CS143: Joins

Professor Junghoo “John” Cho

Motivation

- Q: How do we process
  SELECT * FROM Student WHERE sid > 30?

- Q: How do we process
  SELECT * FROM Student S, Enroll E WHERE S.sid = E.sid?

$R \bowtie S$?

R

A	
40	T1
60	T2
30	T3
10	T4
20	T5

S

A	
10	T6
60	T7
40	T8
20	T9

Four Join Algorithms

- Nested-Loop Join (NLJ)
- Index Join (IJ)
- Sort-Merge Join (SMJ)
- Hash Join (HJ)

Nested-Loop Join (NLJ)

For each r $\in$ R:

  For each s $\in$ S:

    if r.A = s.A, then output (r,s)

R

40	T1
60	T2
30	T3
10	T4
20	T5

S

10	T6
60	T7
40	T8
20	T9

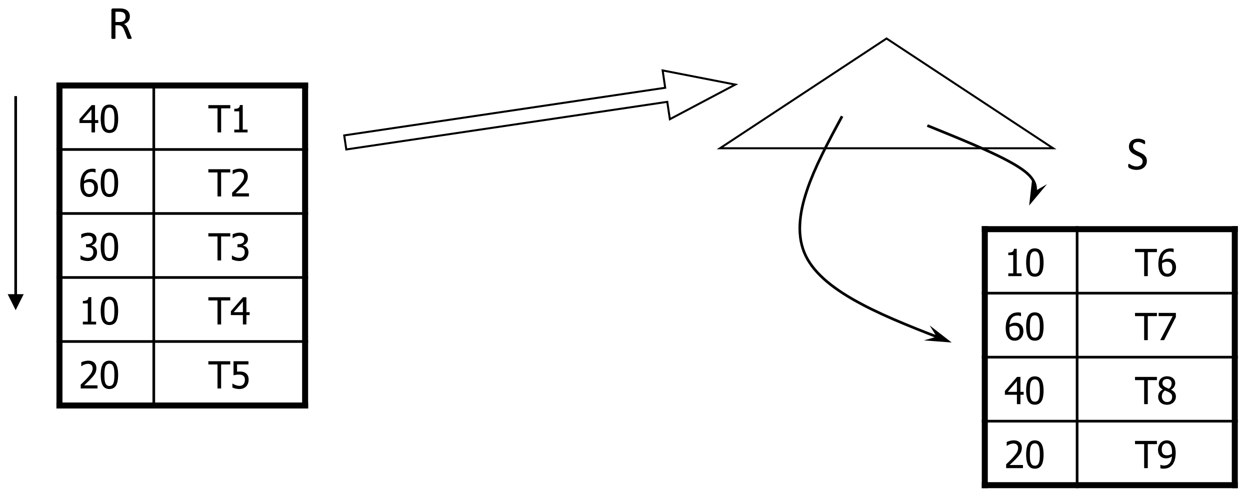
Index Join (IJ)

(1) Create an index for S.A if needed

(2) For each $r \in R$:

X := lookup index on S.A with r.A value

For each $s \in X$, output (r,s)



Sort-Merge Join (SMJ)

- Sort the relations first, then join

R

10	T4
20	T5
30	T3
40	T1
60	T2

S

10	T6
20	T9
40	T8
60	T7

Sort-Merge Join (SMJ)

(1) if not, sort R and S by A

(2) $i \leftarrow 1$; $j \leftarrow 1$;

while $(i \leq |R|) \wedge (j \leq |S|)$:

if (R[i].A = S[j].A) then output (R[i], S[j]); $i \leftarrow i+1$; $j \leftarrow j+1$;

else if (R[i].A > S[j].A) then $j \leftarrow j+1$

else if (R[i].A < S[j].A) then $i \leftarrow i+1$

R

10	T4
20	T5
30	T3
40	T1
60	T2

S

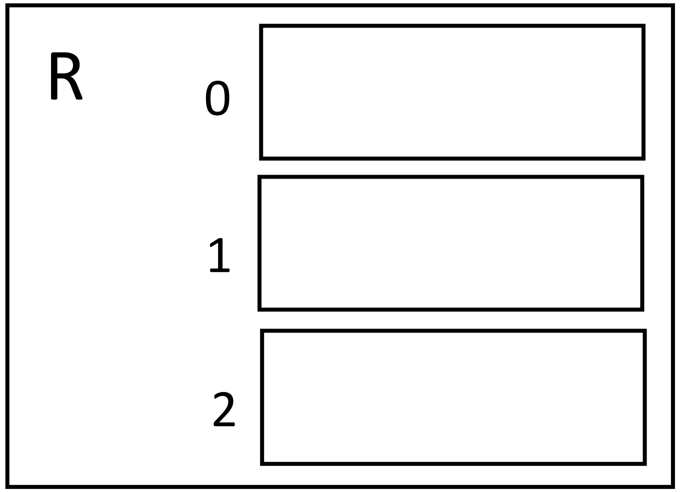
10	T6
20	T9
40	T8
60	T7

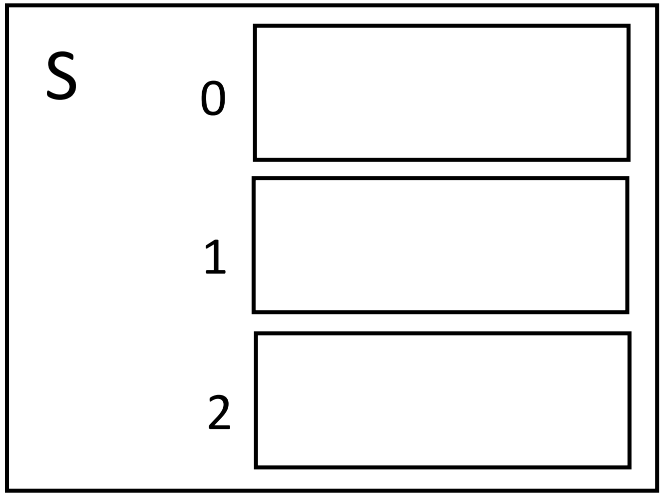
Hash Join (HJ)

