Query Evaluation

R&G Chapter 12,14

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

Query Plans

- A tree of relational algebra operators
  - Multiple algorithms for some operators
  - Some algorithms only for special cases.
- Each operator "pulls" tuples from operators below it in the query plan when needed.
  - Contrast with operators "pushing" tuples as they become available (used in Streaming)

General Strategies

- **Indexing**: Build a datastructure to organize your data, then access the data.
- **Iteration**: Look at each data value individually.
- **Partitioning**: If your data is too big, break it up into smaller chunks and process each chunk individually.

Query Optimization

- Search through “equivalent” query plans.
- Two main issues:
  - Should we consider all equivalent plans?
  - How do we compute the cost of a plan?
- **Ideally**: Find the best plan!
- **Practically**: Avoid the worst plans!

Query Optimization

- Search through "equivalent" query plans.
- Two main issues:
  - ~~Should~~ Can we consider all equivalent plans?
  - How do we compute the cost of a plan?
- **Ideally**: Find the best plan!
- **Practically**: Avoid the worst plans!

Query Optimization

- Search through "equivalent" query plans.
- Two main issues:
  - ~~Should~~ Can we consider all equivalent plans? (no)
  - How do we compute the cost of a plan?
- **Ideally**: Find the best plan!
- **Practically**: Avoid the worst plans!

Query Optimization

- Search through "equivalent" query plans.
- Two main issues:
  - ~~Should~~ Can we consider all equivalent plans? (no)
  - How do we compute the cost of a plan?
- **Ideally**: Find the best plan!
- **Practically**: Avoid the worst plans!

(We'll soon cover the approach taken by System R)

Query Plans

```
SELECT O.FirstName
FROM Officers O, Ships S
WHERE O.Ship = S.ID
  AND S.Name = ‘Enterprise’
```

Query Plans

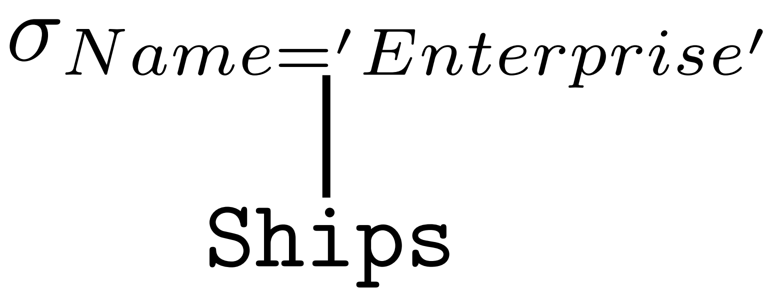
```
SELECT O.FirstName
FROM Officers O, Ships S
WHERE O.Ship = S.ID
  AND S.Name = ‘Enterprise’
```

$$\pi_{FirstName}(\texttt{Officers} \bowtie_{Ship=ID} (\sigma_{Name='Enterprise'}\texttt{Ships}))$$

Query Plans

$$\pi_{FirstName}(\texttt{Officers} \bowtie_{Ship=ID} (\sigma_{Name='Enterprise'}\texttt{Ships}))$$

Query Plans

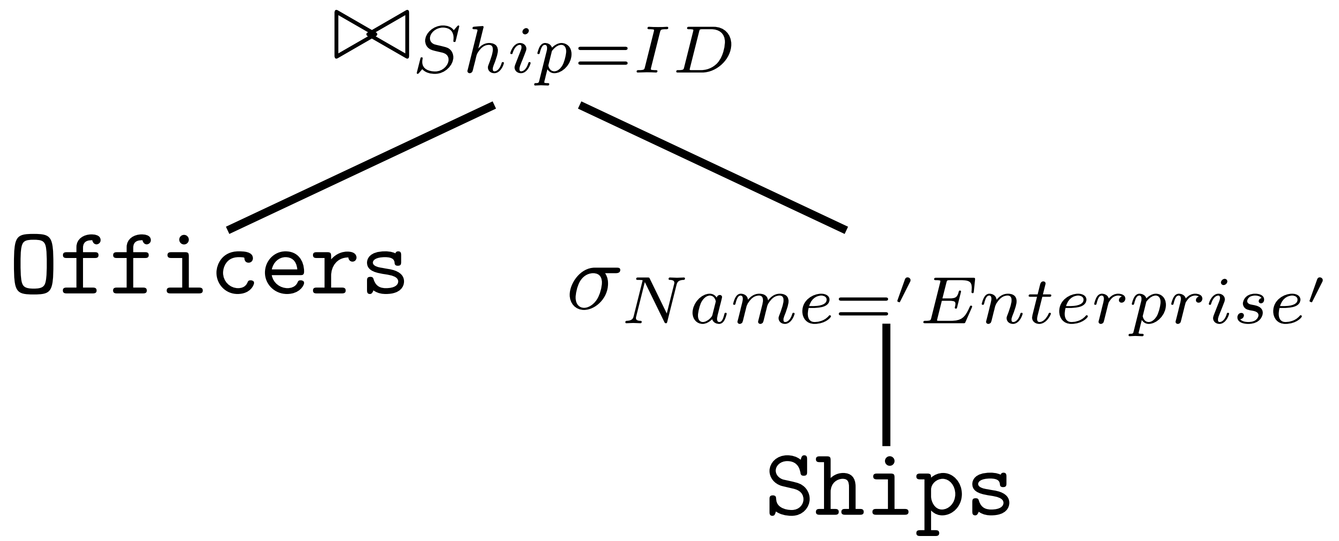


$\sigma_{Name='Enterprise'}$

$\texttt{Ships}$

$$\pi_{FirstName}(\texttt{Officers} \bowtie_{Ship=ID} (\sigma_{Name='Enterprise'}\texttt{Ships}))$$

Query Plans



$\bowtie_{Ship=ID}$

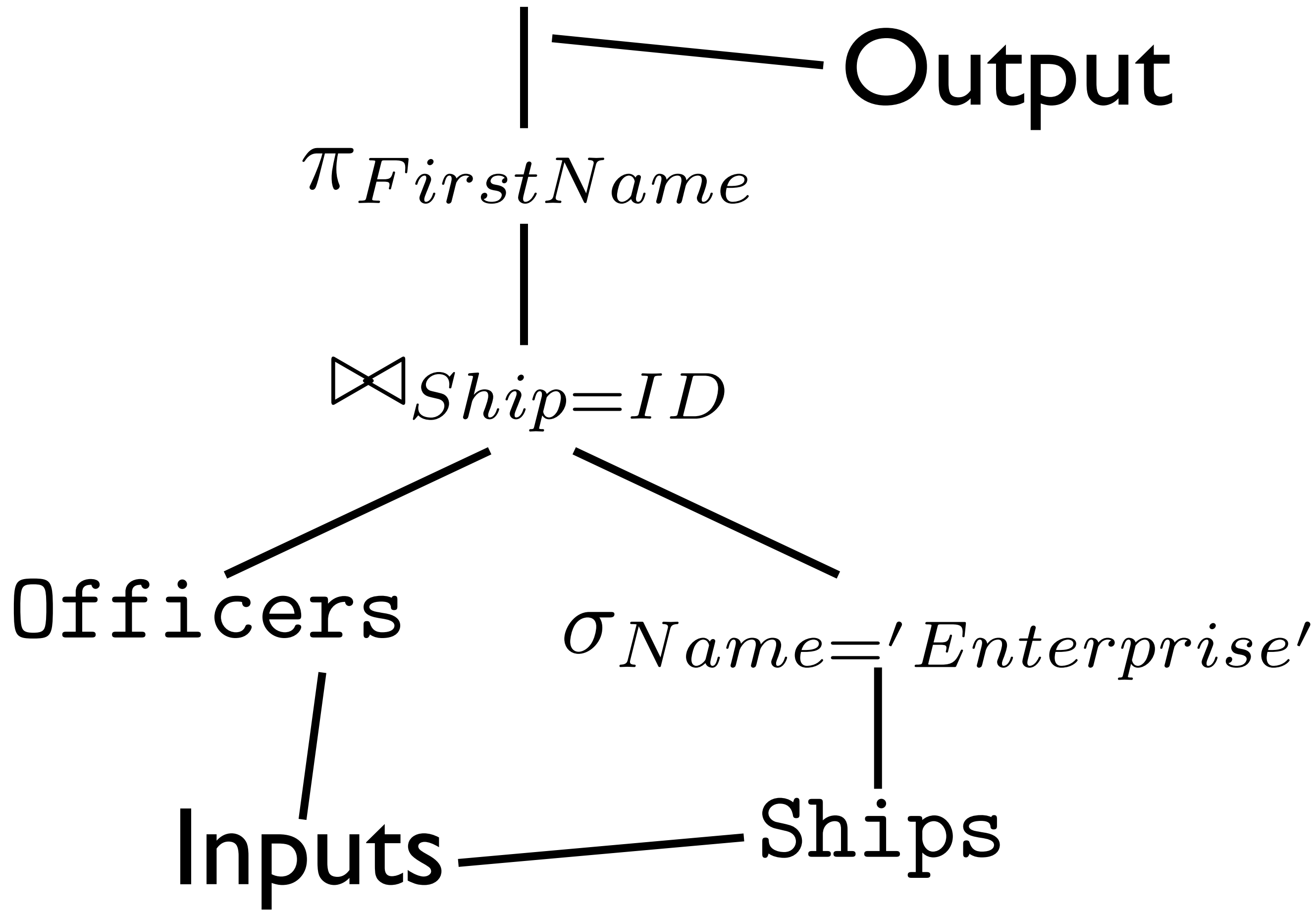
$\texttt{Officers}$

$\sigma_{Name='Enterprise'}$

$\texttt{Ships}$

$$\pi_{FirstName}(\texttt{Officers} \bowtie_{Ship=ID} (\sigma_{Name='Enterprise'}\texttt{Ships}))$$

