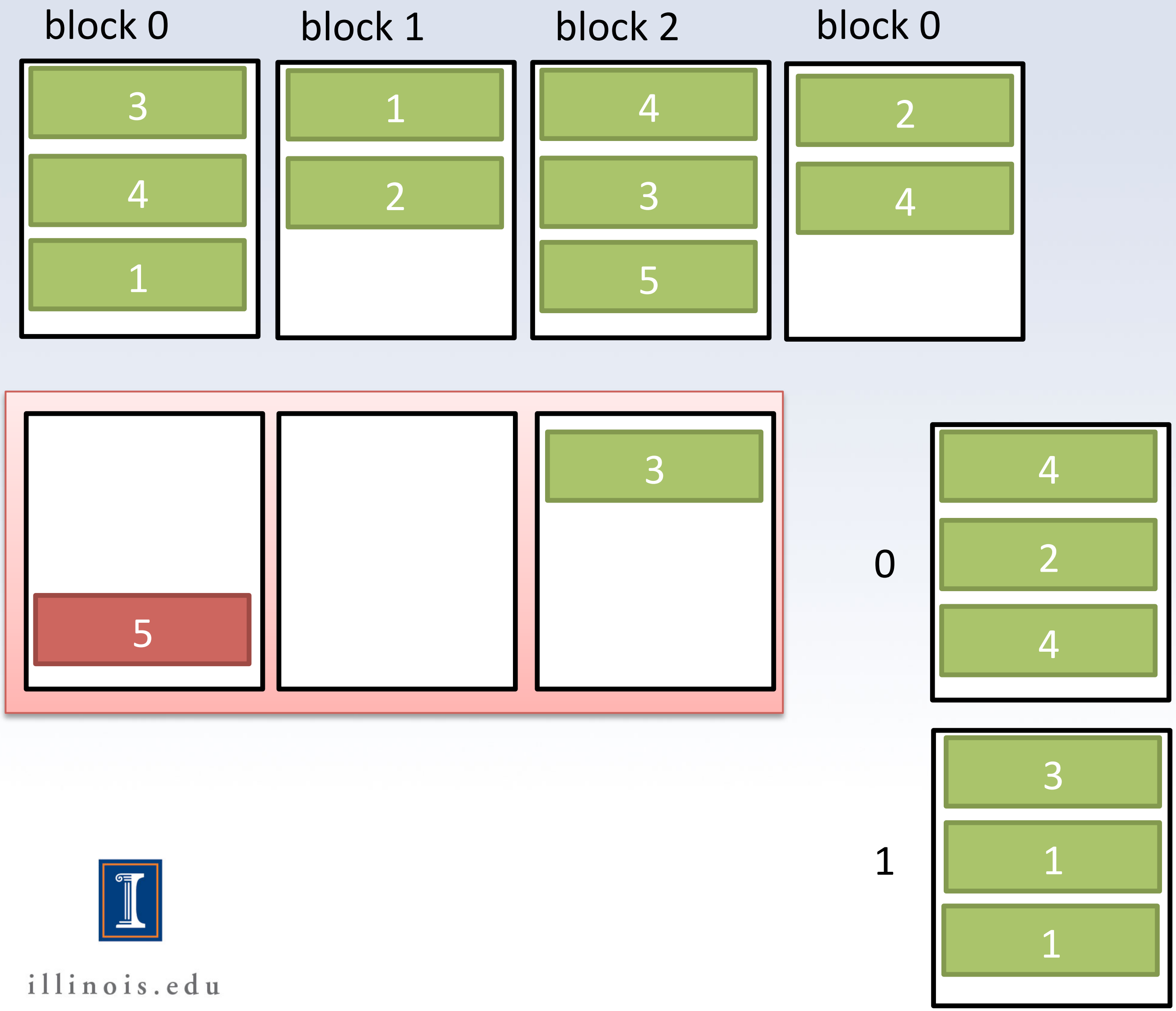
Two Pass Hashing for δ



Two Pass Hashing for δ

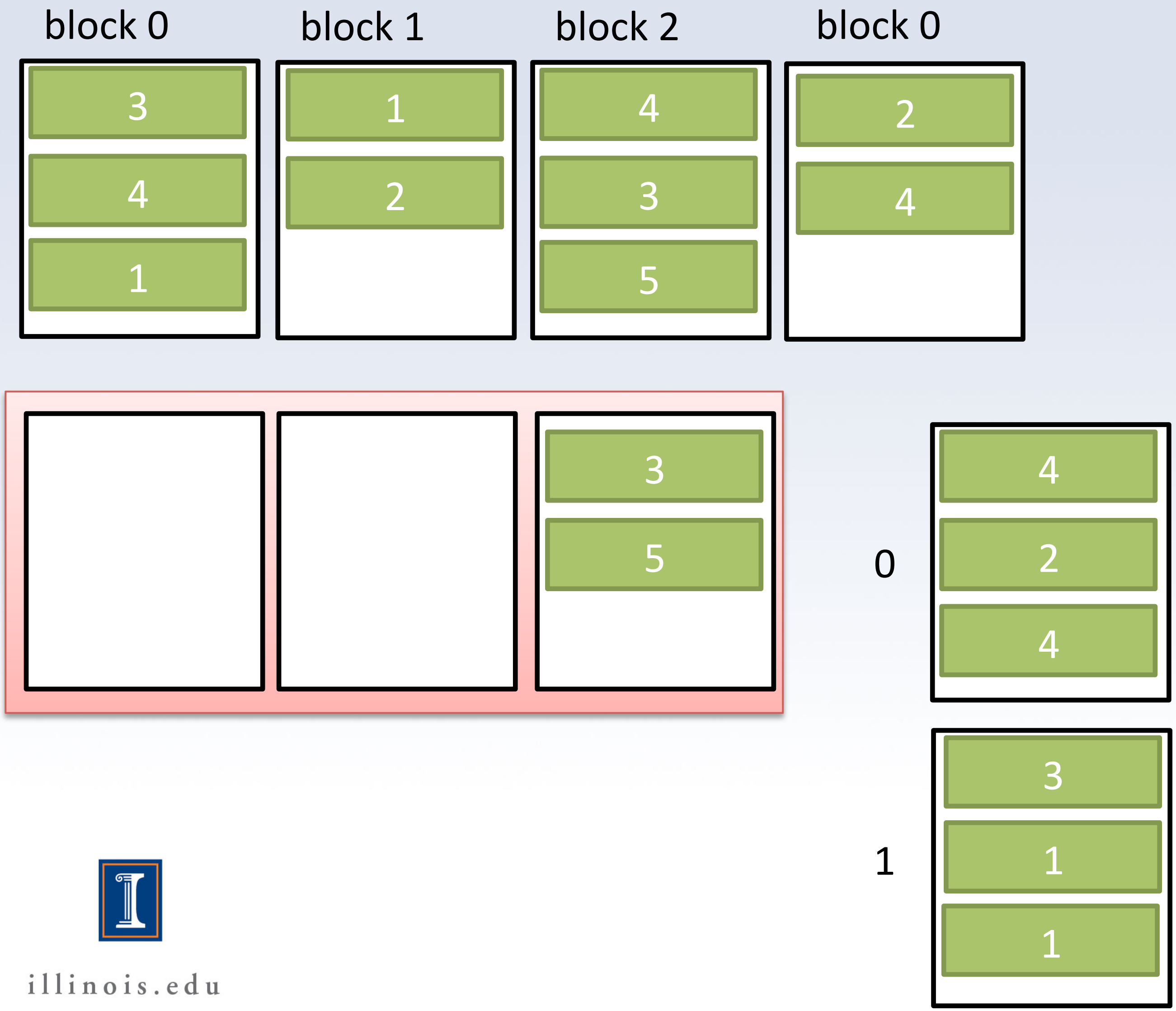


Two Pass Hashing for δ



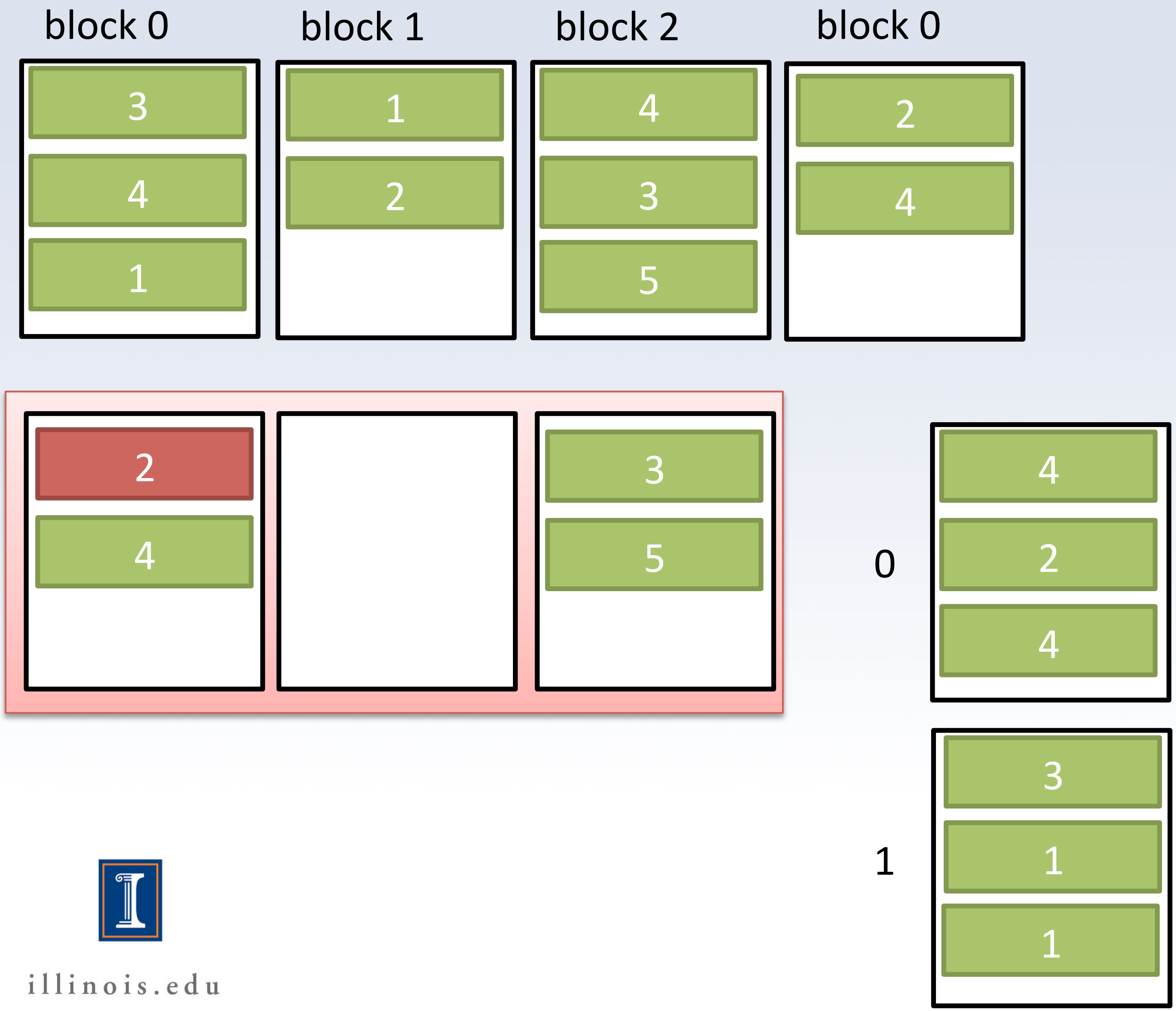
illinois.edu

Two Pass Hashing for δ

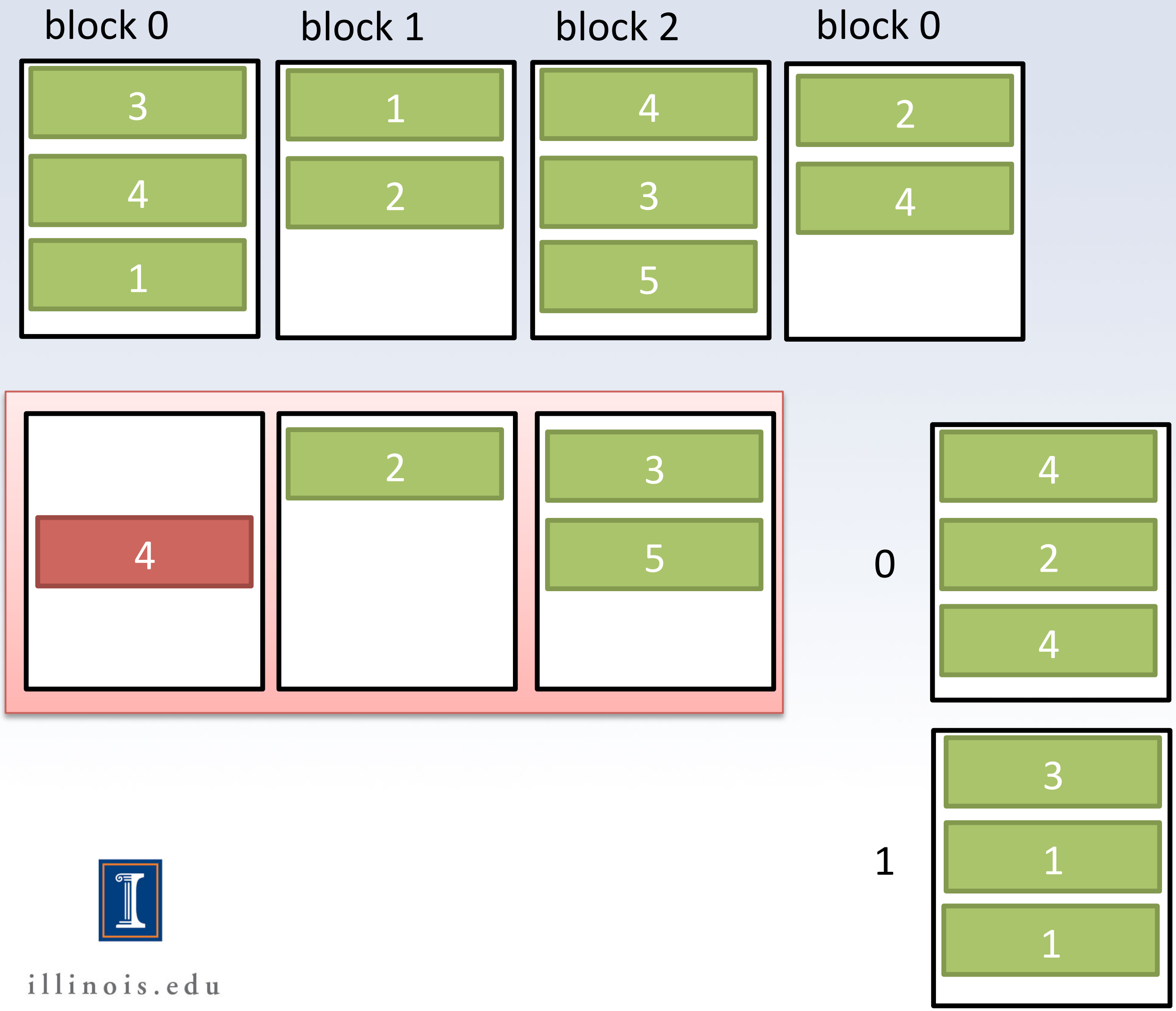


illinois.edu

Two Pass Hashing for δ

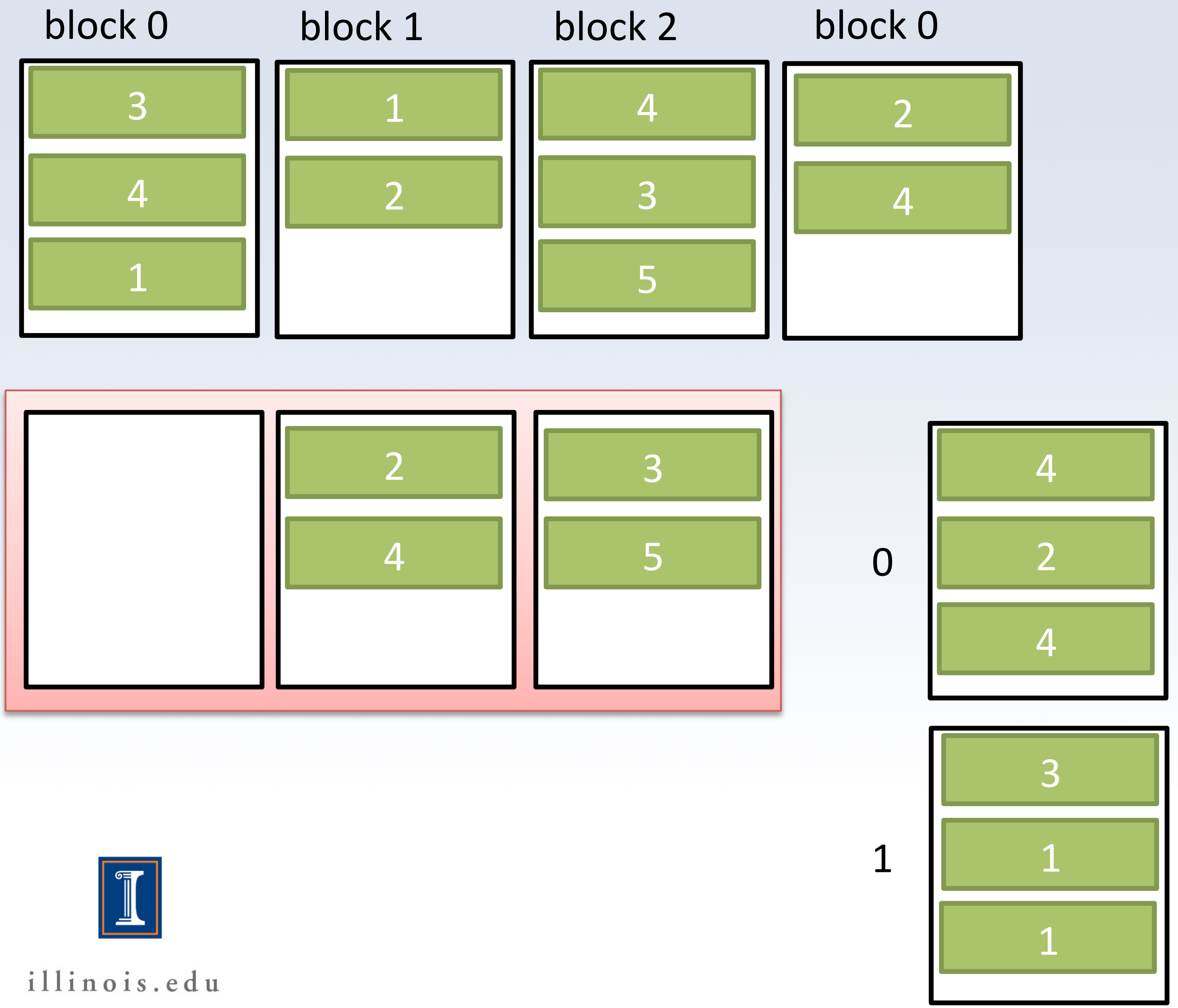