- Hash function: $h(v) \to [1, k]$
- Q: Given ($r \in R$) and ($s \in S$), can r and s join if $h(r.A) \neq h(s.A)$?
- Main idea
  - Partition tuples in R and S based on hash values on join attributes
  - Perform “joins” only between partitions of the same hash value

Hash Join (HJ)

- H(k) = k mod 3





40	T1
60	T2
30	T3
10	T4
20	T5

10	T6
60	T7
40	T8
20	T9

Hash Join (HJ)

Hash function: h(v) $\rightarrow$ [1, k]

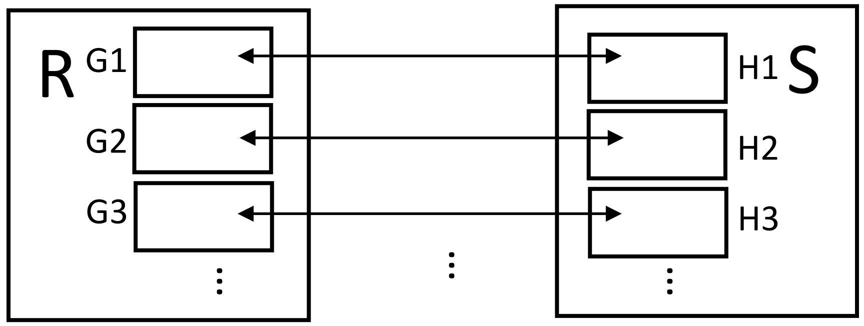
(1) Hashing stage (bucketizing): hash tuples into buckets

- Hash R tuples into G1,…,Gk buckets
- Hash S tuples into H1,…,Hk buckets

(2) Join stage: join tuples in matching buckets

- For i = 1 to k do

    match tuples in Gi, Hi buckets



Comparison of Join Algorithms

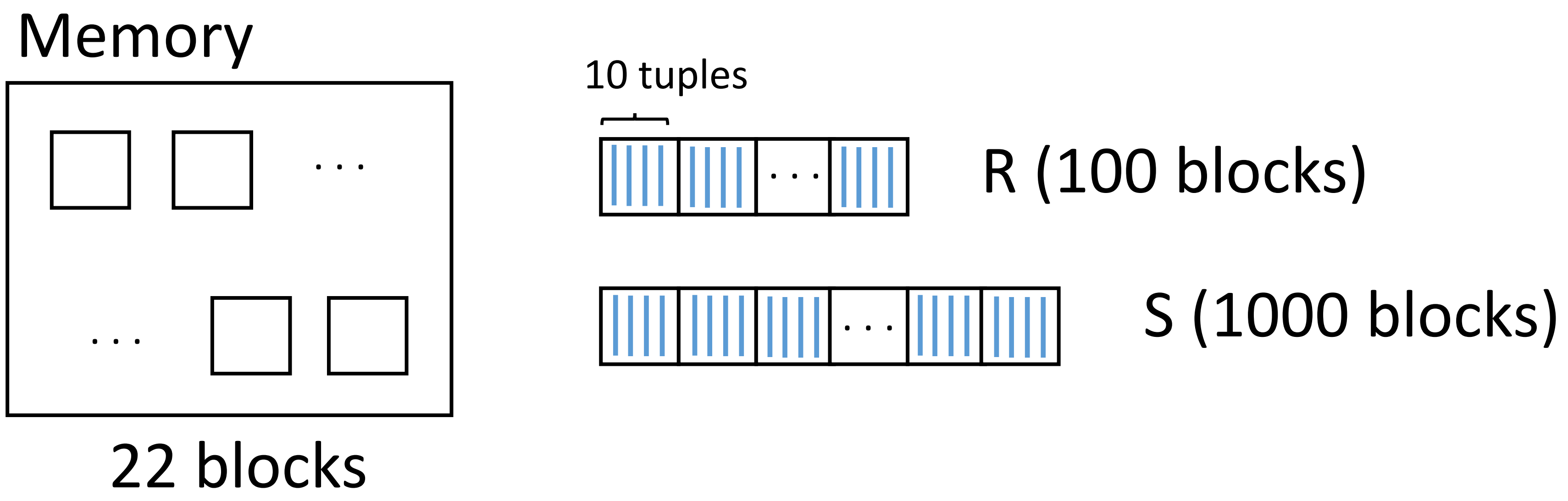
- Q: Which algorithm is better?
  - Q: What does “better” mean?
- Ultimate bottom line: Which algorithm is the “fastest”?
  - Q: How does the system know which algorithm runs fast? Run all join algorithms and pick the fastest one?

Cost Model

- A model to estimate the performance of a join algorithm
  - Multiple cost models are possible depending on their sophistication
- Our cost model: ***# disk blocks that are read/written during join***
  - Not perfect: ignores random vs sequential IO differnce, CPU cost, …
  - But simple to analyze
  - And “good enough” to pick the best join algorithm
    - Cost of join is dominated by disk IO
    - Most join algorithms have similar disk access pattern
  - ***Our cost model ignores the last IO for writing the final result***
    - This cost is the same for all algorithms

Running Example

- Join two tables: R ⋈ S
- |R| = 1,000 tuples, |S| = 10,000 tuples
- b_R= 100 blocks, b_S = 1,000 blocks (10 tuples/block)
- M = main memory "cache" 22 disk blocks



Cost of Join Algorithms

	Cost	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

Sort-Merge Join (SMJ)