Query Plans



Output

$\pi_{FirstName}$

$\bowtie_{Ship=ID}$

Officers

$\sigma_{Name='Enterprise'}$

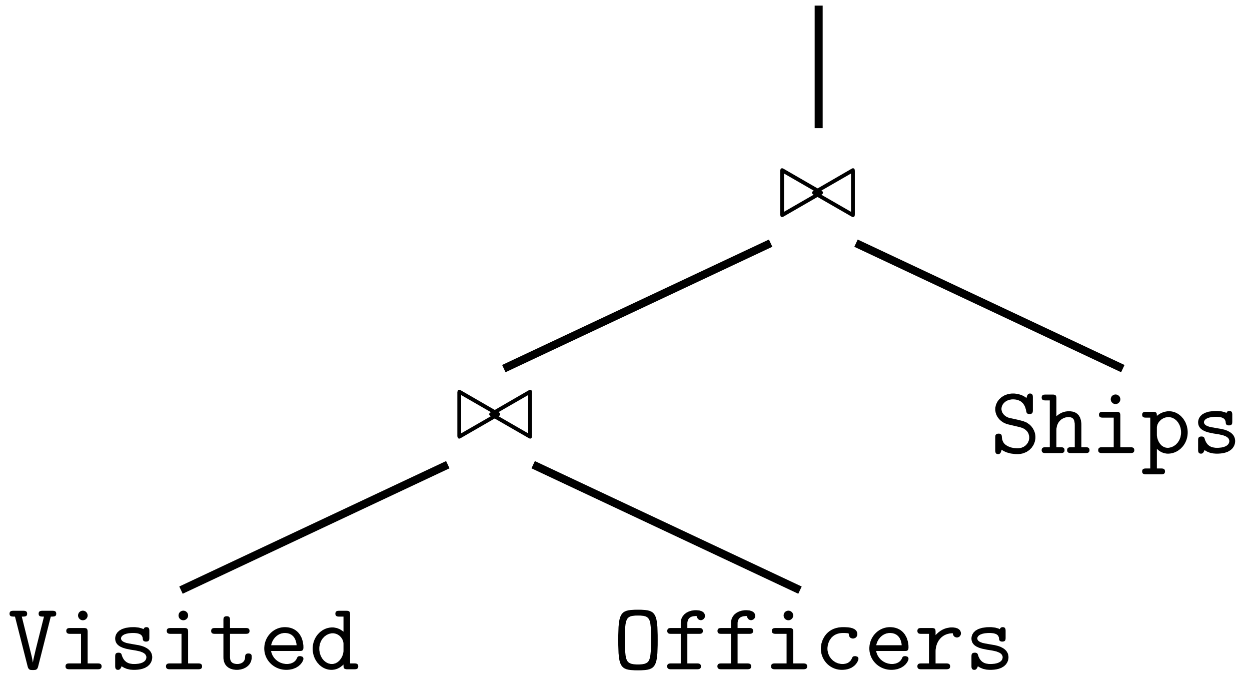
Inputs

Ships

$$\pi_{FirstName}(\texttt{Officers} \bowtie_{Ship=ID} (\sigma_{Name='Enterprise'}\texttt{Ships}))$$

Left-Deep Query Plans

$\texttt{Visited} \bowtie \texttt{Officers} \bowtie \texttt{Ships}$



Make the join-tree as deep as possible (to the left)

Left-Deep Query Plans

- Easy to construct/optimize
  - Small(er) search space of plans
- Easy to pipeline the output of one join directly into the next.
  - Joins are half-blocking.
- The only type of plan used in System R
  - Works well for < 10 joins (Widely used)

The Iterator Interface

- void open()
- Tuple get_next()
- void close()
- [optional] void reset()

Implementing: Selection

Solution 1 (Naive/On-the-fly)

Remove tuples from the stream as they arrive

$$\boxed{\sigma}$$

Implementing: Selection

Solution 1 (Naive/On-the-fly)

Remove tuples from the stream as they arrive

$$\boxed{\sigma}$$

Implementing: Selection

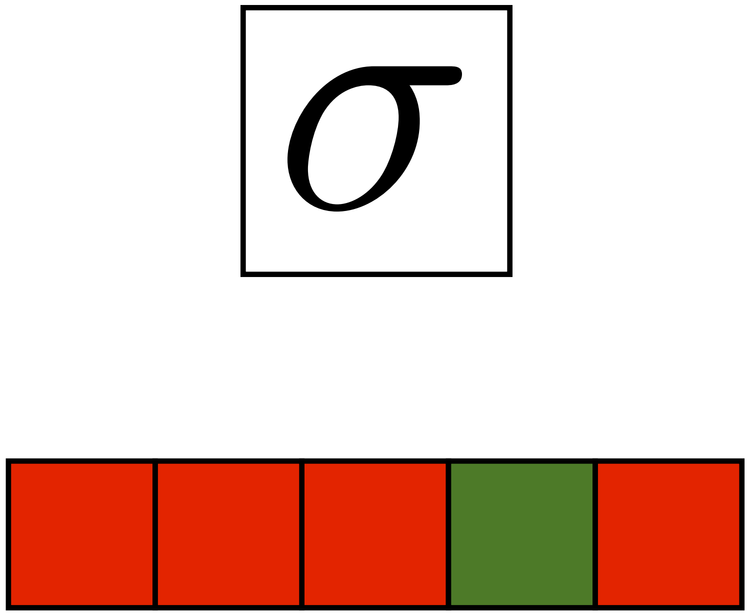
Solution 2 (Point/Range Lookup)

$$\boxed{\sigma}$$

Sort if necessary

Implementing: Selection

Solution 2 (Point/Range Lookup)



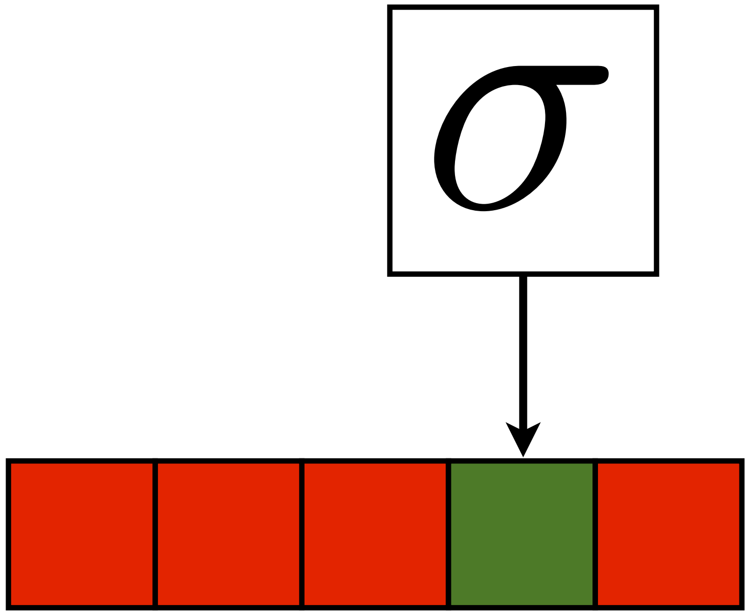
Materialize
the inputs
(if necessary)

Sort if
necessary

Implementing: Selection

Solution 2 (Point/Range Lookup)

Binary search
to find the
desired value(s)

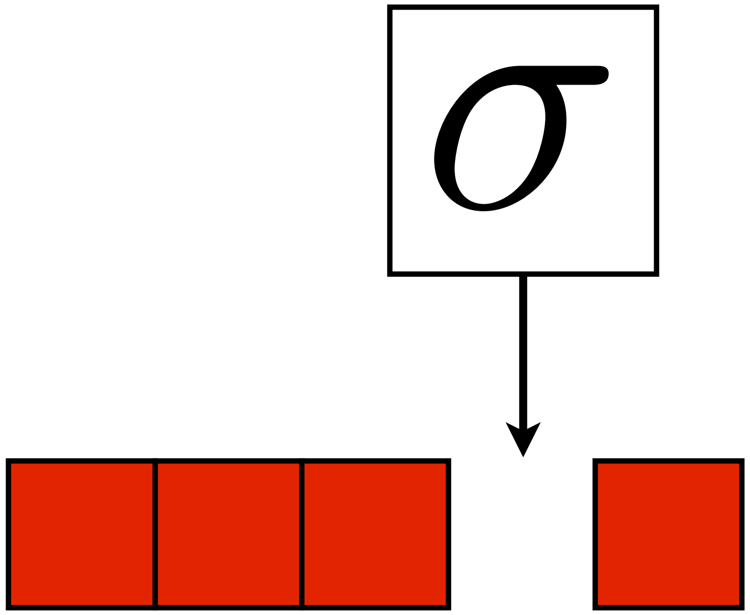


Materialize
the inputs
(if necessary)

Sort if
necessary

Implementing: Selection

Solution 2 (Point/Range Lookup)



Binary search to find the desired value(s)

Materialize the inputs (if necessary)

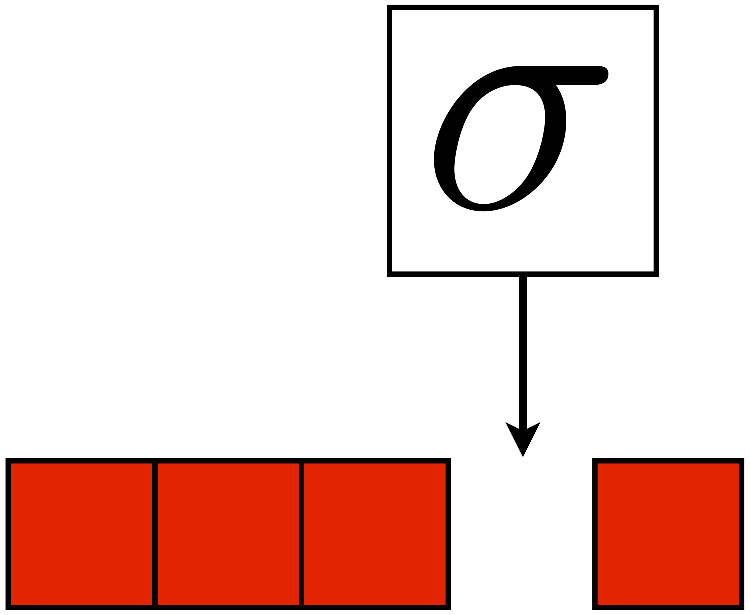
Sort if necessary

Implementing: Selection

Solution 2 (Point/Range Lookup)

When is it not necessary to materialize the inputs?