Two Pass Hashing for δ



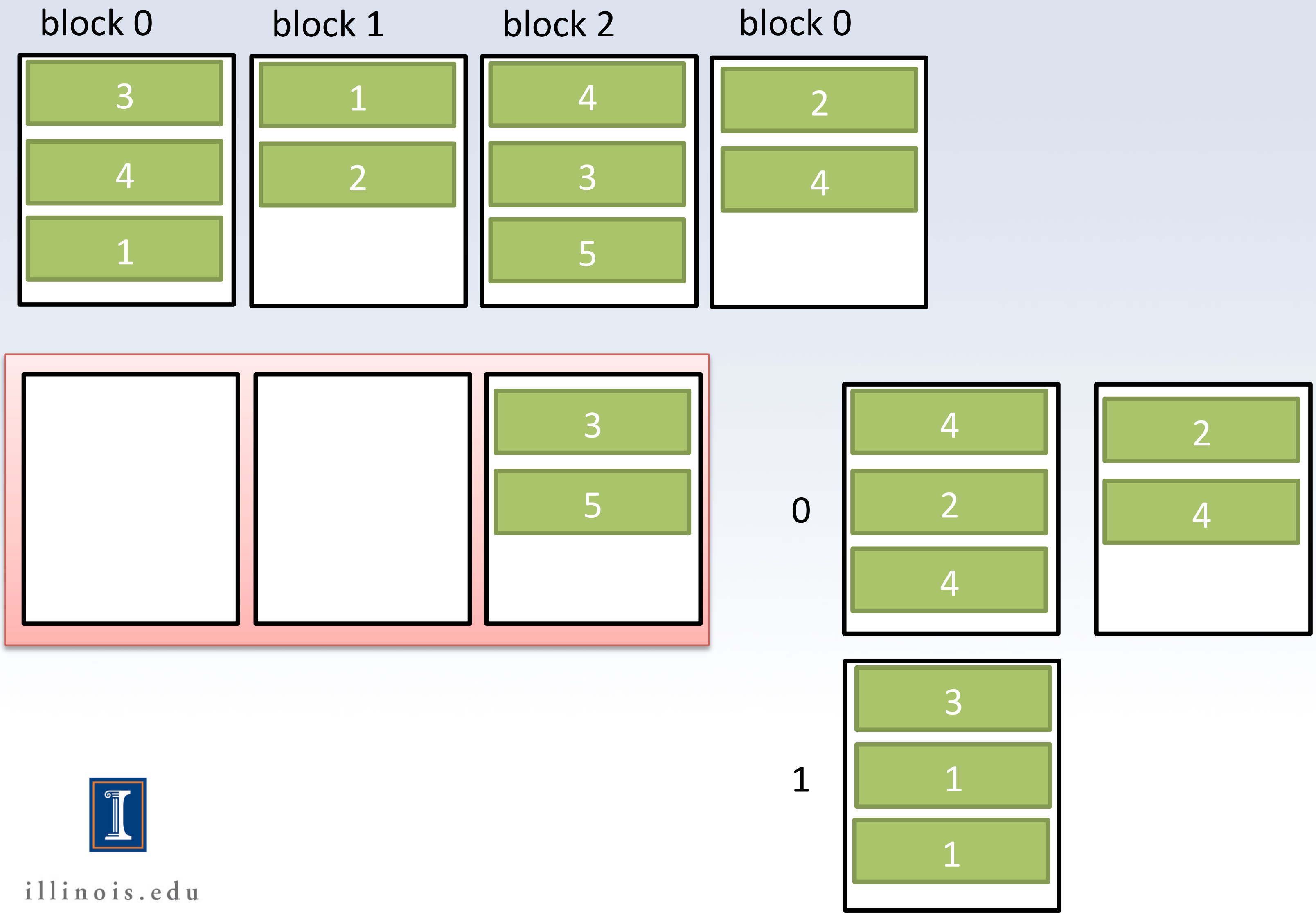
illinois.edu

Two Pass Hashing for δ



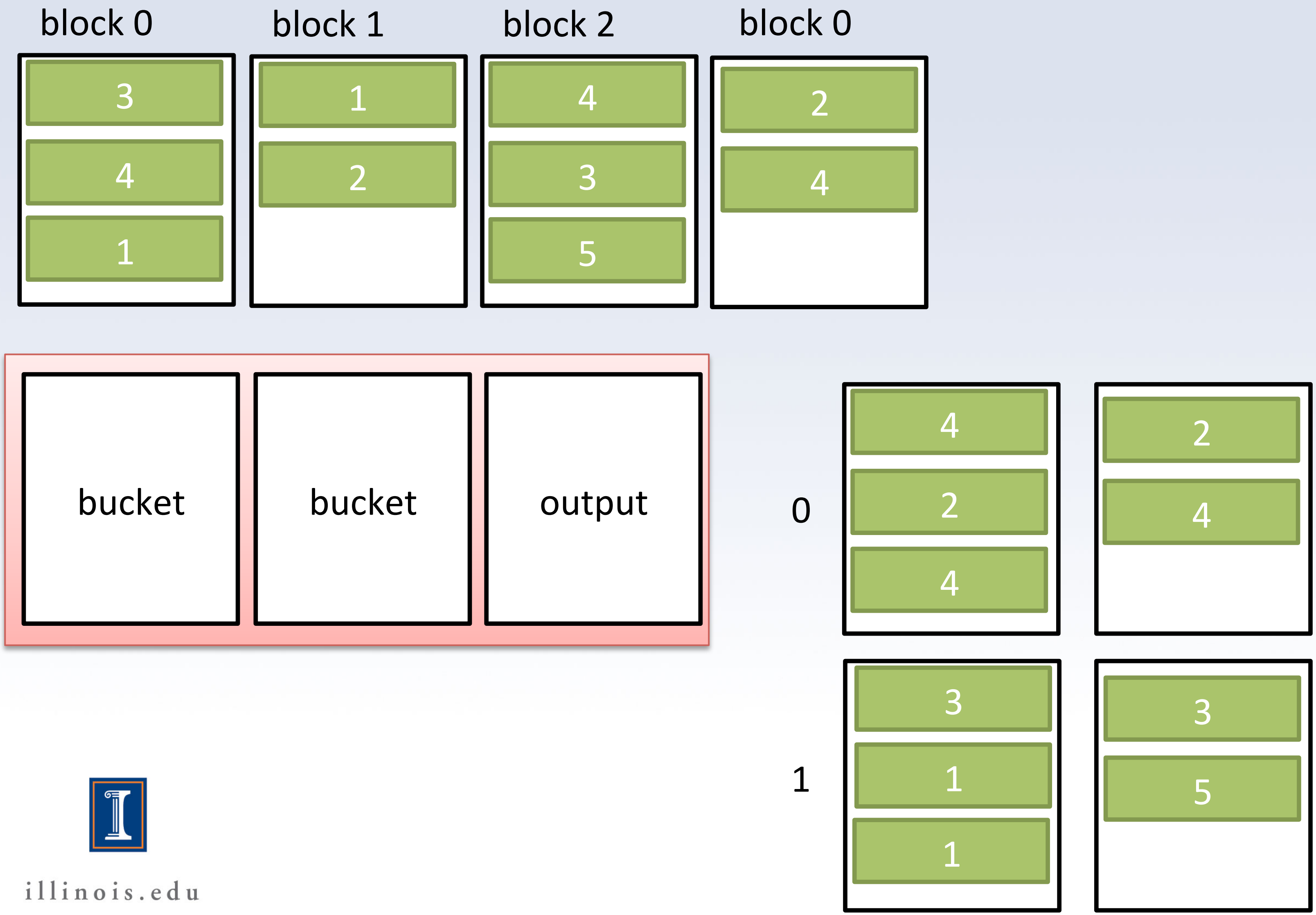
illinois.edu

Two Pass Hashing for δ

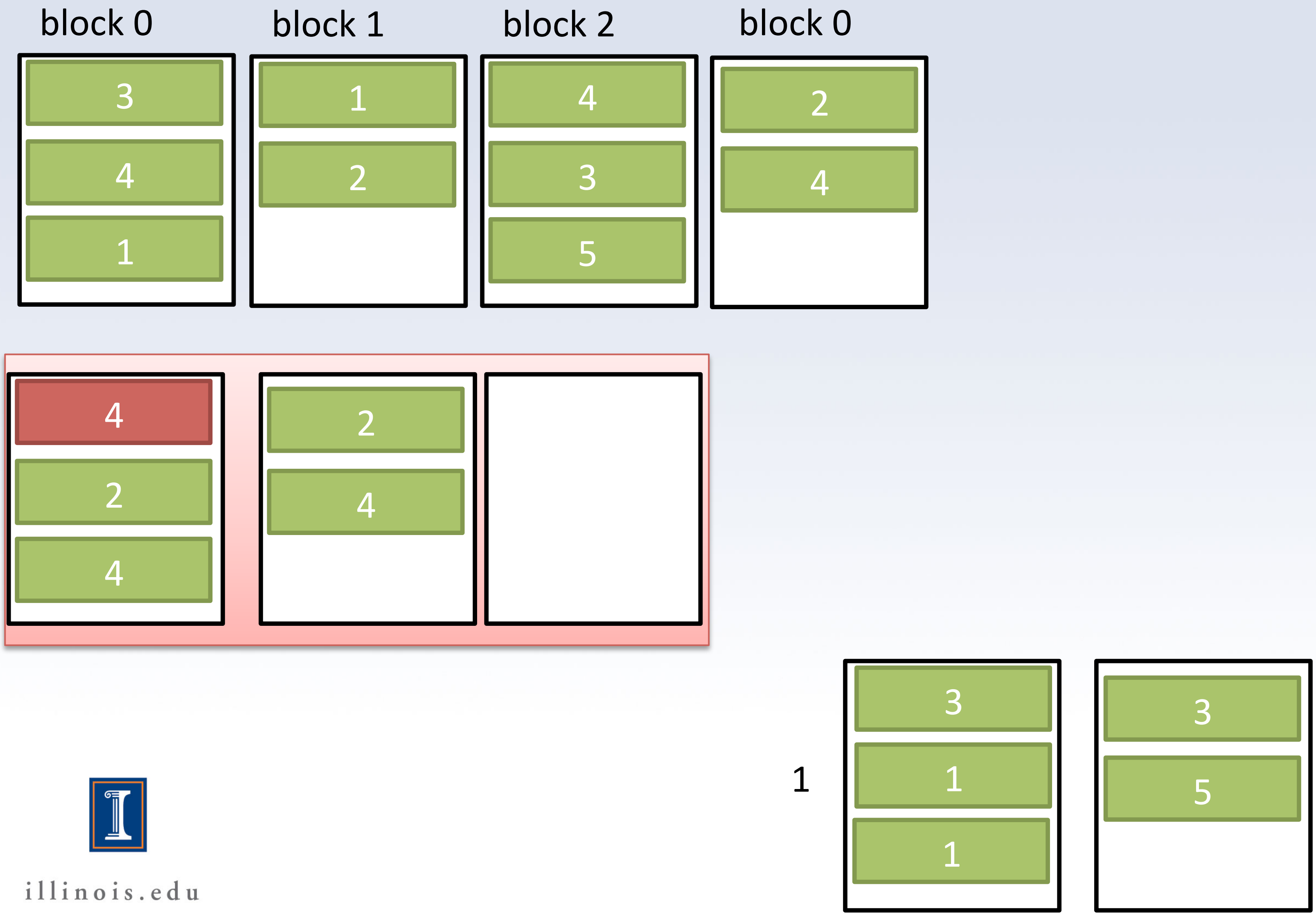


illinois.edu

Two Pass Hashing for δ

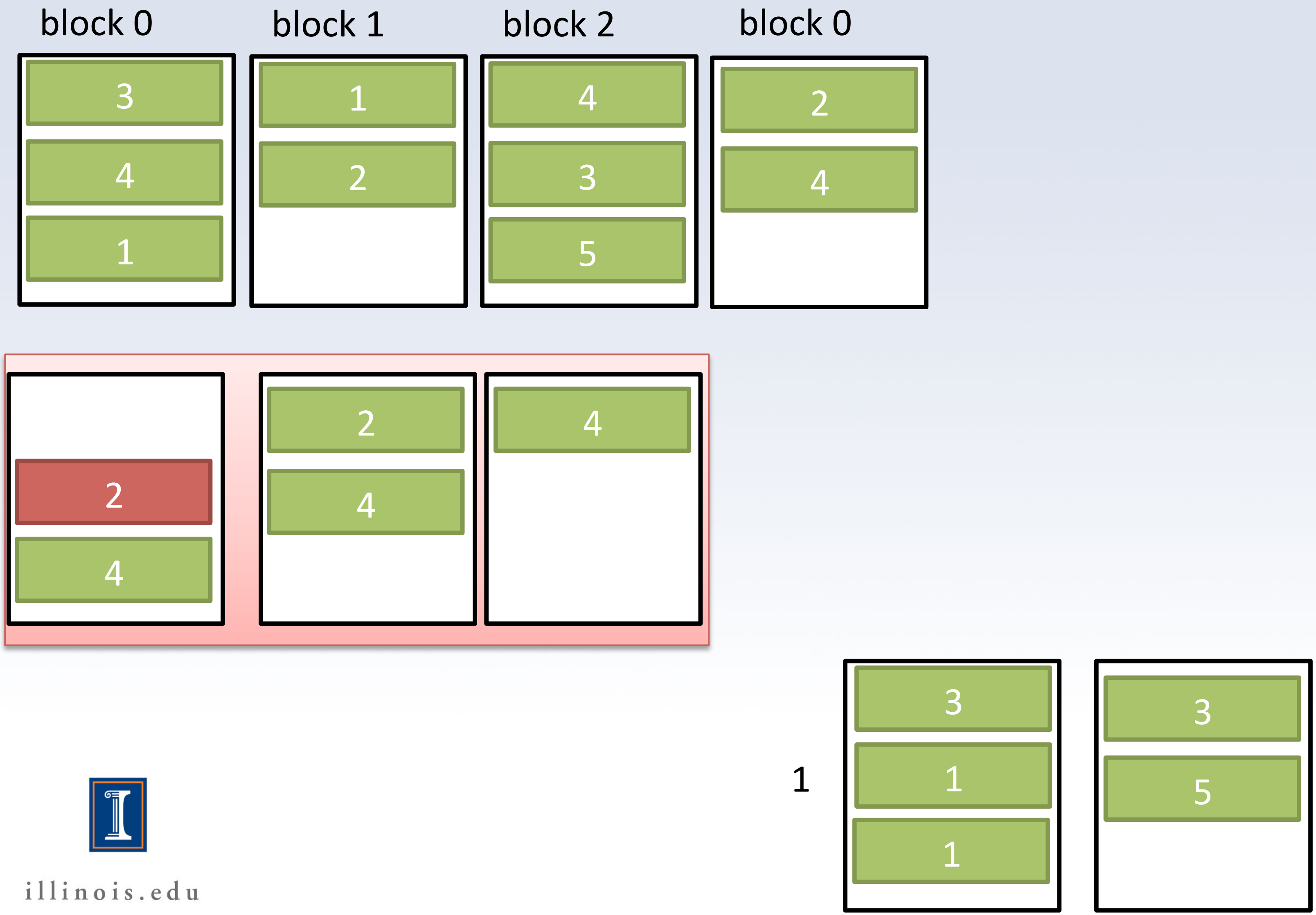


Two Pass Hashing for δ

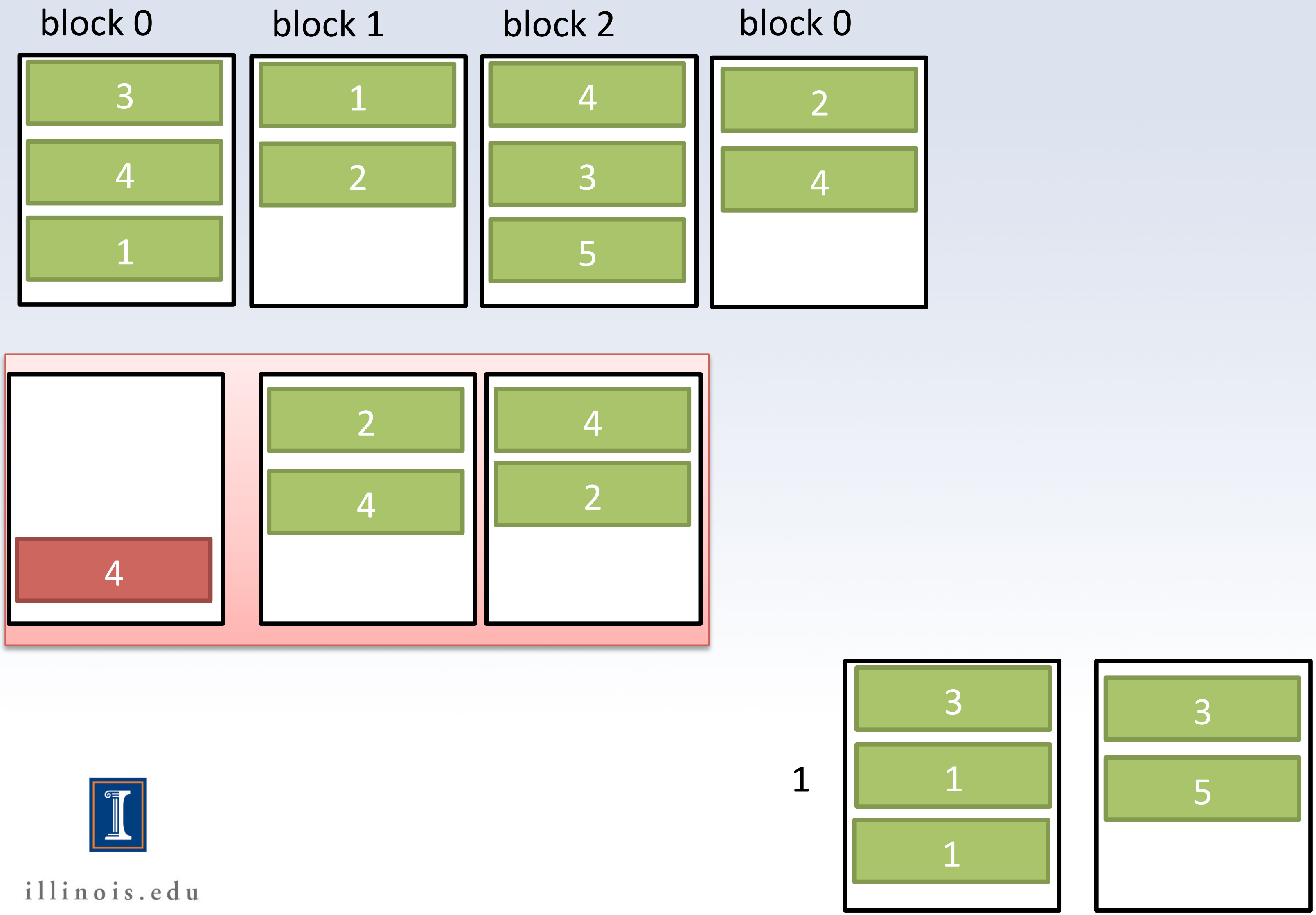