(1) if not, sort R and S by A

(2) $i \leftarrow 1$; $j \leftarrow 1$;

while $(i \leq |R|) \wedge (j \leq |S|)$:

if (R[i].A = S[j].A) then output (R[i], S[j]); $i \leftarrow i+1$; $j \leftarrow j+1$;

else if (R[i].A > S[j].A) then $j \leftarrow j+1$

else if (R[i].A < S[j].A) then $i \leftarrow i+1$

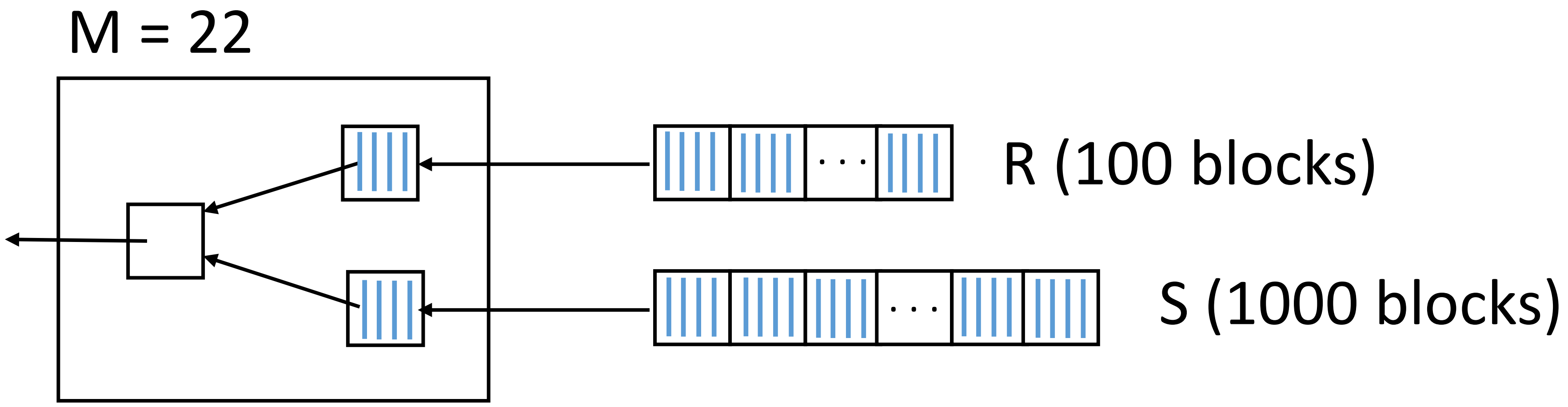
R

10	T4
20	T5
30	T3
40	T1
60	T2

S

10	T6
20	T9
40	T8
60	T7

Cost of Join Stage of Sort-Merge Join

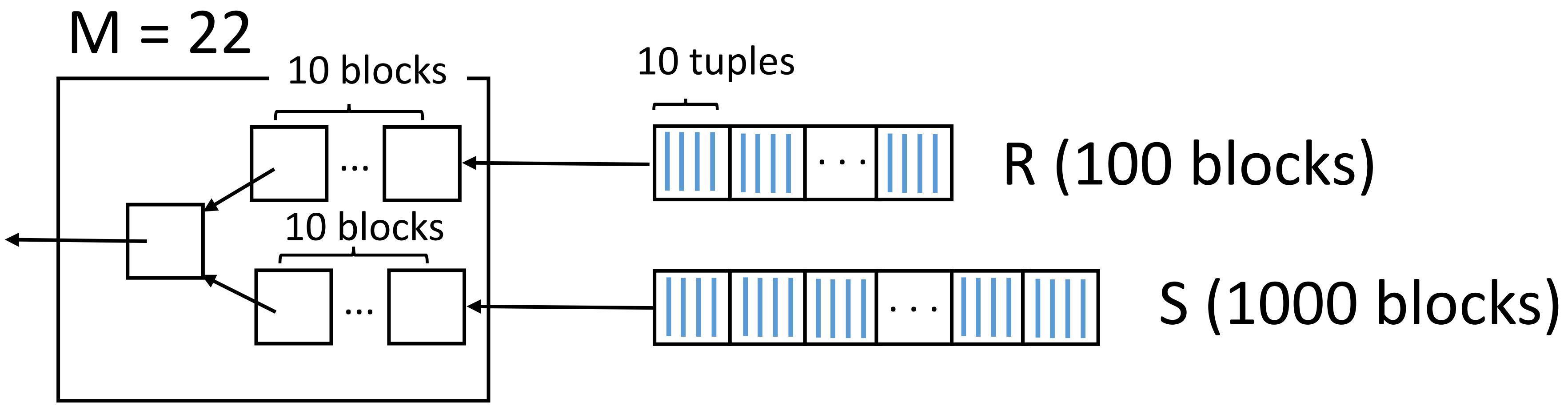


Q: Ignoring the final write of output, how many disk blocks are read during join?

Q: We only used 3 memory blocks. Can we use the rest to make things better?

What About?

- Q: Will this lead to fewer disk block reads?



Cost of Join Algorithms

	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

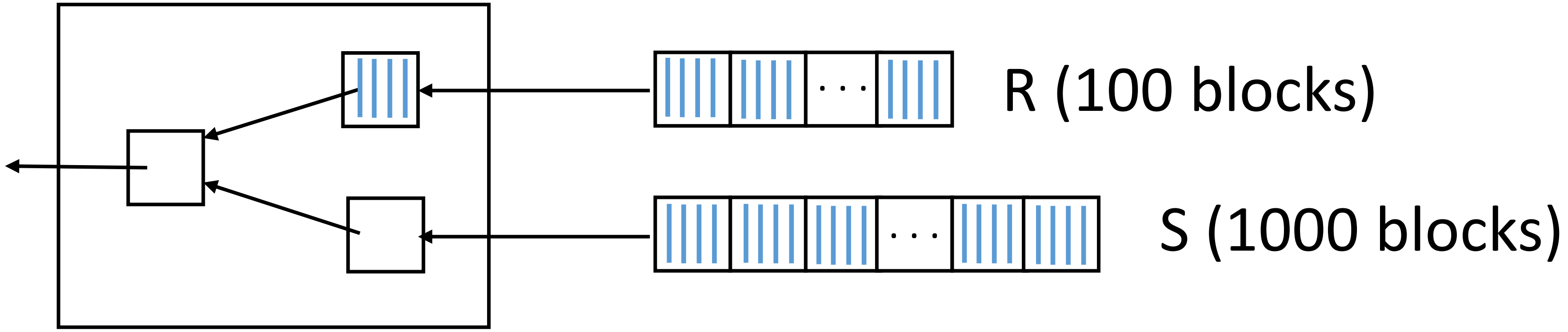
Nested-Loop Join (NLJ): R ⋈ S

For each $r \in R$:

    For each $s \in S$:

        if r.A = s.A, then output (r,s)

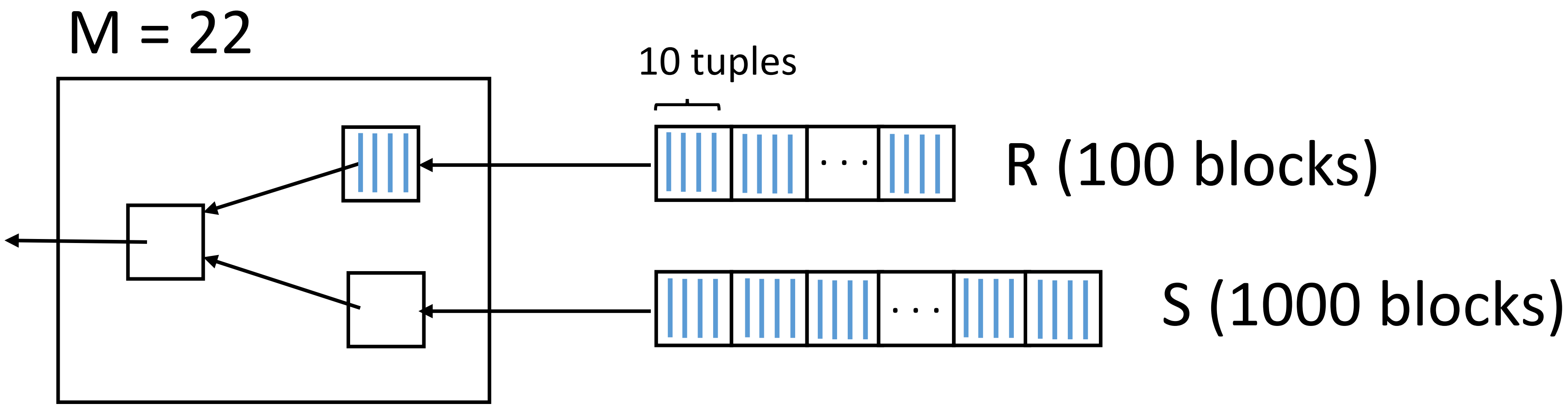
M = 22



Scan S table once for every tuple of R

Nested Loop Join

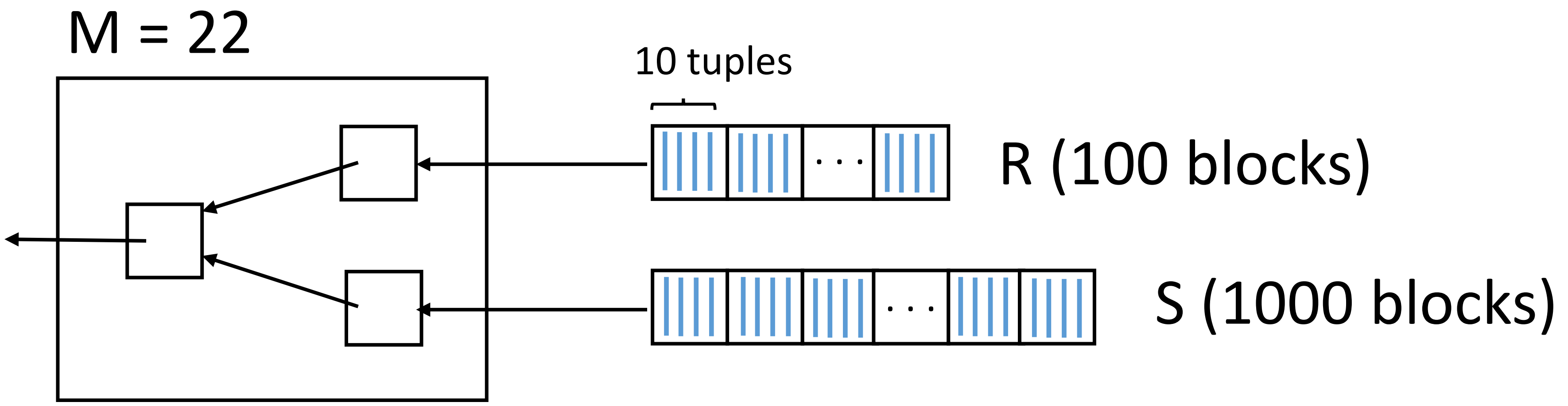
- Scan S table once for every tuple of R



- Q: Can we do better?