Binary search to find the desired value(s)



Materialize the inputs (if necessary)

Sort if necessary

Materialization

- **Materialization**: Fully computing an operator's output before proceeding.
- Files are already materialized.
- Some operators must materialize their output.
    - (Group-by) Aggregation
    - Sorting
    - Projection (but only set-projection)
    - Set Difference
- Some operators must materialize their input.
    - Joins, Cross Product

Implementing: Projection

Solution 1 (Naive/On-the-fly)

Remove fields from the stream as they arrive

$$\boxed{\pi}$$

Implementing: Projection

Solution 1 (Naive/On-the-fly)

Remove fields from the stream as they arrive

$$\boxed{\pi}$$

Implementing: Projection

Solution 1 (Naive/On-the-fly)

Remove fields from the stream as they arrive

$\boxed{\pi}$

Problem! This is Bag-projection

Implementing: Distinct

Solution 1 (Sort)

2	3	1	5	1	4	2	4

Implementing: Distinct

Solution 1 (Sort)

Sort Inputs

1	1	2	2	3	4	4	5

Implementing: Distinct

Solution 1 (Sort)

Sort
Inputs

Scan for (adjacent) duplicates while reading out values

Implementing: Distinct

Solution 1 (Sort)

Sort
Inputs

Scan for (adjacent) duplicates while reading out values

If the data is already sorted, can compute distinct pipelined

Hash Functions

- A hash function is a function that maps a large data value to a small fixed-size value
  - Typically is deterministic & pseudorandom
- Used in Checksums, Hash Tables, Partitioning, Bloom Filters, Caching, Cryptography, Password Storage, …
- Examples: MD5, SHA1, SHA2

Implementing: Distinct

Solution 2 (Hash Table)

1

3

2

5

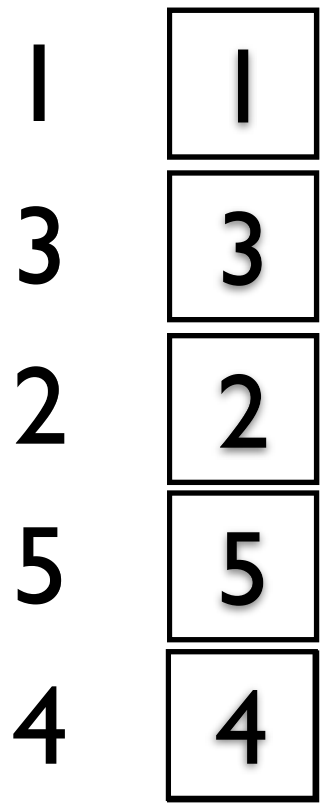
4

What happens if we run out of memory for the hash table?

- Only the current hash bucket needs to fit

Implementing: Distinct

Solution 2 (Hash Table)

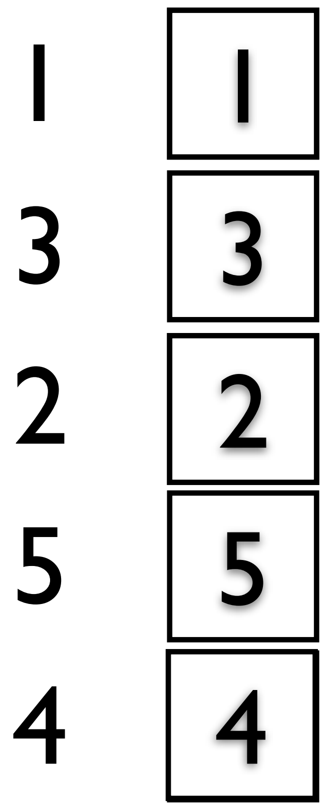


What happens if we run out of memory for the hash table?

- Only the current hash bucket needs to fit

Implementing: Distinct

Solution 2 (Hash Table)


1 1
3 3
2 2
5 5
4 4

(Only the current hash bucket needs to fit in memory)

What happens if we run out of memory for the hash table?

- Only the current hash bucket needs to fit

Distinct

Sort vs Hash

When should either be used?

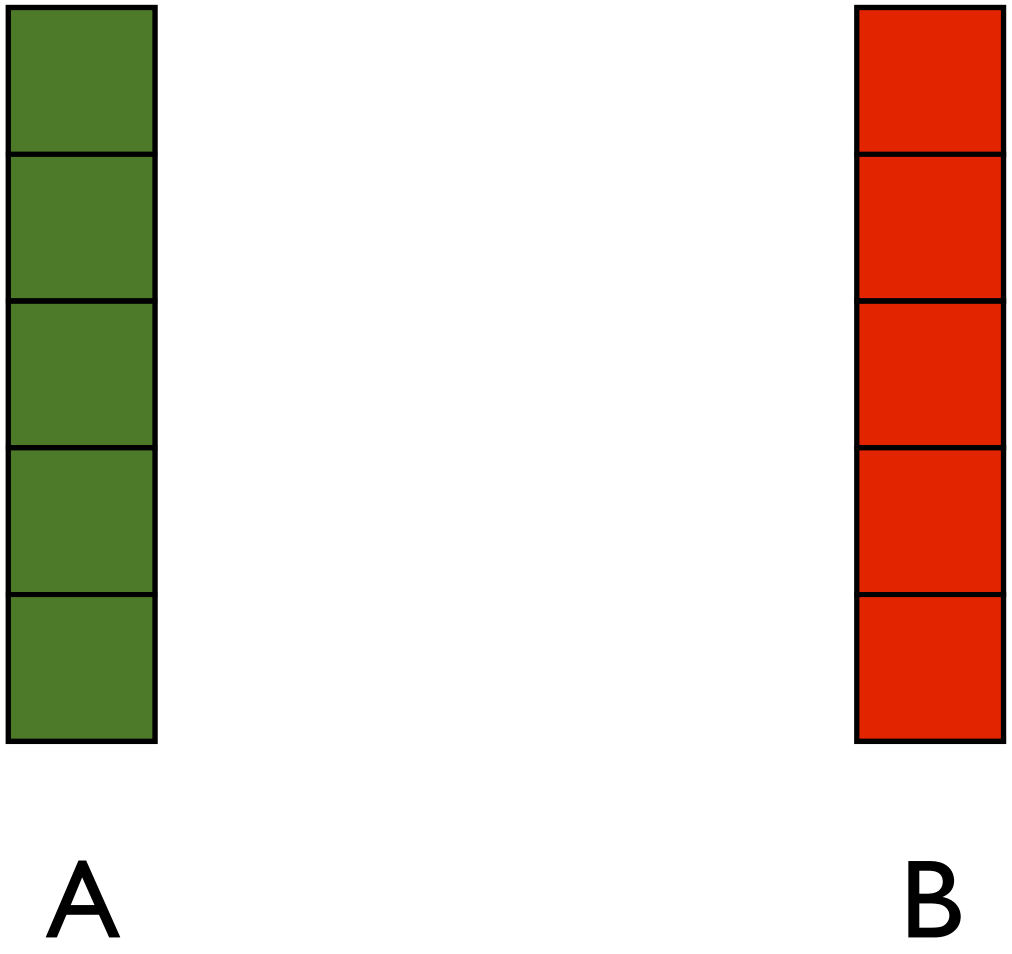
Sorting

- is O(n*log(n)) computations, and “requires” random access to the data

Implementing: Joins

Solution 1 (Nested-Loop)