illinois.edu

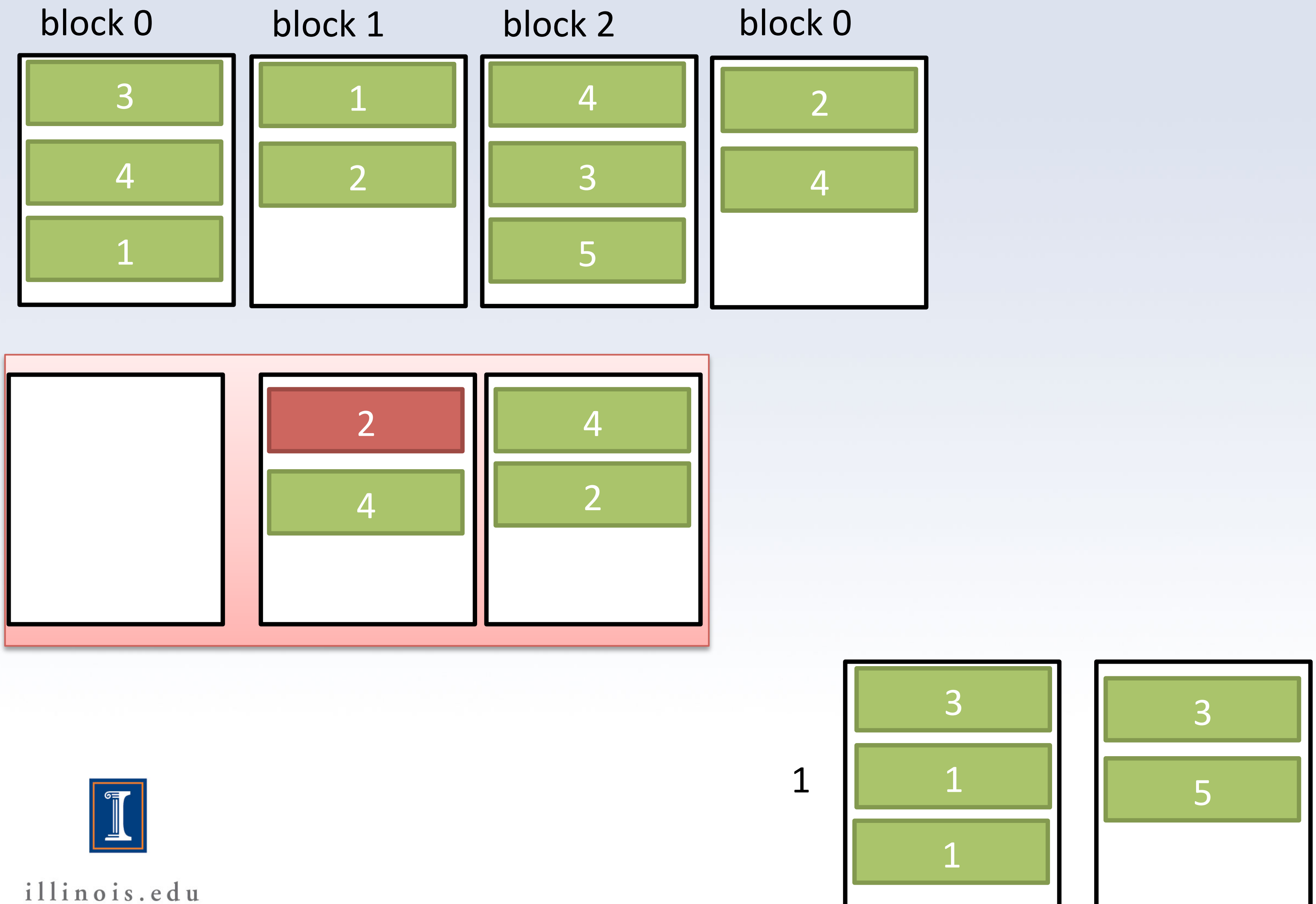
Two Pass Hashing for δ



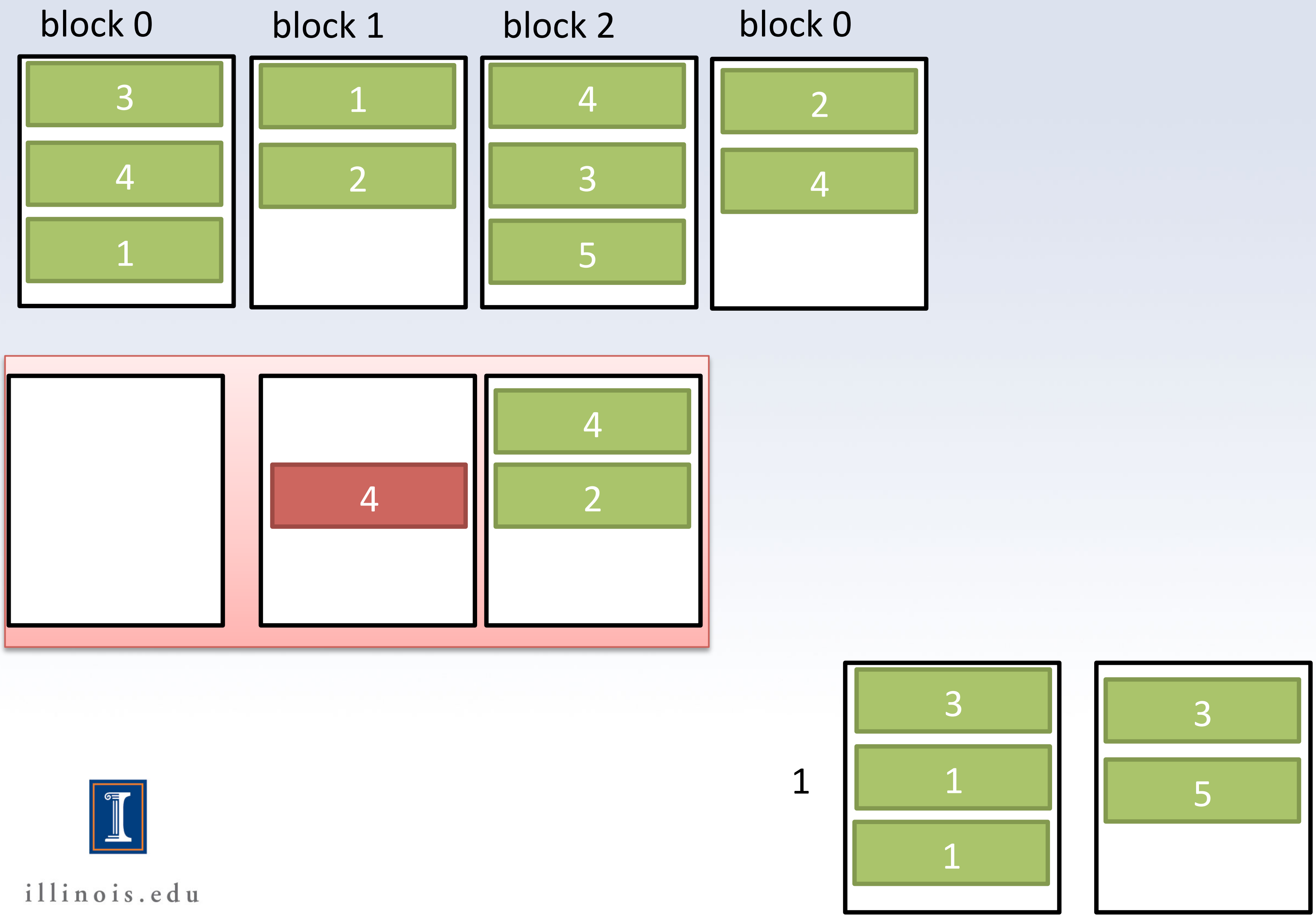
Two Pass Hashing for δ



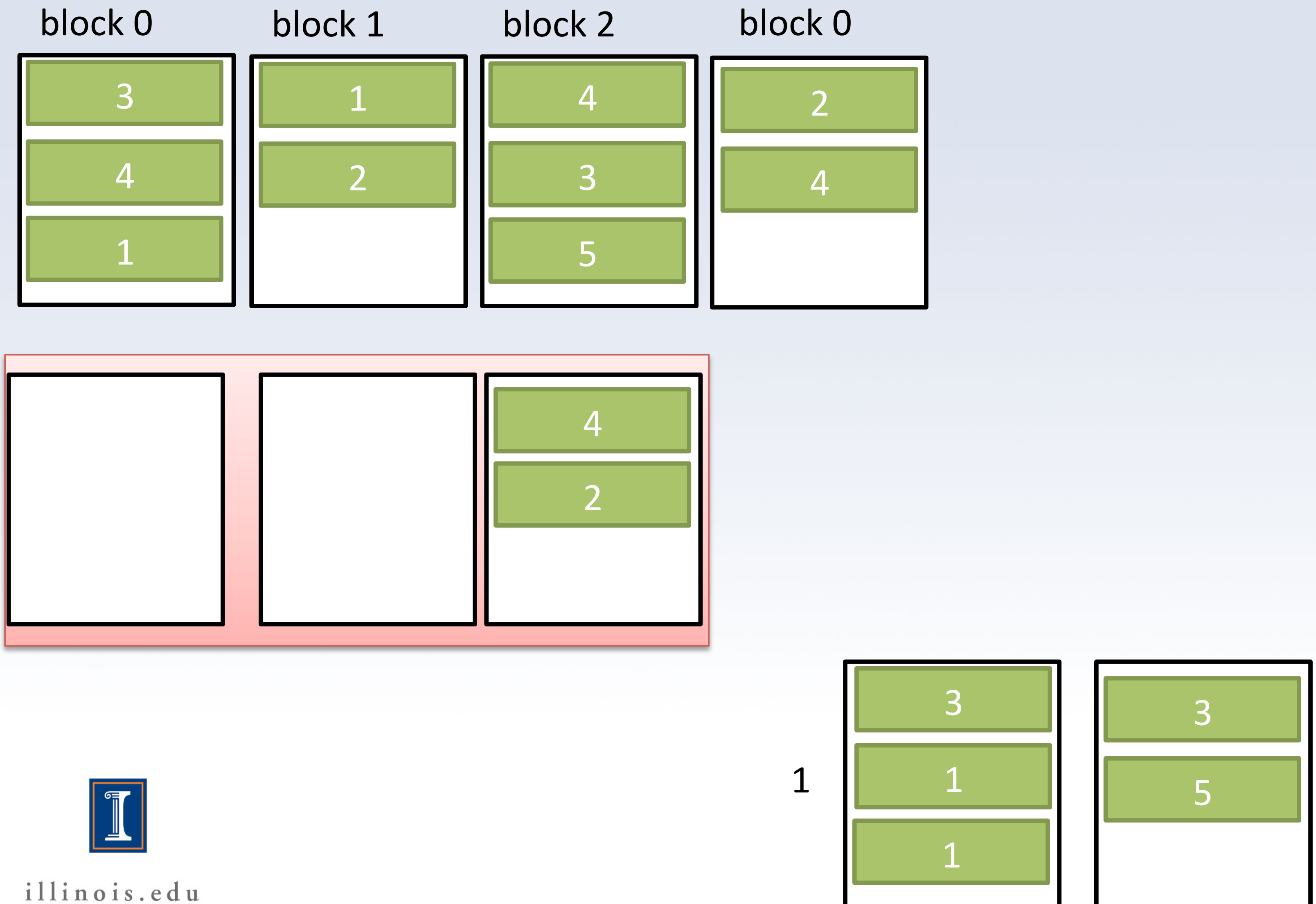
Two Pass Hashing for δ



Two Pass Hashing for δ

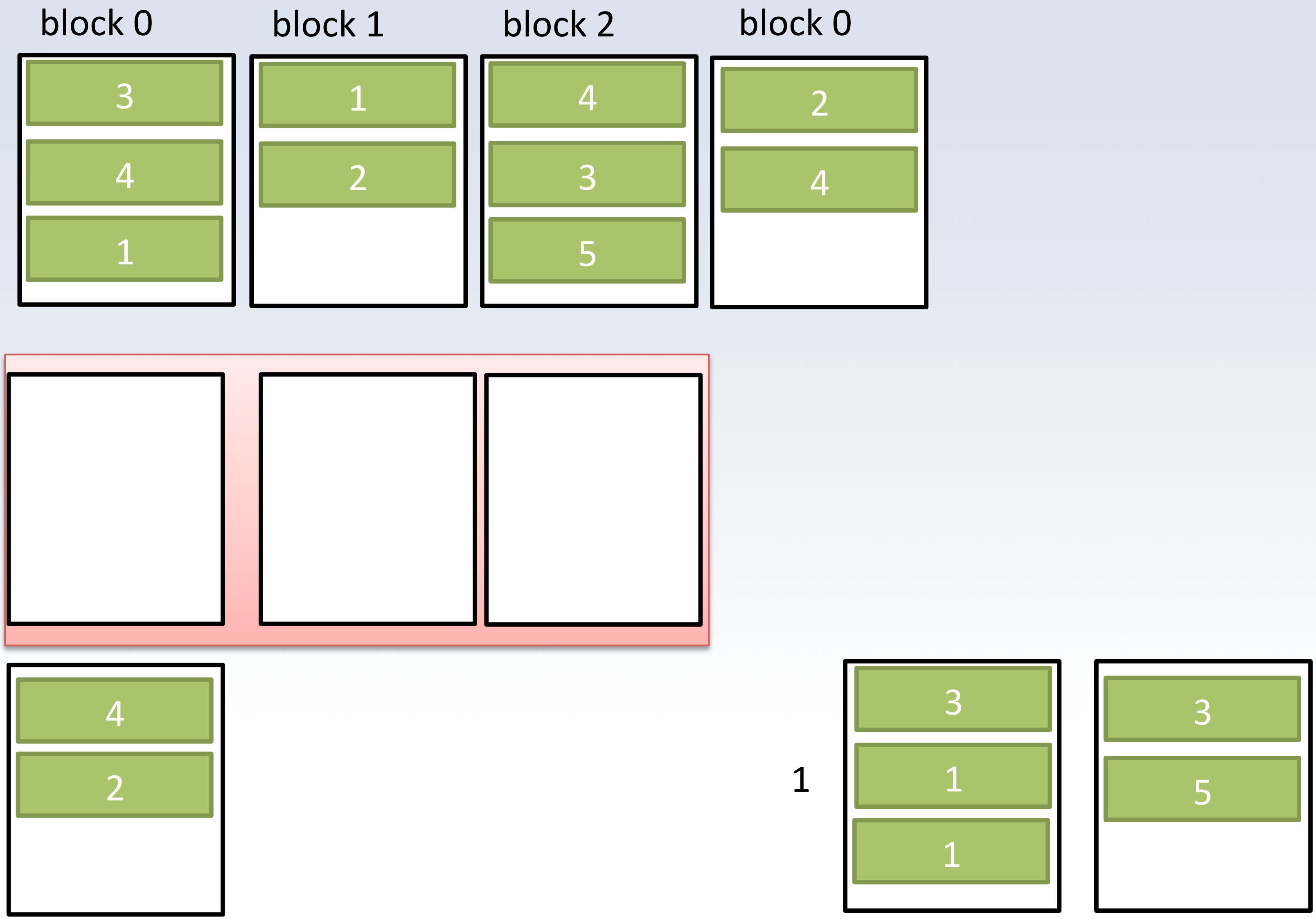


Two Pass Hashing for δ

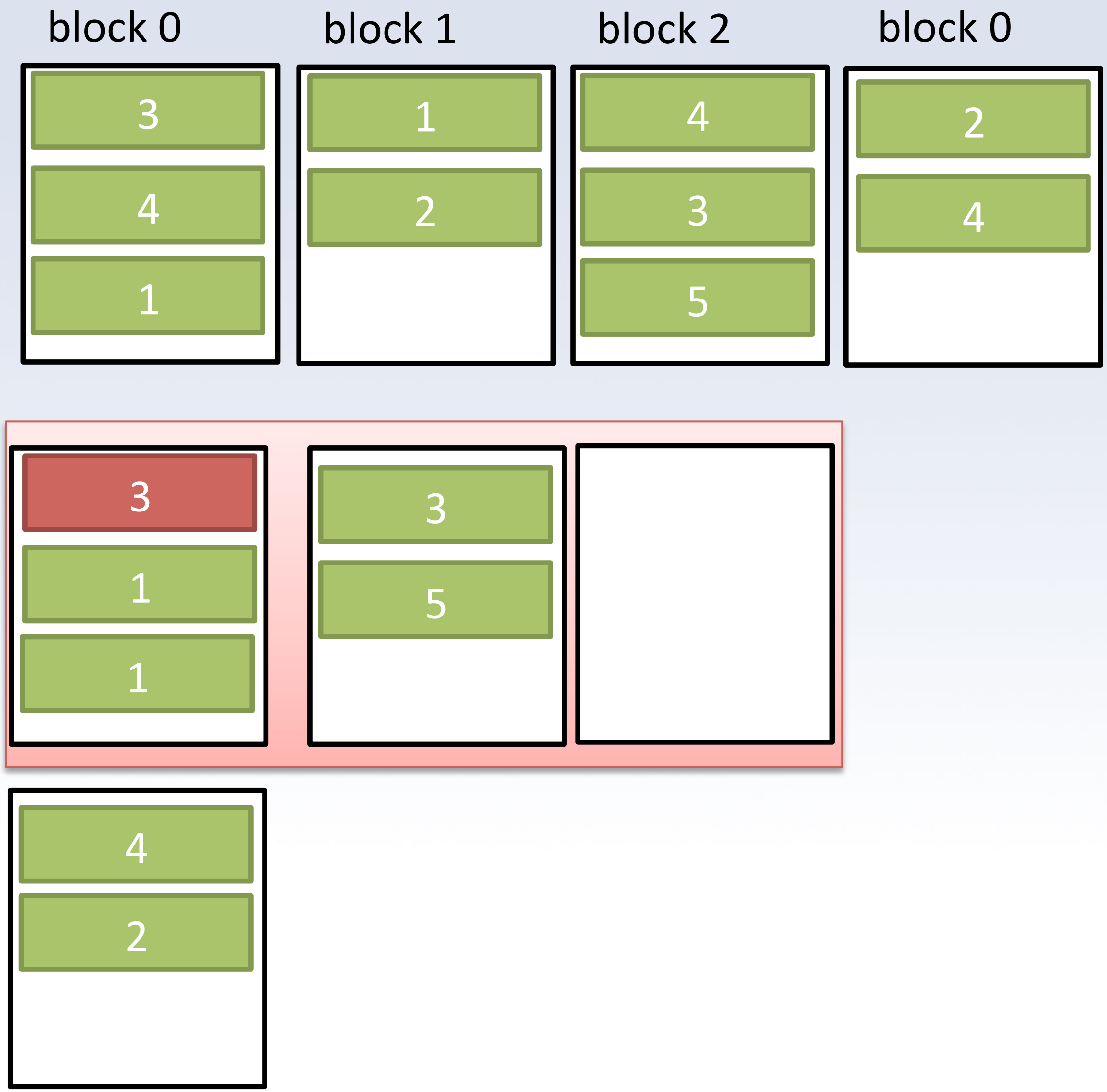