Block Nested Loop Join

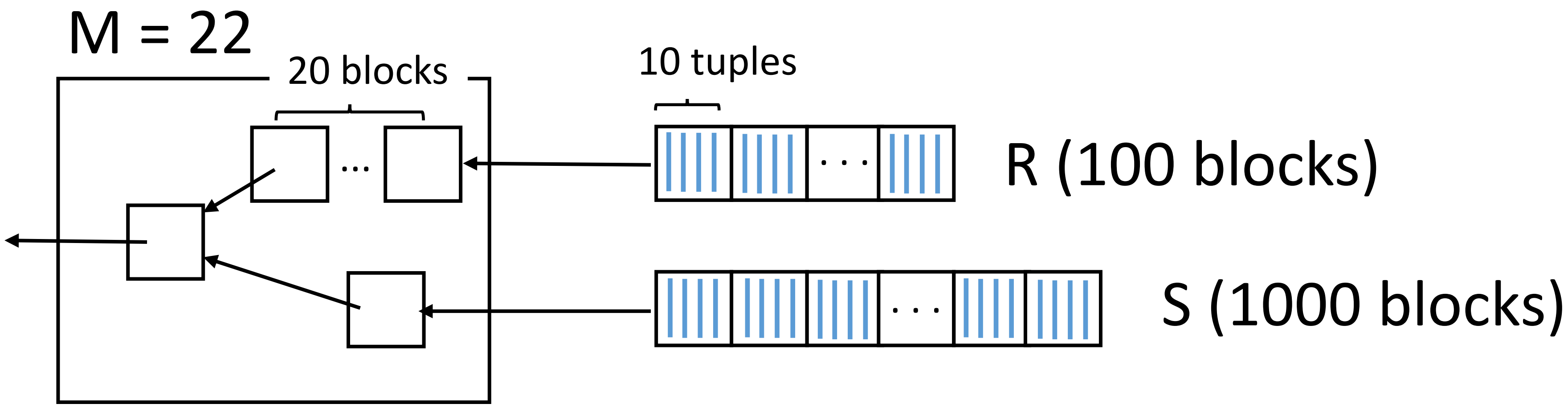
- Scan S table once for every ***block*** of R



- Q: Can we do even better? What is the maximum # of blocks that we can read in one batch from R?

Block Nested Loop Join

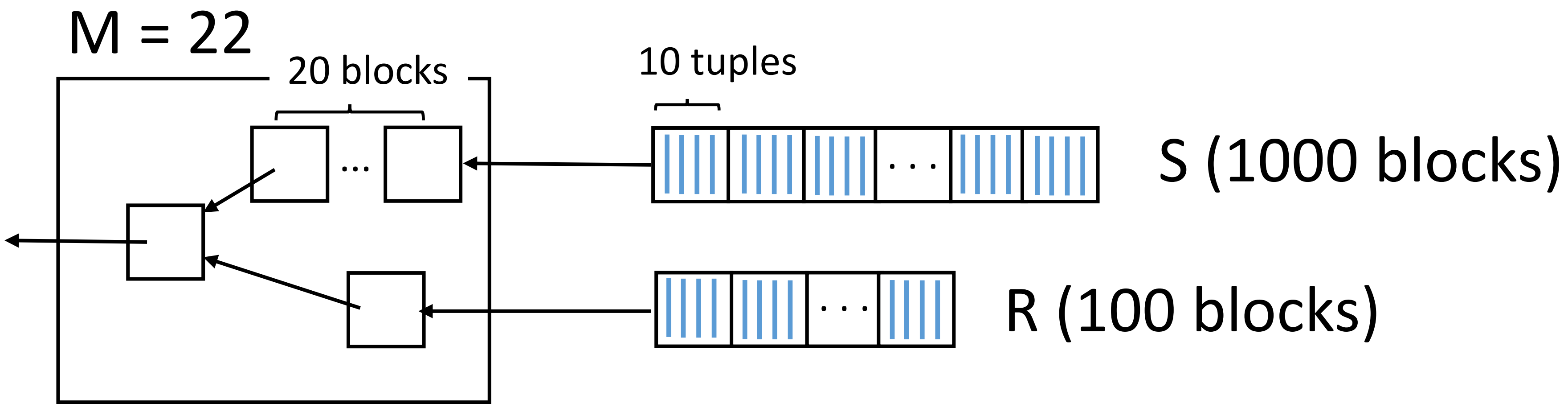
- Scan S table once for every ***20 blocks*** of R



- Q: What if we read S first?

Block Nested Loop Join

- Scan R table once for every 20 blocks of S



Cost of Join Algorithms

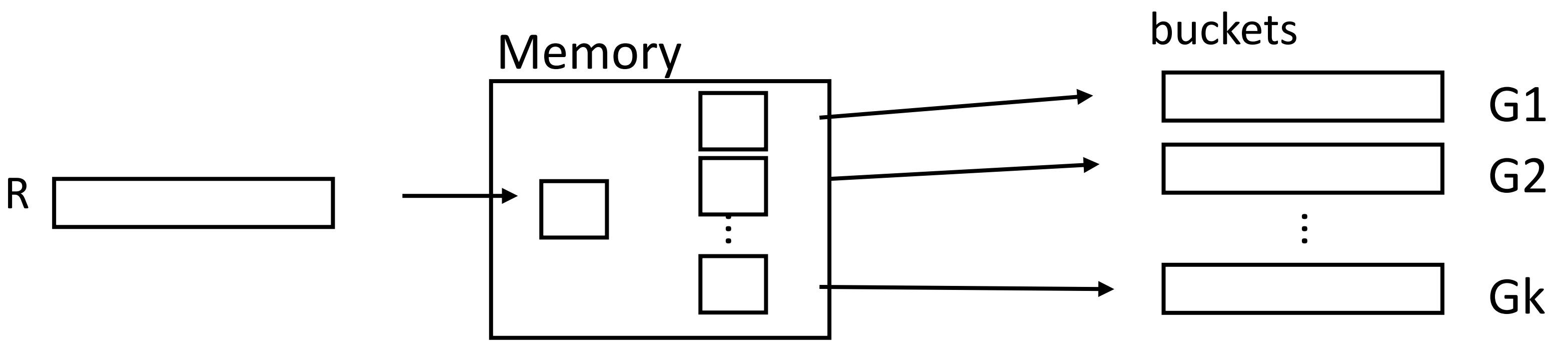
	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