For Each (a in A) { For Each (b in B) { emit (a, b); }}



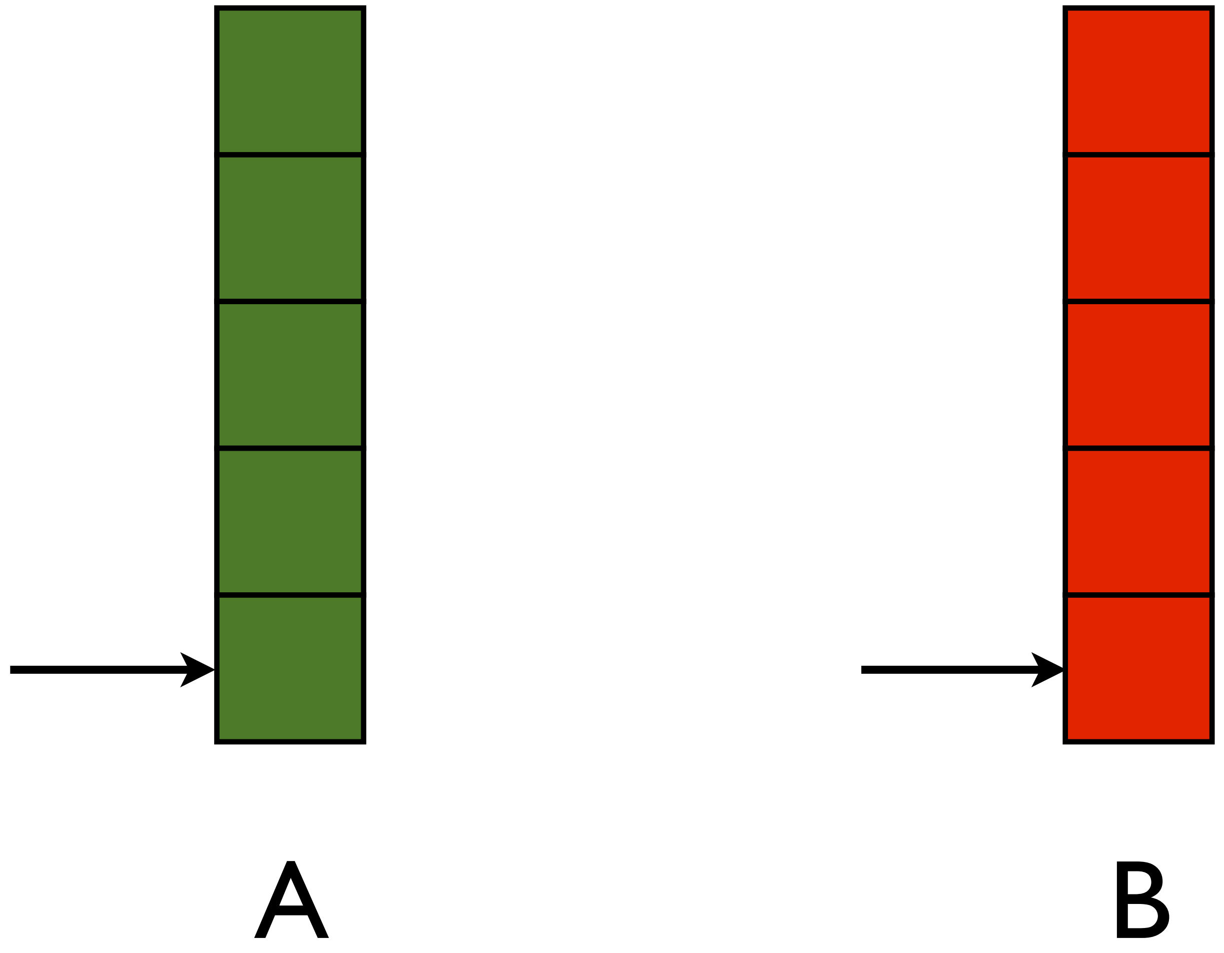
How many scans of each relation are required?

- One scan of A, |A| scans of B.

Implementing: Joins

Solution 1 (Nested-Loop)

For Each (a in A) { For Each (b in B) { emit (a, b); }}



How many scans of each relation are required?

- One scan of A, |A| scans of B.

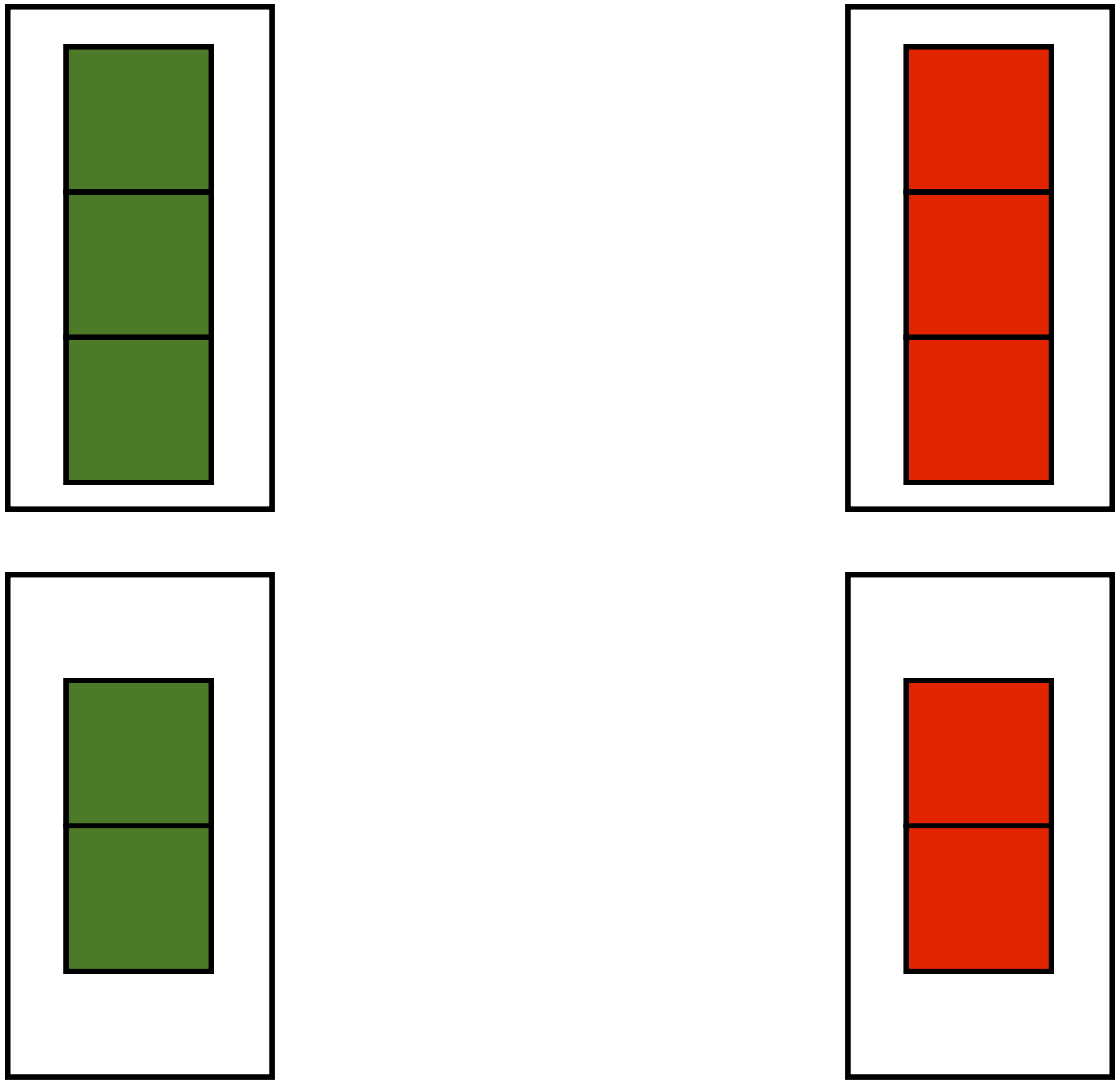
Implementing: Joins

Solution 2 (Block-Nested-Loop)

Implementing: Joins

Solution 2 (Block-Nested-Loop)

1) Partition into Blocks



Is this only beneficial if A or B don't fit in memory?

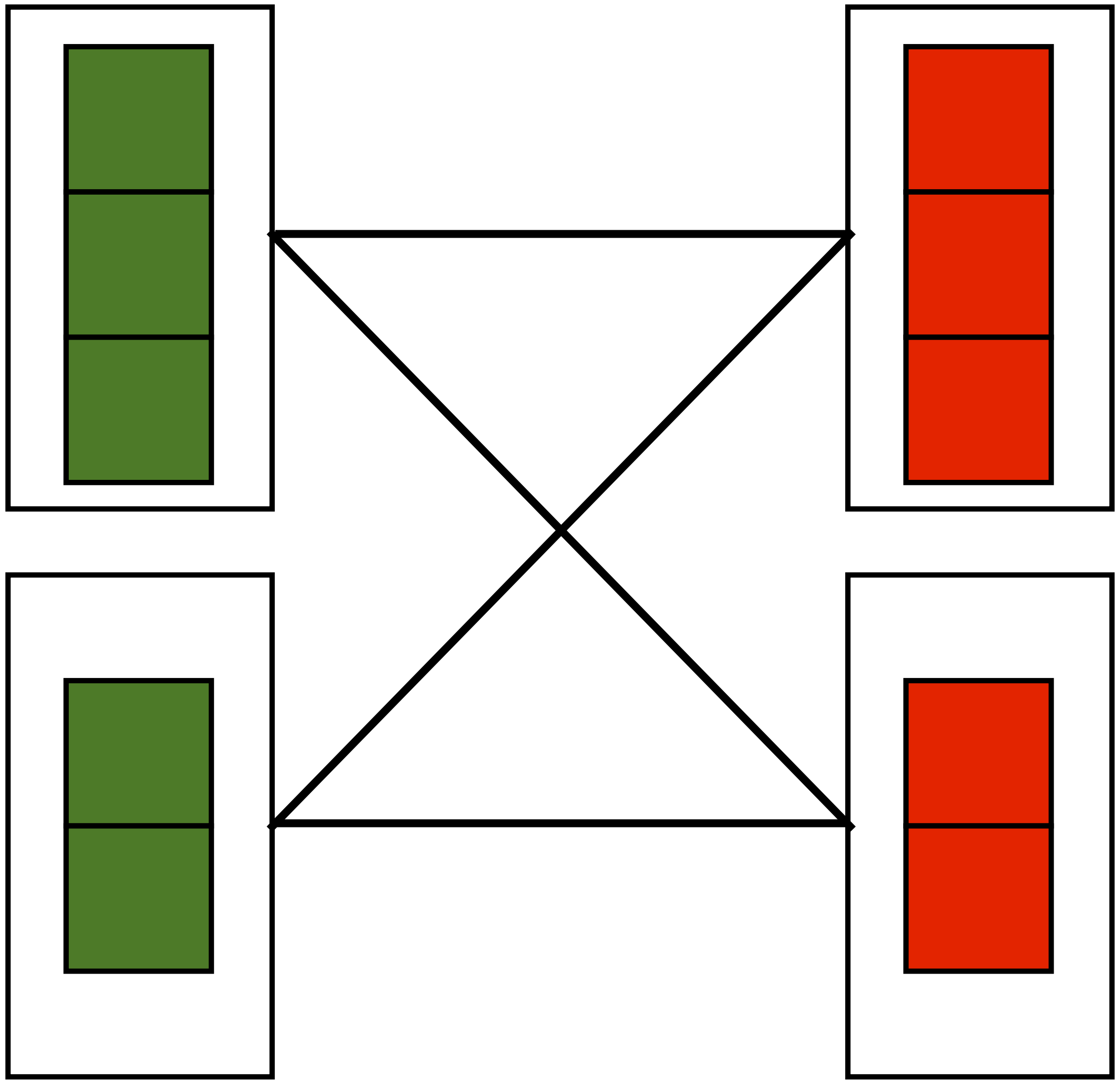
- No. BNLJ lets us exploit cache lines better.