illinois.edu

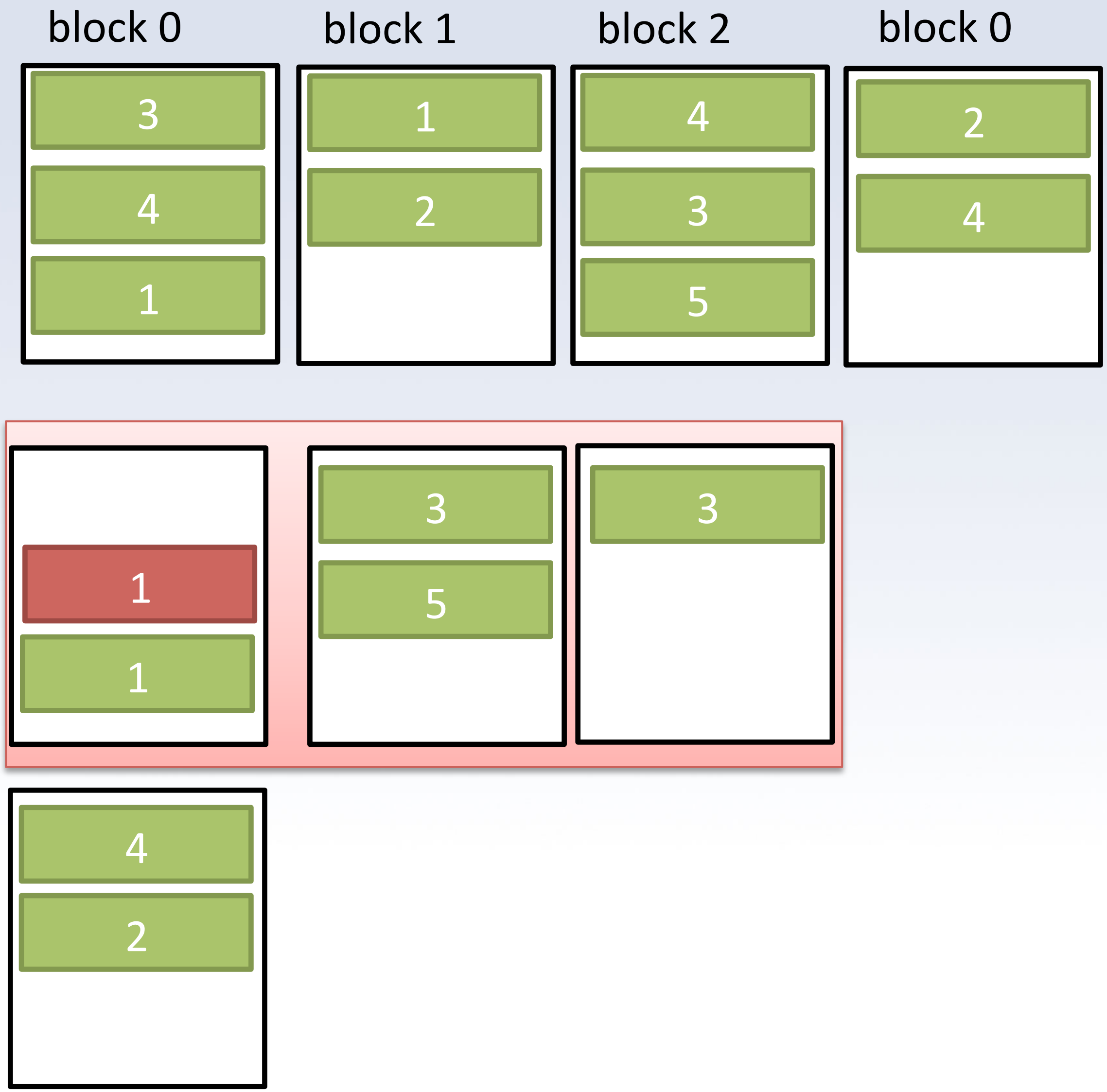
Two Pass Hashing for δ



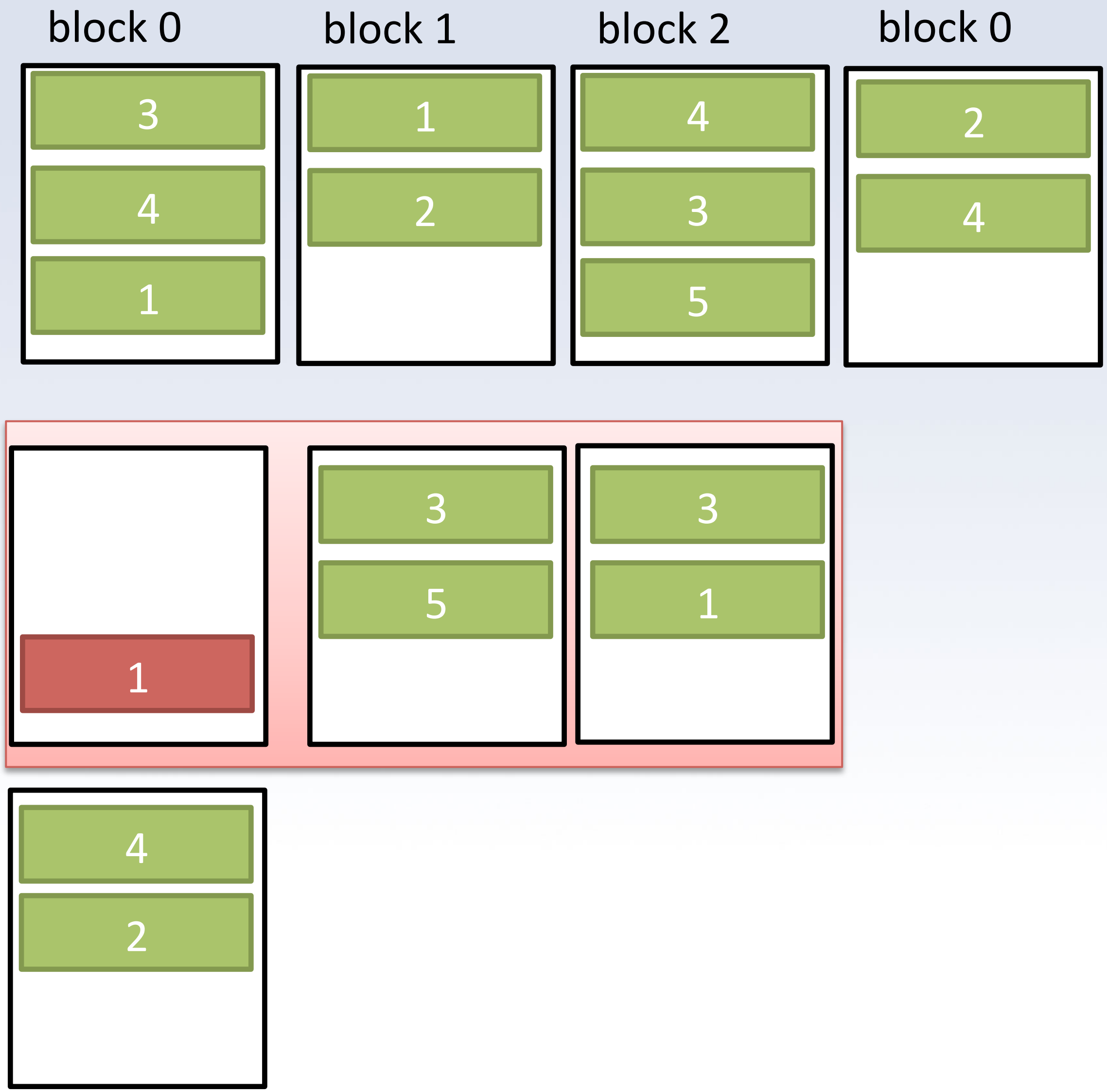
Two Pass Hashing for δ



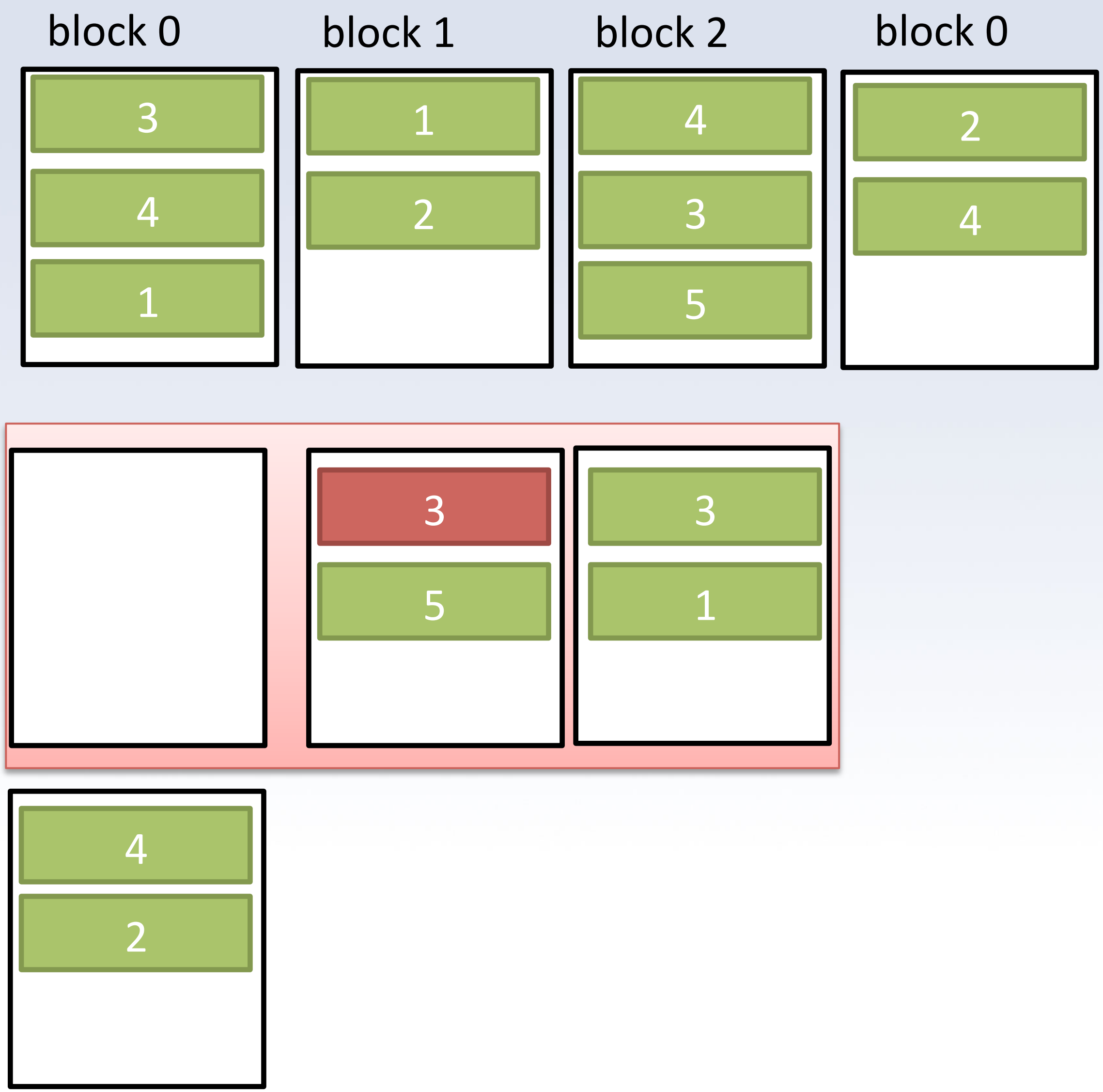
Two Pass Hashing for δ



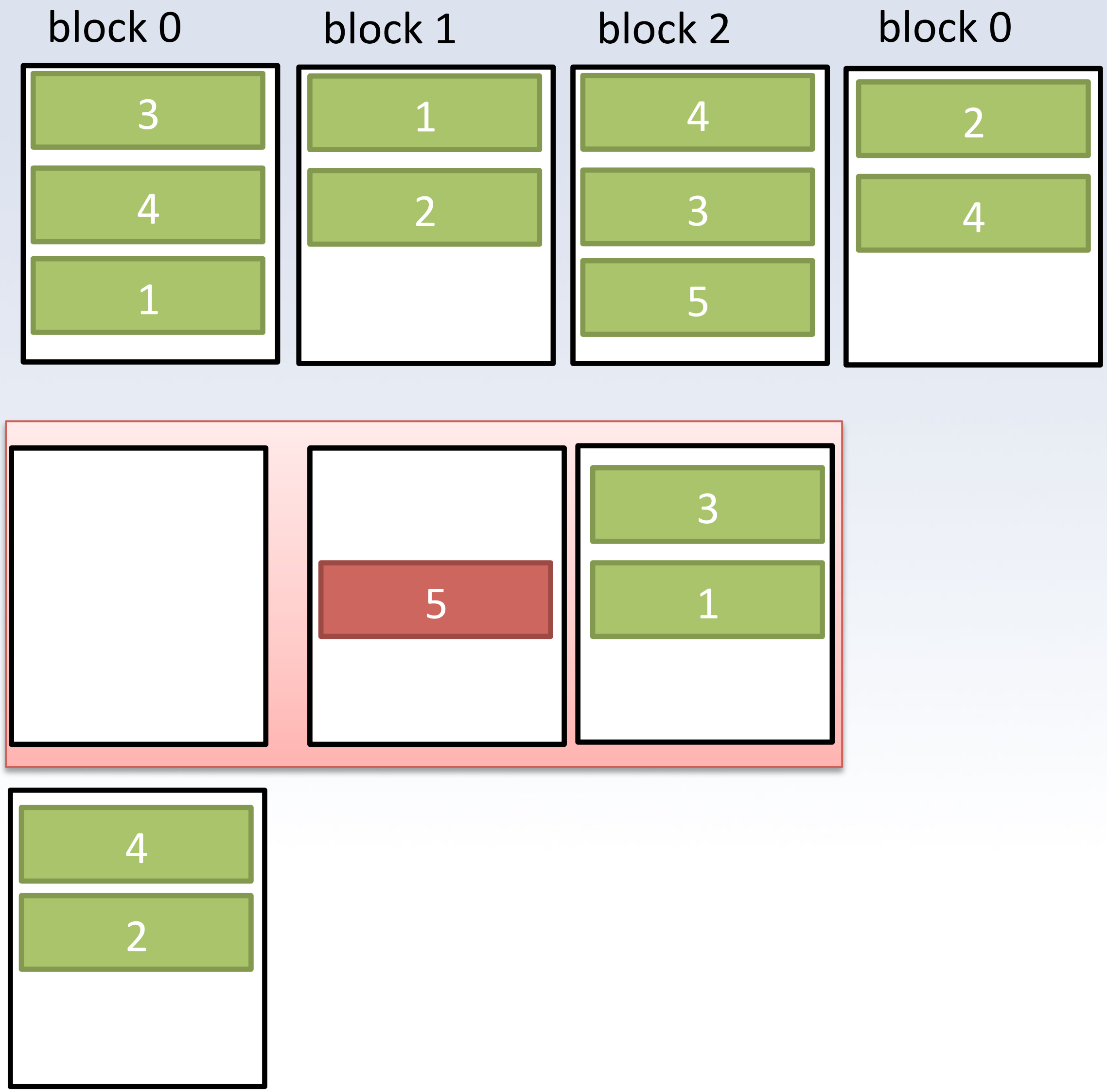
Two Pass Hashing for δ



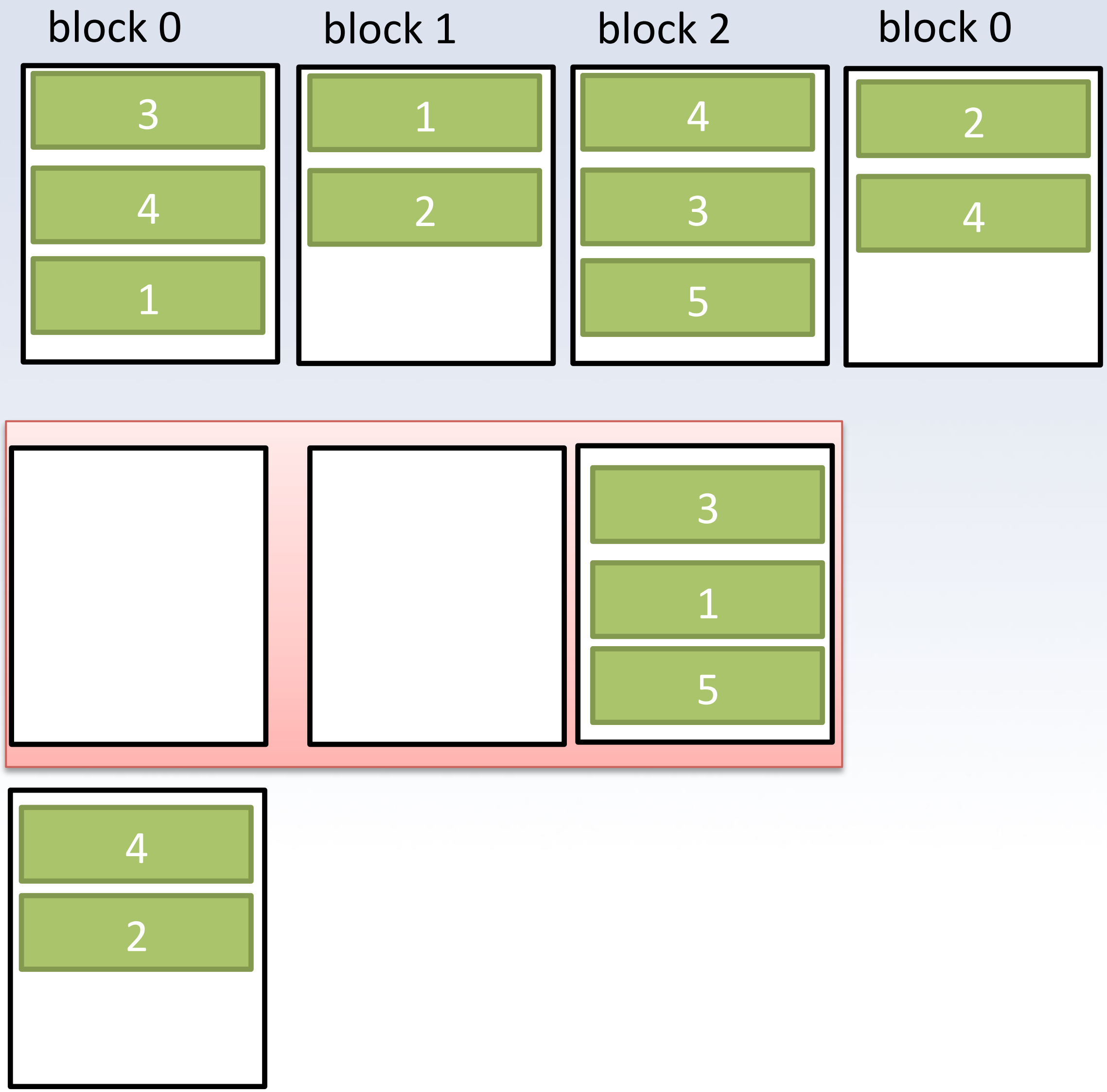
Two Pass Hashing for δ