Nested Loop Join Summary

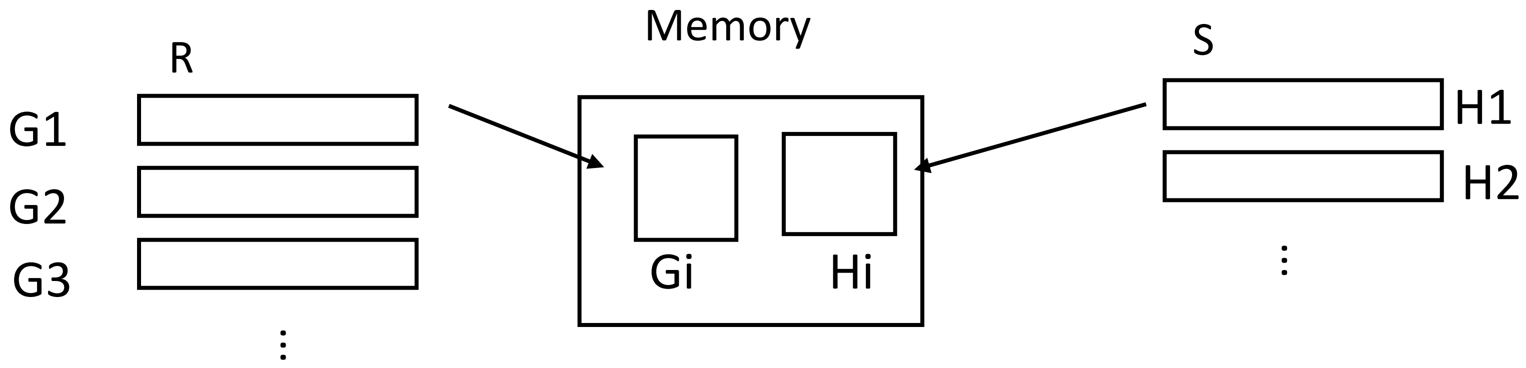
- Always use block nested loop join (not the naïve algorithm)
- Read as many blocks as possible for the left table in one iteration
- Use the smaller table on the left (i.e., outer loop)

Hash Join (HJ)

- Step (1): Hashing stage: h(v) → [1, k]