Implementing: Joins

Solution 2 (Block-Nested-Loop)

1) Partition into Blocks

2) NLJ on each pair of blocks



Is this only beneficial if A or B don't fit in memory?

- No. BNLJ lets us exploit cache lines better.

Implementing: Joins

Solution 3 (Index-Nested-Loop)

Like nested-loop, but use an index to make the inner loop much faster!

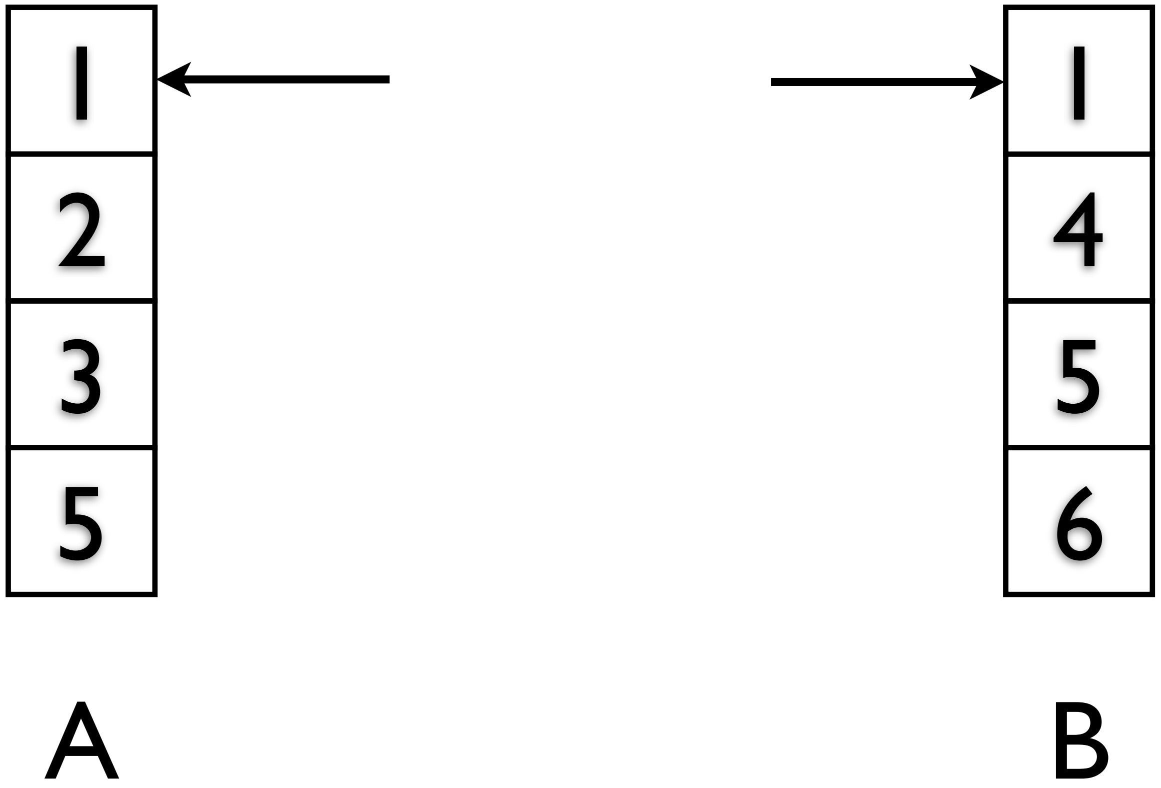
(We'll return to this soon)

Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both



A	B
1	1
2	4
3	5
5	6

When is sort-merge join a good idea?

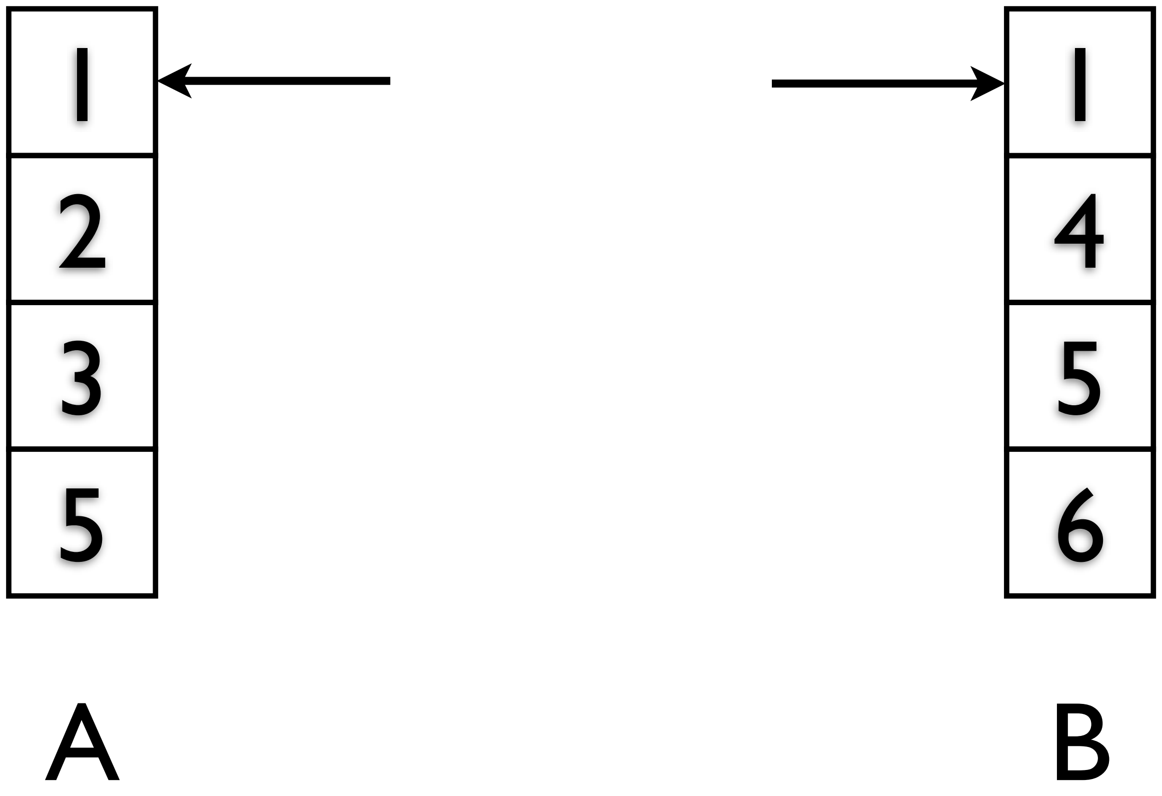
- When the cost of sorting both A and B is low

Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both



When is sort-merge join a good idea?

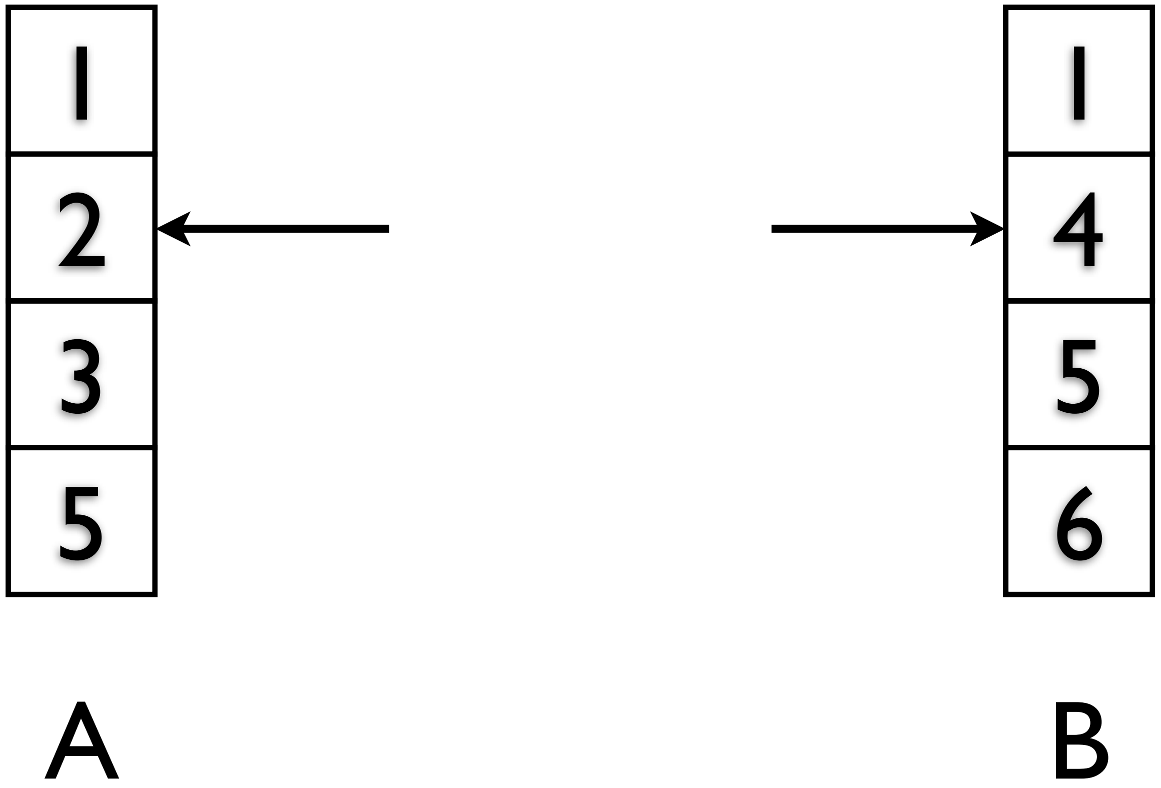
- When the cost of sorting both A and B is low

Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both



When is sort-merge join a good idea?