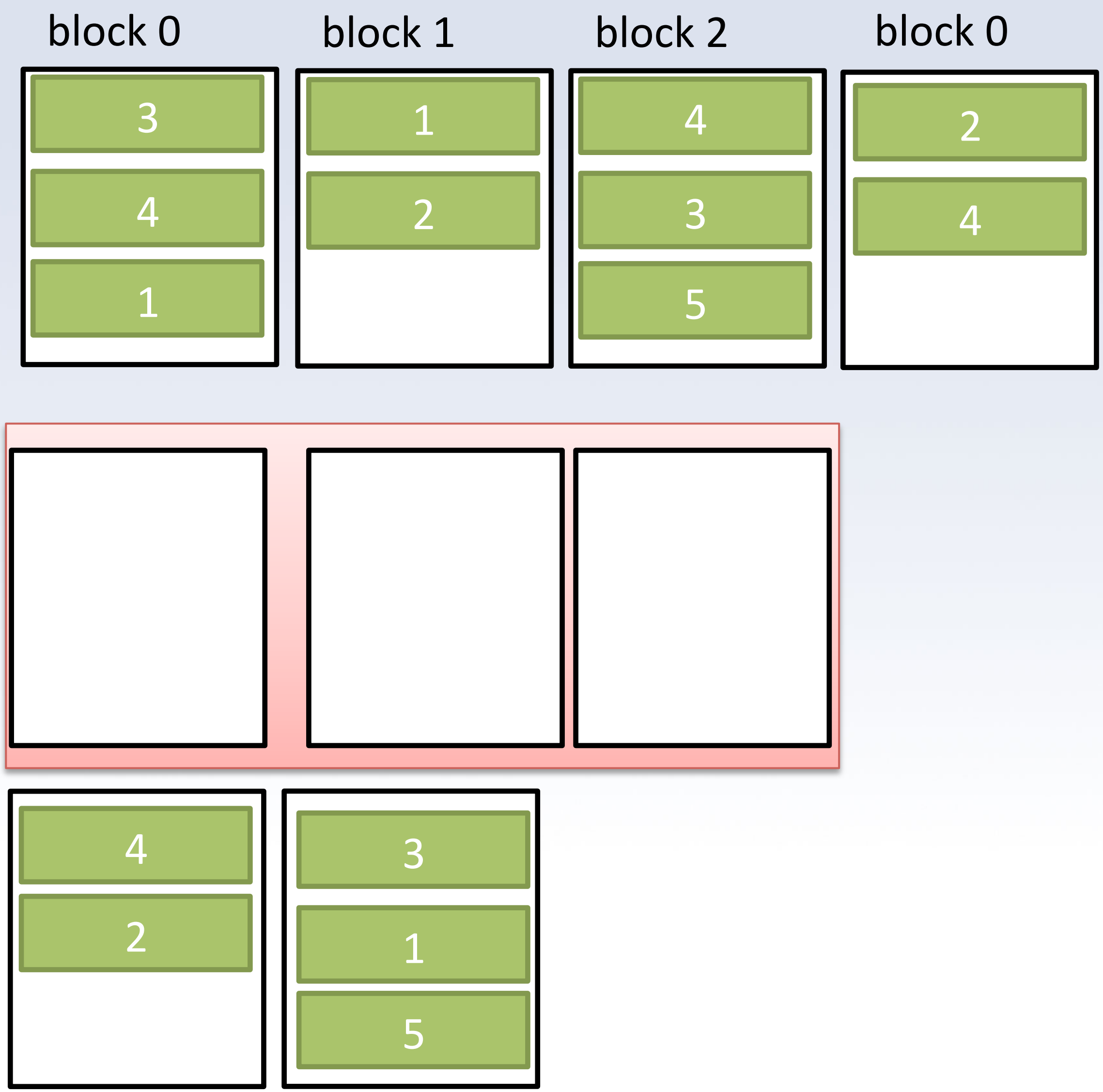
Two Pass Hashing for δ



Two Pass Hashing for δ



Two Pass Hashing for δ



Analysis

- Requires that each bucket fits in memory
  - $B(R)/(M-1) \leq M$
  - Approximately $B(R) \leq M^2$
- Hashing all the data takes B(R) reads and B(R) writes
- Reading in the buckets takes B(R) reads
- Total cost: 3B(R)

Two Pass Hashing for γ