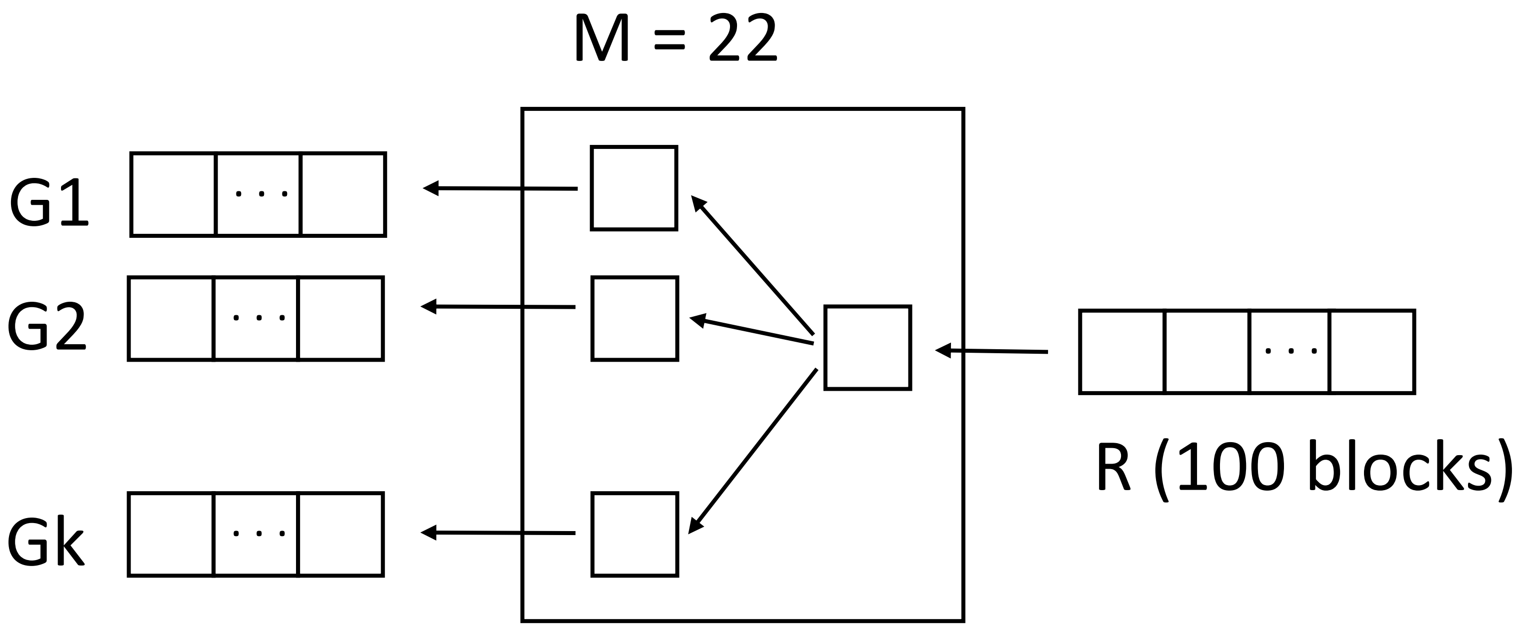


- Step (2): Join stage



HJ: Bucketizing Stage

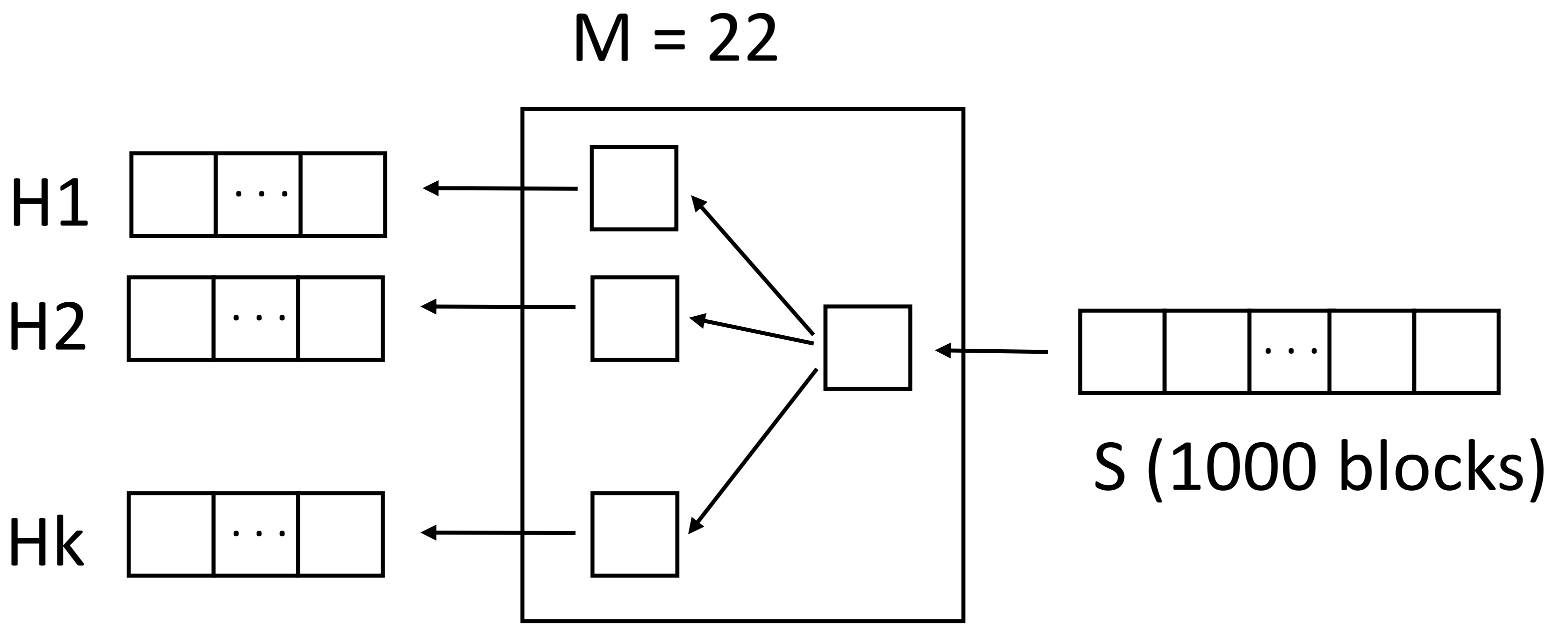
- Read R table and hash them into k buckets



- Q: Given M=22, what is the maximum k?
- Q: How many disk IOs to bucketize R?

HJ: Bucketizing Stage

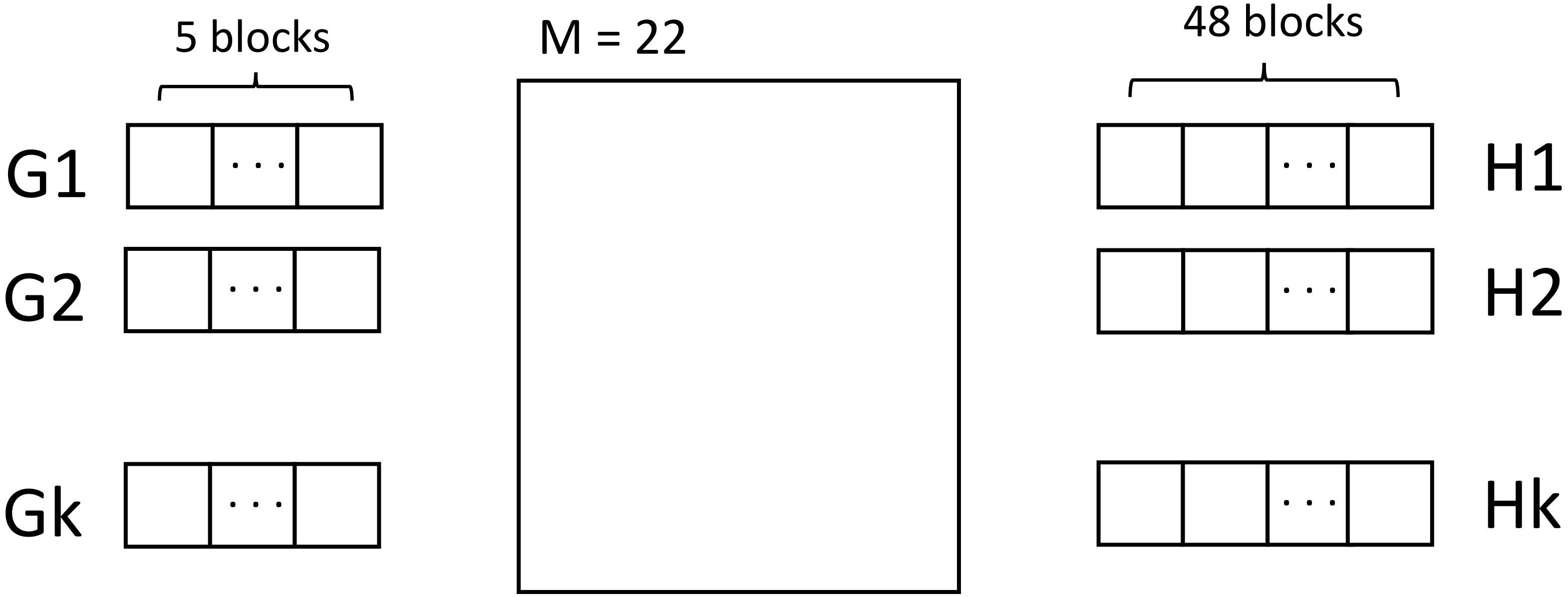
- Read S table and hash them into k buckets



- Q: In general, what is the cost for bucketizing R and S?