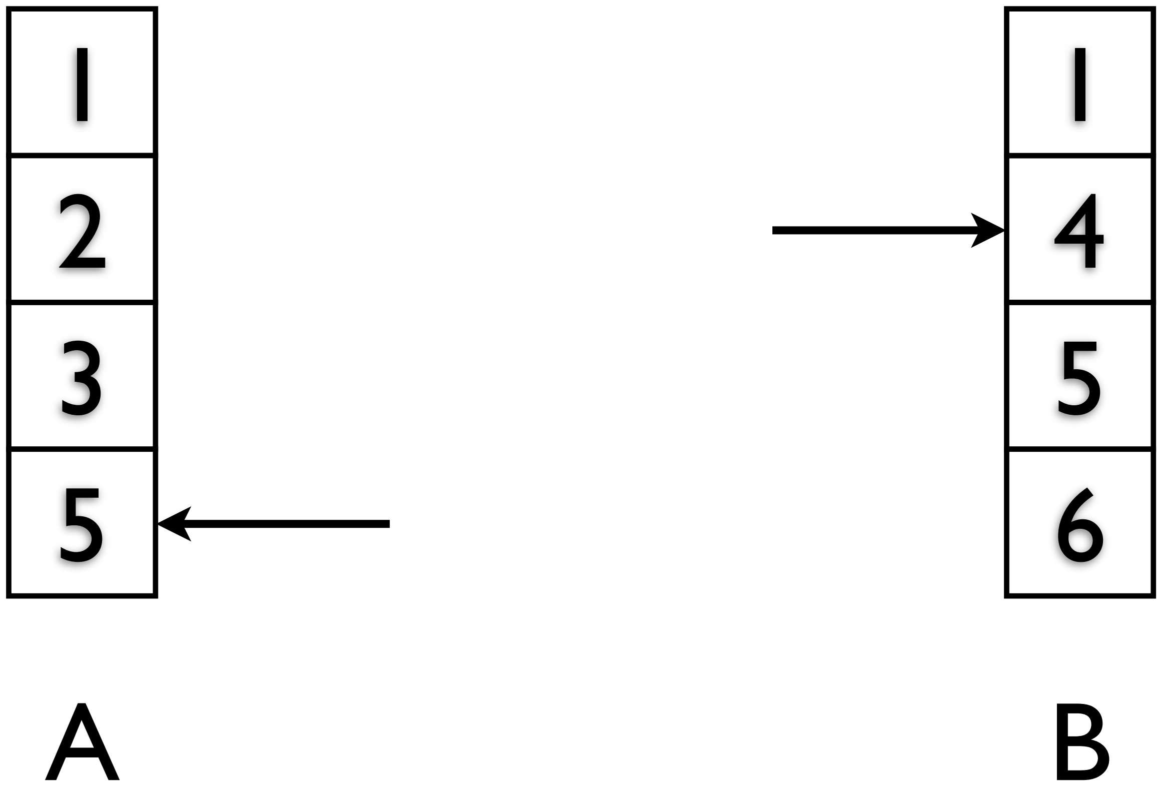
- When the cost of sorting both A and B is low

Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both



1
2
3
5

A

1
4
5
6

B

When is sort-merge join a good idea?

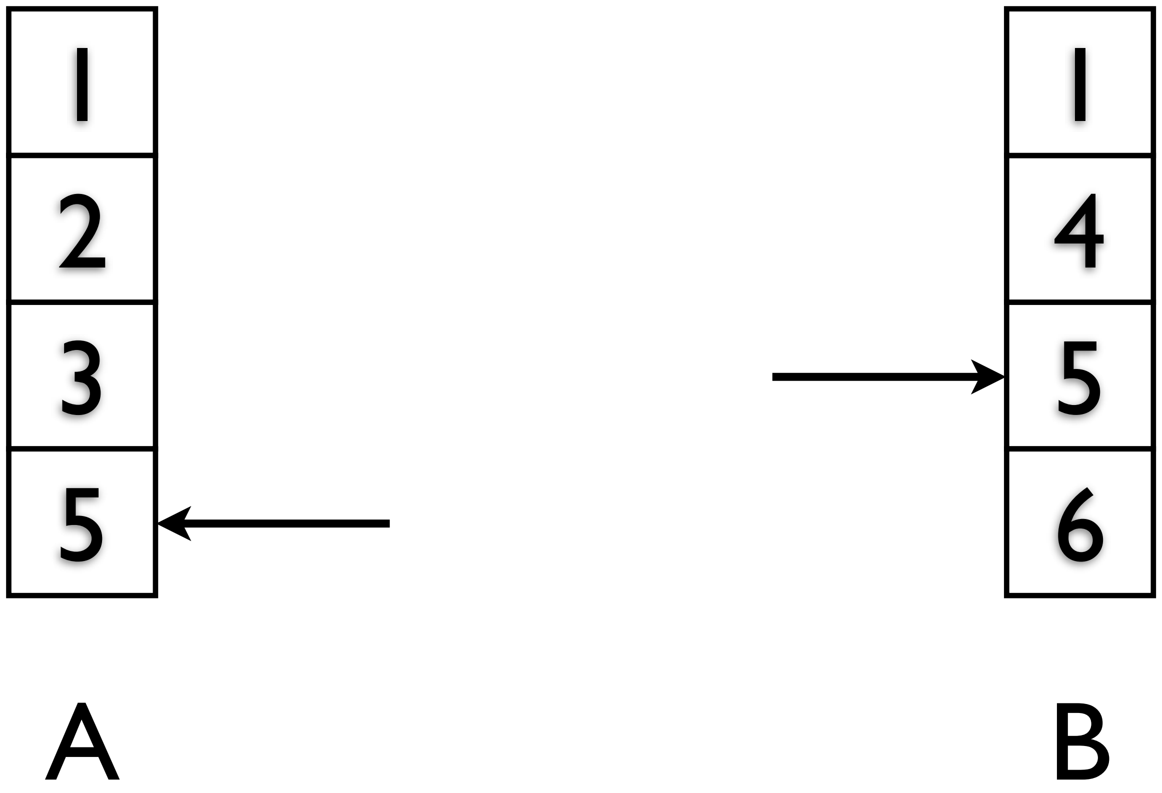
- When the cost of sorting both A and B is low

Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both


1
2
3
5
A
1
4
5
6
B

When is sort-merge join a good idea?

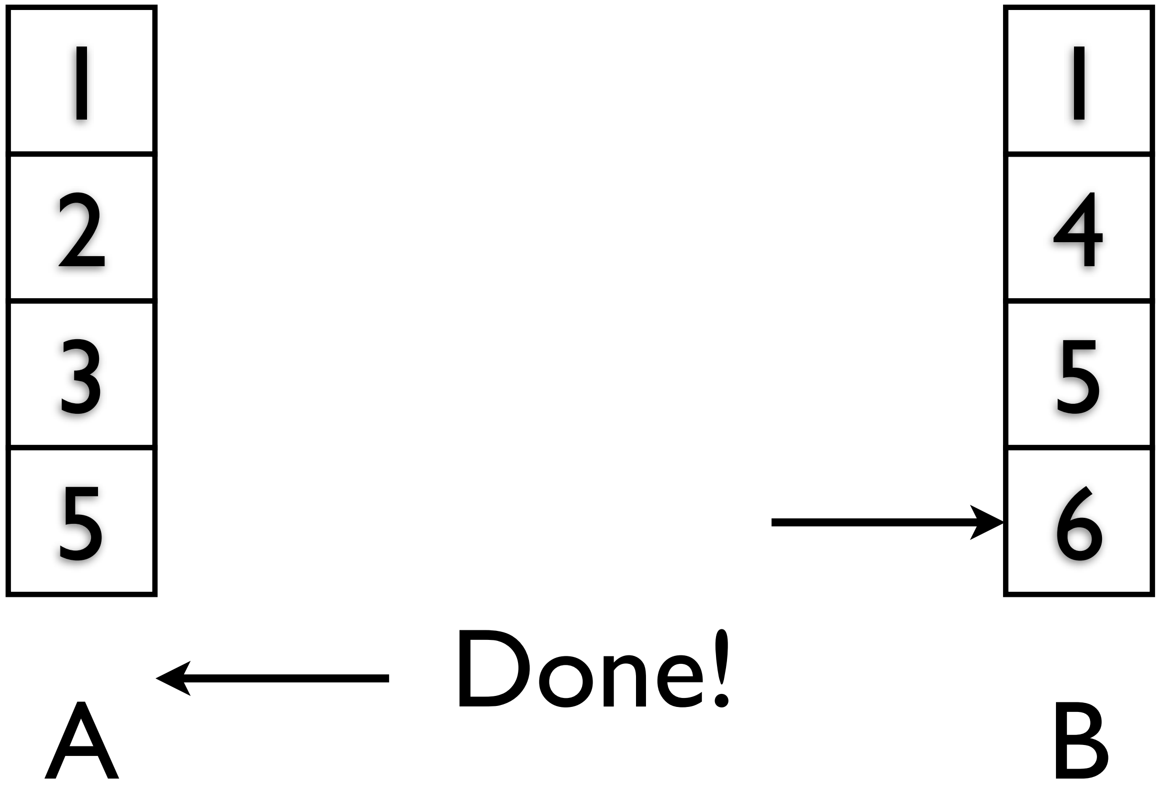
- When the cost of sorting both A and B is low

Implementing: Joins

Solution 4 (Sort-Merge Join)

Keep iterating on the set with the lowest value.