- We can do γ in a similar way
  - Hash each record into buckets using grouping attributes
  - Bring bucket into memory
  - Compute aggregation functions on each bucket

Two Pass Hashing for $\cup$, $\cap$, -

- Hash both relations using same hash function
- Perform one pass algorithms on each bucket
- Requires 3(B(R)+B(S)) I/O operations
- Requires bucket of smaller table to fit in memory: $\min(B(R),B(S)) \leq M^2$

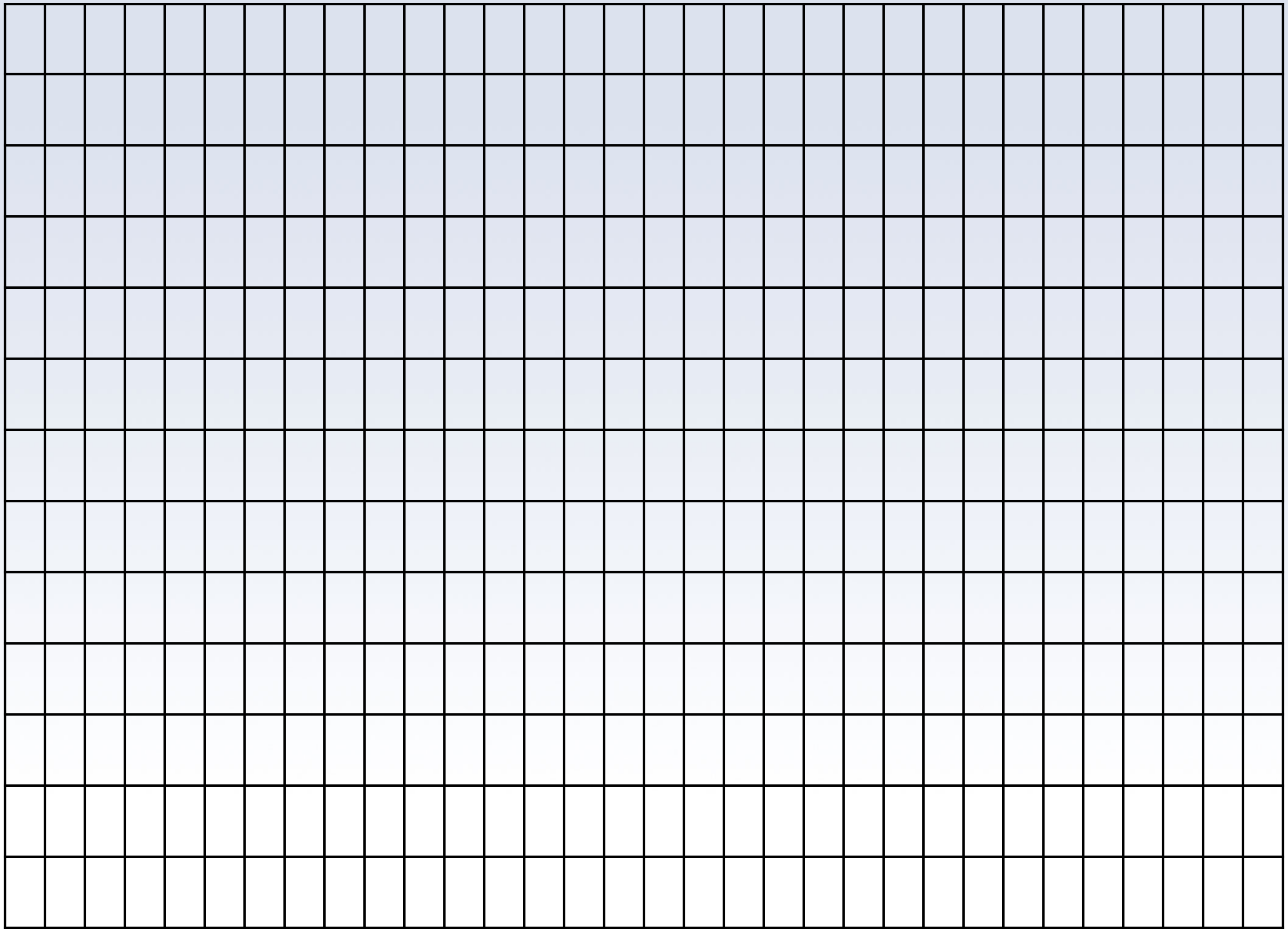
Two Pass Hashing for ⋈

- Hash both relations *on join attributes* using same hash function
- Perform one pass join algorithm on each bucket
- Requires 3(B(R)+B(S)) I/O operations
- Requires bucket of smaller table to fit in memory: $\min(B(R),B(S)) \leq M^2$