When you hit two that match, emit, then iterate both



A	B
1	1
2	4
3	5
5	6

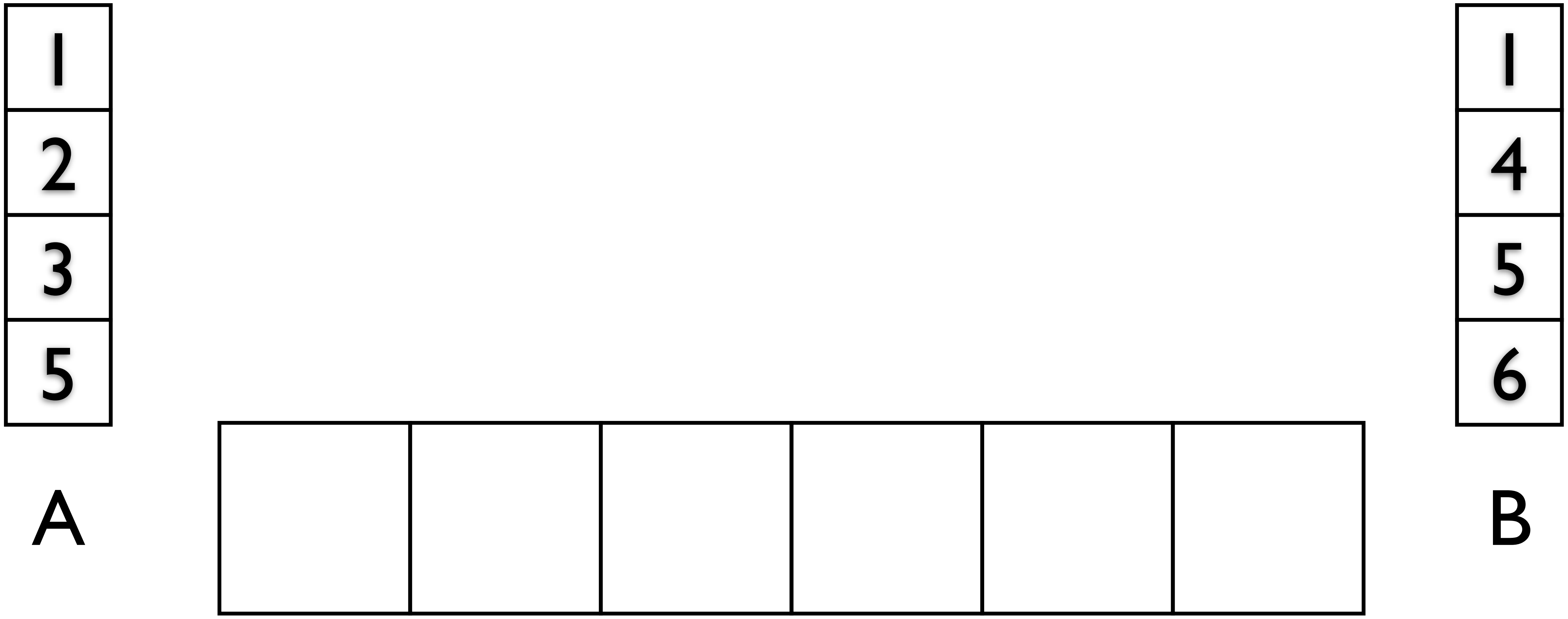
Done!

When is sort-merge join a good idea?

- When the cost of sorting both A and B is low

Implementing: Joins

Solution 5 (Hash)



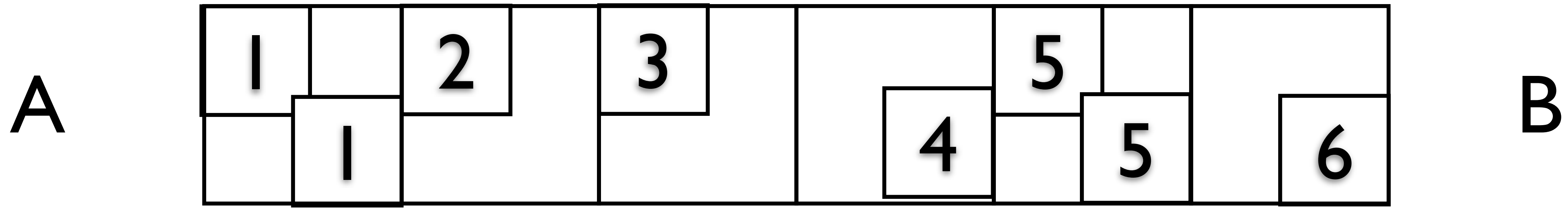
What is a significant limitation of hash-joins?

- It only support equality predicates

Implementing: Joins

Solution 5 (Hash)

1) Build a hash table on both relations



What is a significant limitation of hash-joins?

- It only support equality predicates

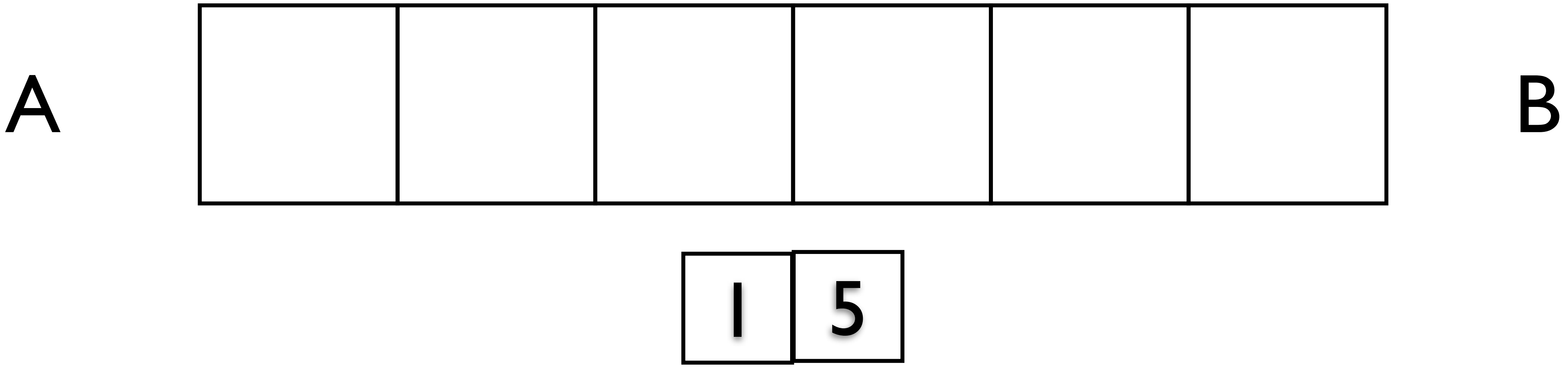
Implementing: Joins

Solution 5 (Hash)

1) Build a hash table on both relations

2) In-Memory Nested-Loop Join on each hash bucket

(subdivide buckets using a different hash fn if needed)



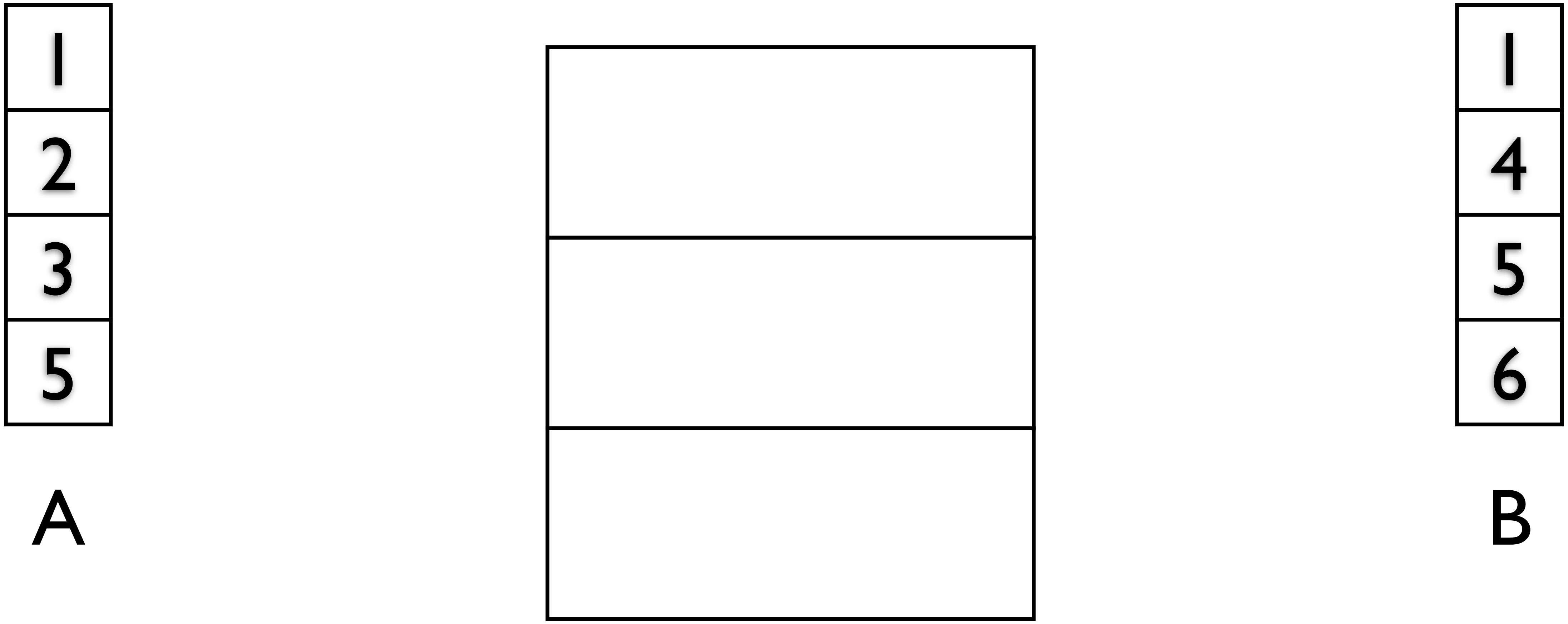
What is a significant limitation of hash-joins?

- It only support equality predicates

Implementing: Joins

Solution 6 (Hybrid Hash)

Keep the hash table in memory



1
2
3
5

A

1
4
5
6

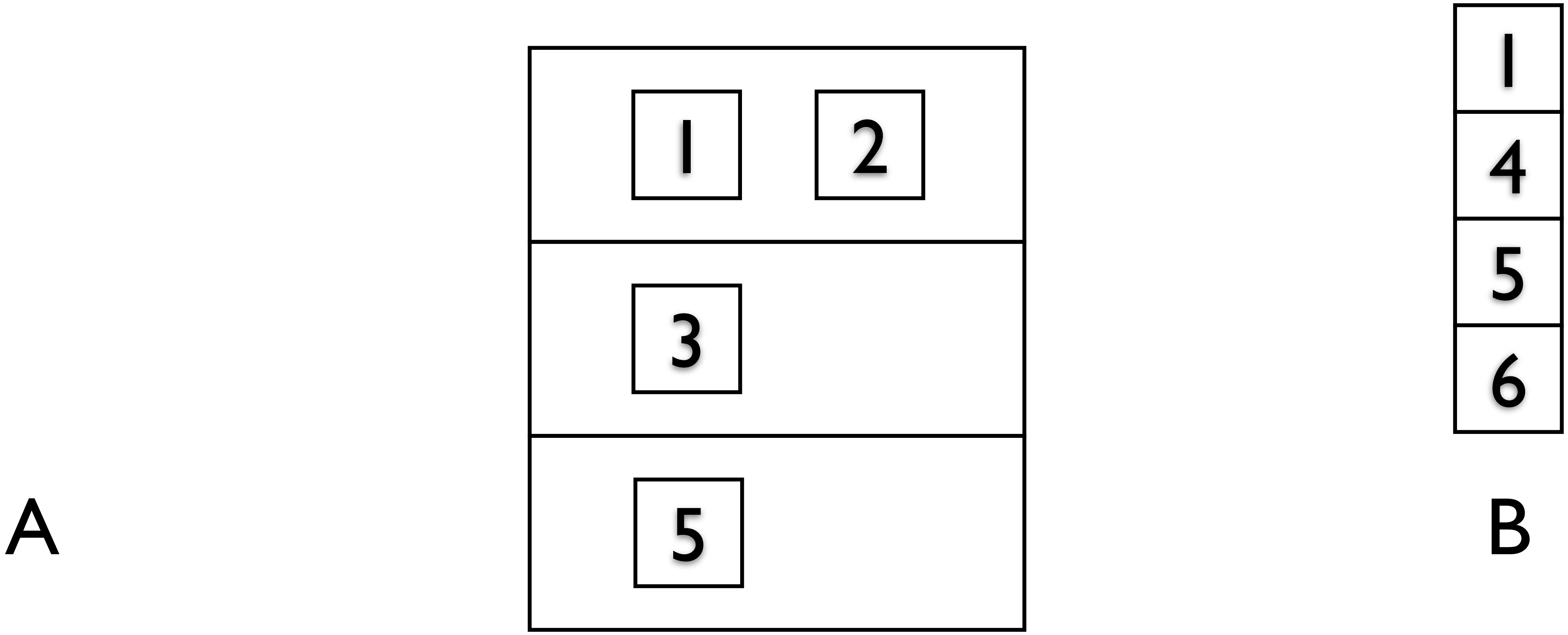
B

(Essentially a more efficient nested loop join)

Implementing: Joins

Solution 6 (Hybrid Hash)

Keep the hash table in memory

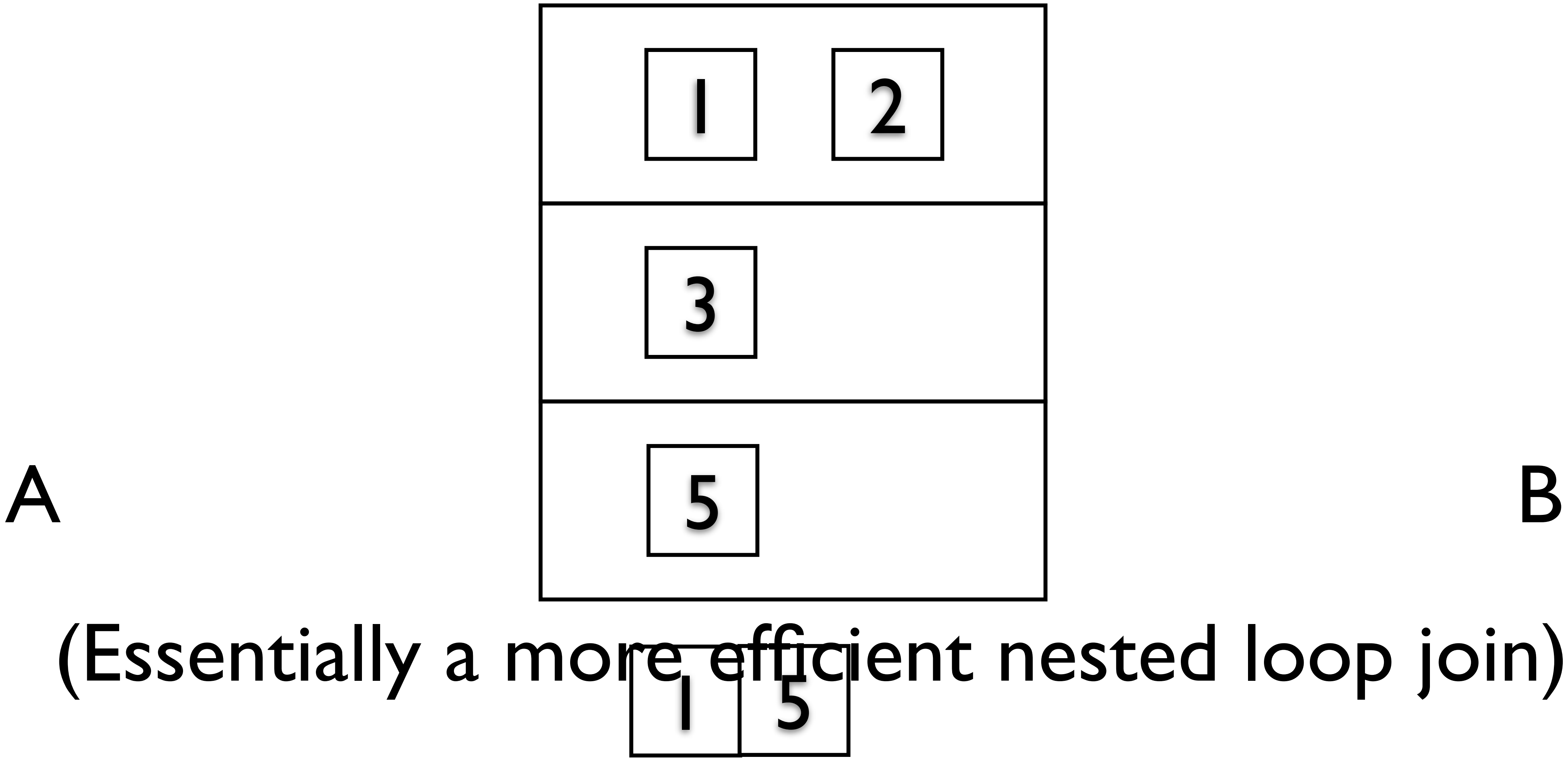


(Essentially a more efficient nested loop join)

Implementing: Joins

Solution 6 (Hybrid Hash)

Keep the hash table in memory



(Essentially a more efficient nested loop join)

Implementing: Joins

Tradeoffs

	Pipelined?	Memory Requirements?	Predicate Limitation?
Nested Loop	1/2	1 Table	No
Block-Nested Loop	No	2 'Blocks'	No
Index-Nested Loop	1/2	1 Tuple (+Index)	Single Comparison
Sort-Merge	If Data Sorted	Same as reqs. of Sorting Inputs	Equality Only
Hash	No	Max of 1 Page per Bucket and All Pages in Any Bucket	Equality Only
Hybrid Hash	1/2	Hash Table	Equality Only