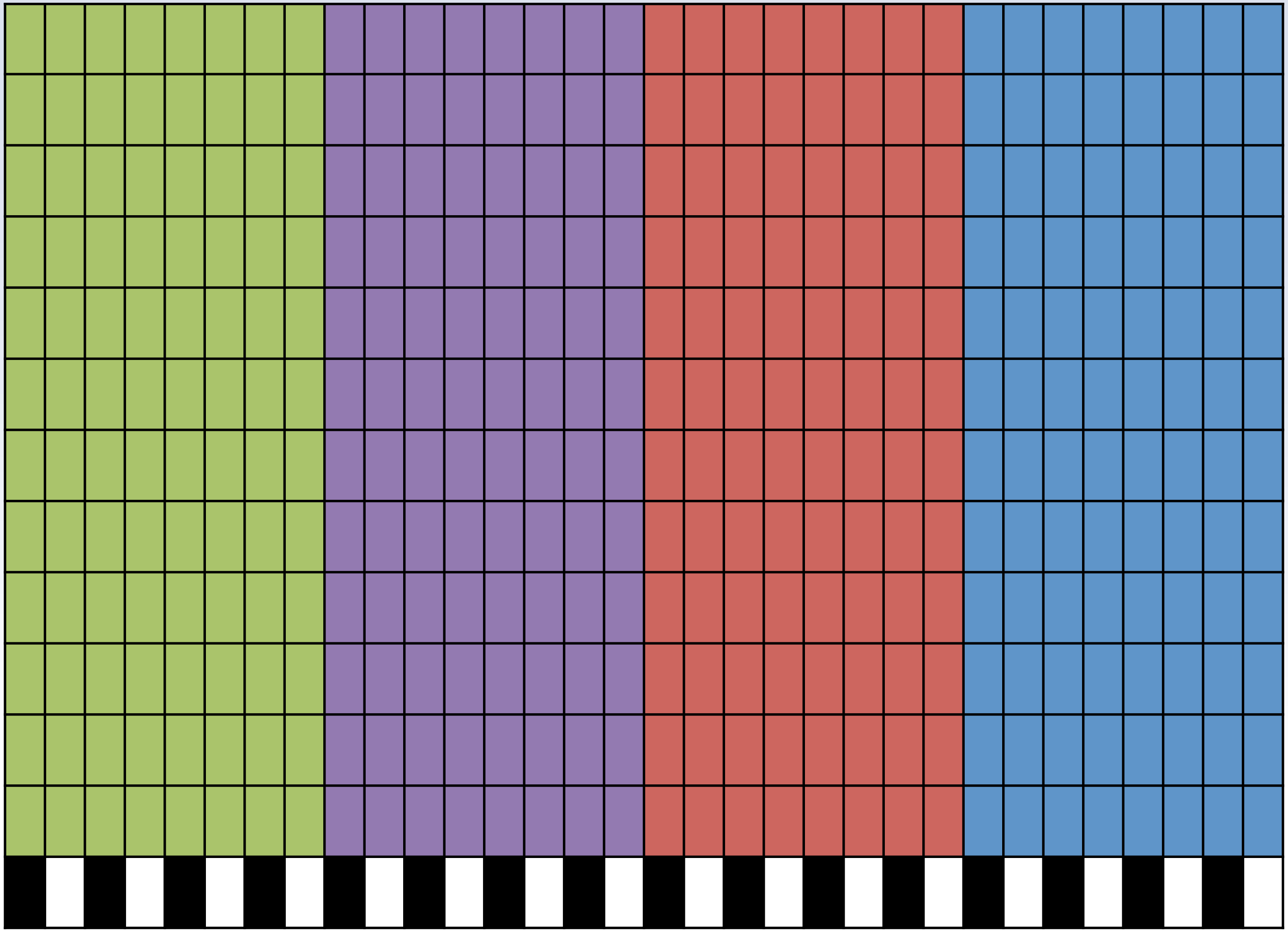
Hybrid Hash Join

- We can do better:
  - keep m of the k buckets in memory
  - never write these buckets to the disk
  - make these buckets larger than one buffer
  - perform joins immediately
  - keep one buffer for each of the remaining k-m buckets, use normal hash join for them

Example



Example



Hybrid Hash Join

- Requires that $mB(S)/k+k-m \leq M$
  - $B(S)/k$ expected size of "big" buckets
  - there are m "big" buckets in memory
  - k-m remaining buckets we write to disk
- Making the "big" buffers as big as possible, $k \approx B(S)/M$
  - Requires $B(S) \leq M$

Hybrid Hash Join

- Compared to hash join, saves 2 I/O operations per block of S in memory
- m/k buckets remain in memory
- Saves 2(m/k) reads and writes
- If k≈B(S)/M, we save 2M/B(S) I/Os
- Cost: (3-(2M/B(S)))(B(R)+B(S))

2 Pass Hash-Based Summary

Operator	M required	I/O Cost
δ, γ	$\sqrt{B}$	3B
$\cup, \cap, -$	$\sqrt{(B(S))}$	3(B(R)+B(S))
$\bowtie$	$\sqrt{(B(S))}$	3(B(R)+B(S))
$\bowtie$	$\sqrt{(B(S))}$	(3-2M/B(S))x (B(R)+B(S))

Practical Example

- On a system with 8GB=2^{33} of RAM
  - we can sort/join/deduplicate 2^{66}
  - Bigger than an exabyte of data!
- On a system with 256GB=2^{38}
  - we can sort/join/deduplicate 2^{76} bytes of data
  - almost a yottabyte!

Multipass Algorithms

- Two pass algorithms can be extended into multipass algorithms through recursion
- Can be used to sort *extremely* large datasets
- In reality, we'd need distributed systems to even *store* this much data!
- Map Reduce framework/Hadoop

Index-Based Algorithms

- With an index, we can perform some operations faster
- Need to talk about two different kinds of indexes
  - *clustering index* - search keys all appear together in blocks ***in the index file***
  - unclustered index - search keys are scattered through the index file

Clustered Index

block x	block x+1	block x+2
kkkkkkkkk	kkkkkkkkkkkkkkkkkkkkkkkkkkkk	kkkkkkkk

Index Based $\sigma_{a=v}(R)$

- Read the index to obtain pointers to records satisfying a=v
- If the index is clustered, only need to read B(R)/V(R,a)
  - proportion of blocks in R with a a=v
- If index is unclustered, read T(R)/V(R,a)
  - values are scattered through index
  - might have to read a block for every tuple!

Index Based R⋈S

- Assume S is indexed on join attribute
- Table scan R and use index to look up corresponding join attributes in S
- Cost:
  - B(R) blocks for table scan
  - B(S)/V(S,Y) for each lookup (clustered)
  - T(S)/V(S,Y) for each lookup (unclustered)

Index Based R⋈S

- Assume S is indexed on join attribute
- Table scan R and use index to look up corresponding join attributes in S
- Cost:
  – B(R)B(S)/V(S,Y) (clustered)
  – B(R)T(S)/V(S,Y) (unclustered)

Zig-zag Join

- Requires a sorted index (e.g. B-tree)
- Start at smallest key in each sorted index
- If keys are equal, start joining
- Otherwise, skip in index with smaller key
- Cost: B(R)+B(S)

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

illinois.edu

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

START JOINING!

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

START JOINING!

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

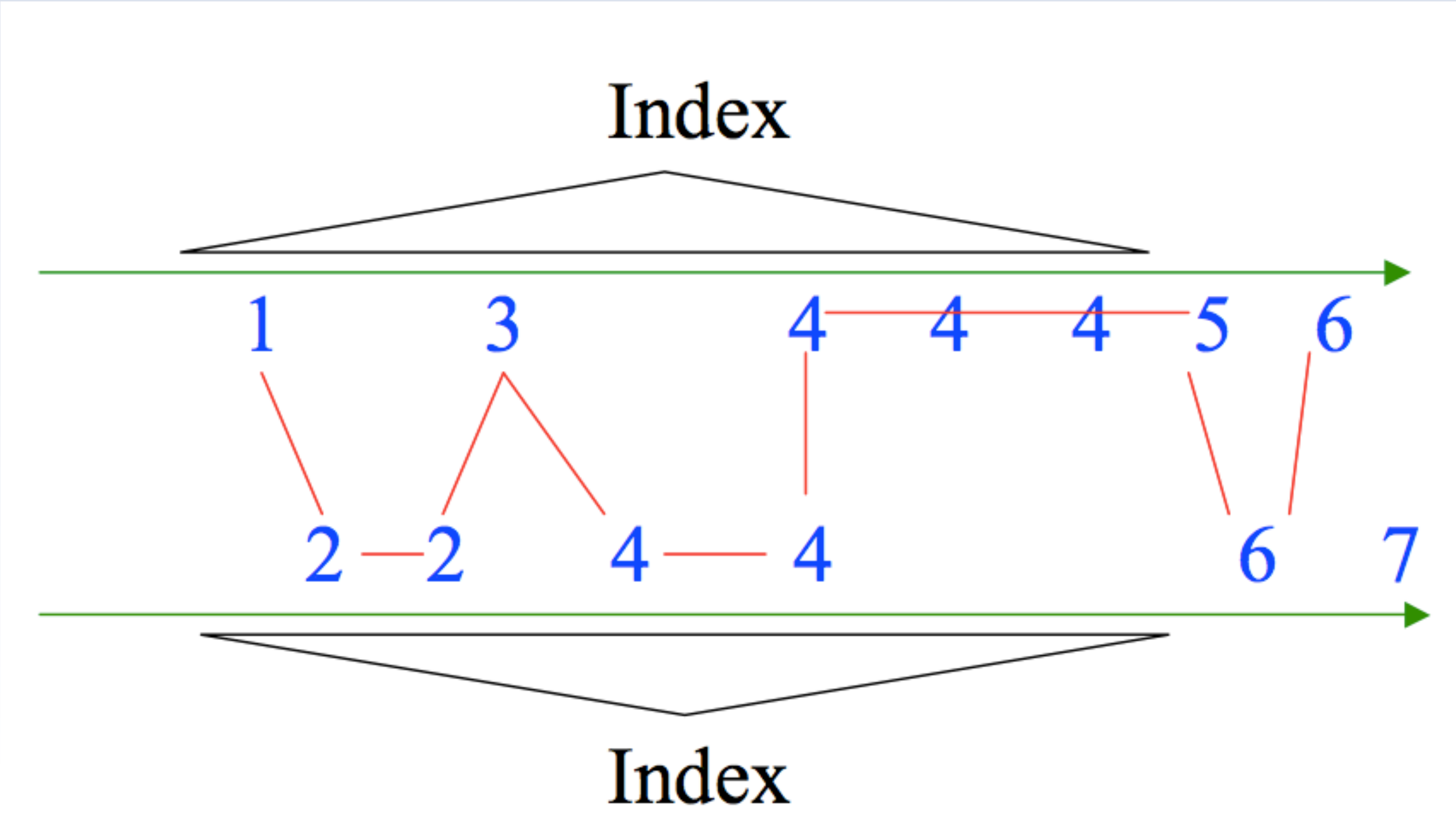
START JOINING!

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Why "zig-zag"?



Zig-zag Join

- With B-tree, might speed this up
  - If joinable entries are sparse, don't scan through both sorted tables
  - Instead, look up bigger value in the other B-tree
  - Skips reading most values that don't join!

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

Example

```
R: 1 3 4 4 4 5 6

S: 2 2 4 4 6 7
```

illinois.edu

Next week

- We'll start putting all this together:
  – Compile SQL queries into operators
  – Estimate costs of query plans
  – Figure out an optimal query plan