HJ: Join Stage

- Join tuples in Gi with those in Hi



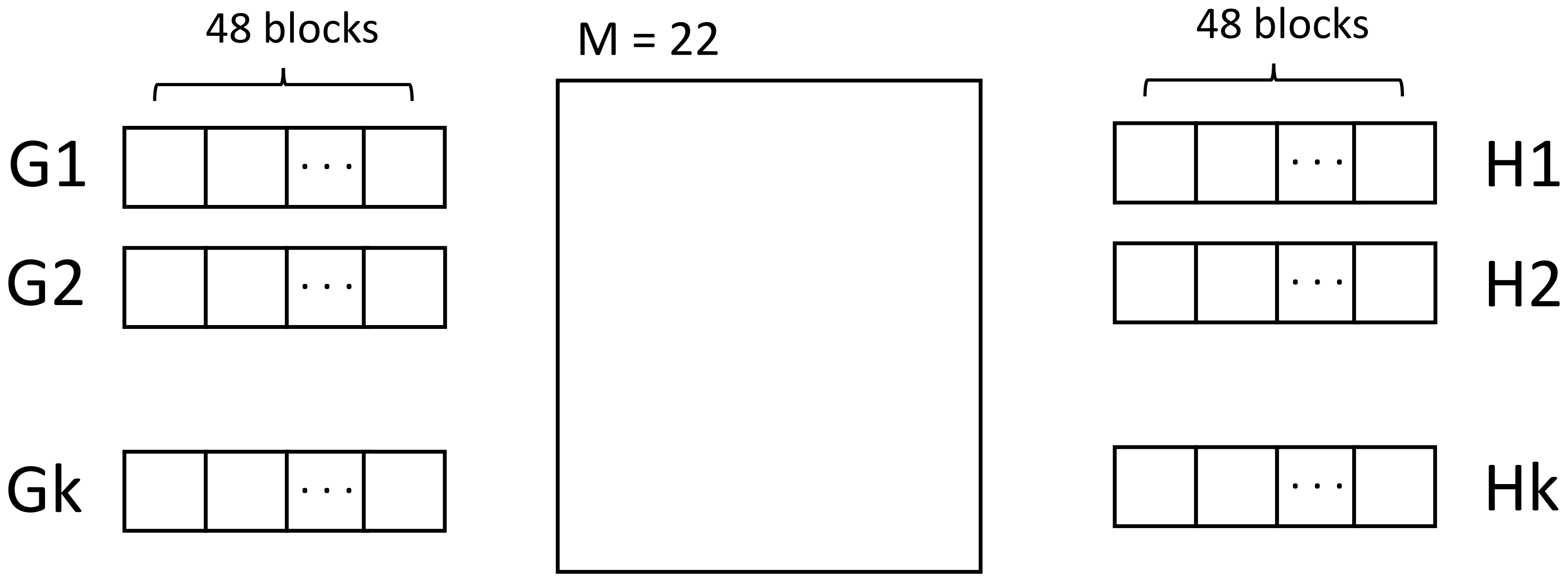
- Q: How can we join tuples in G1 with H1? How should we use memory?

Cost of Join Algorithms

	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

HJ: Join Stage

- Q: What if R is large, say $b_R = 1000$, and Gi > 20?



- A: Exactly the same as standard join problem. Apply "hash join" algorithm to join H1 and G1
  - Apply "hash join" algorithm using a new hash function!

HJ: Recursive Partitioning

- Use a new hash function h'(v) → [1, k] to recursively partition Gi and Hi to even smaller partitions (until one of them fit in main memory)
- # of bucketizing steps needed for R: $\left\lceil \log_{\mathrm{M}-1} \frac{b_R}{M-2} \right\rceil$
  - In each bucketing steps, we perform $2(b_R + b_S)$ disk IOs

Cost of Join Algorithms

	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

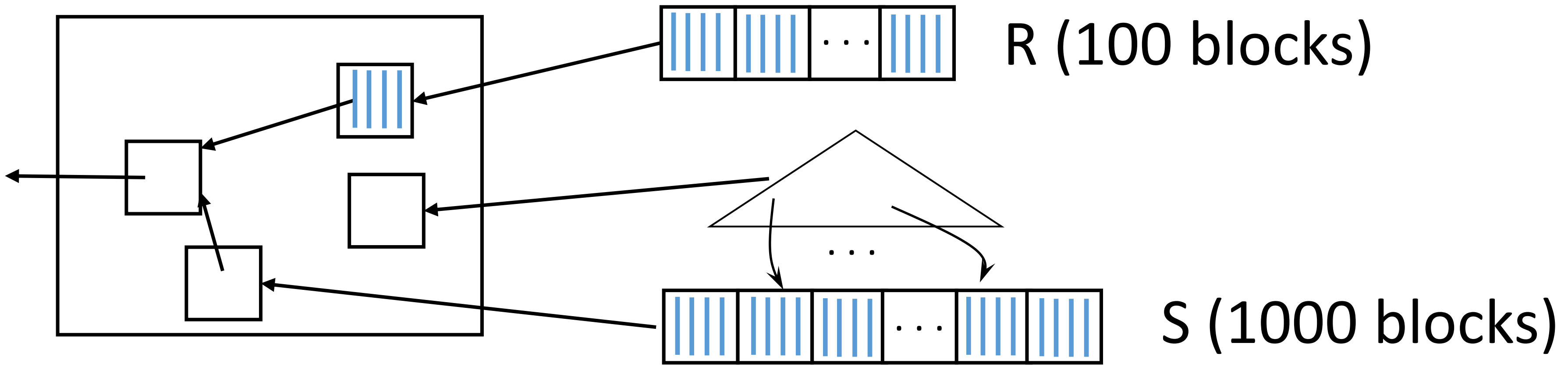
Index Join (IJ): R ⋈ S

For each r ∈ R:

X := lookup index on S.A with r.A value

For each s ∈ X, output (r,s)

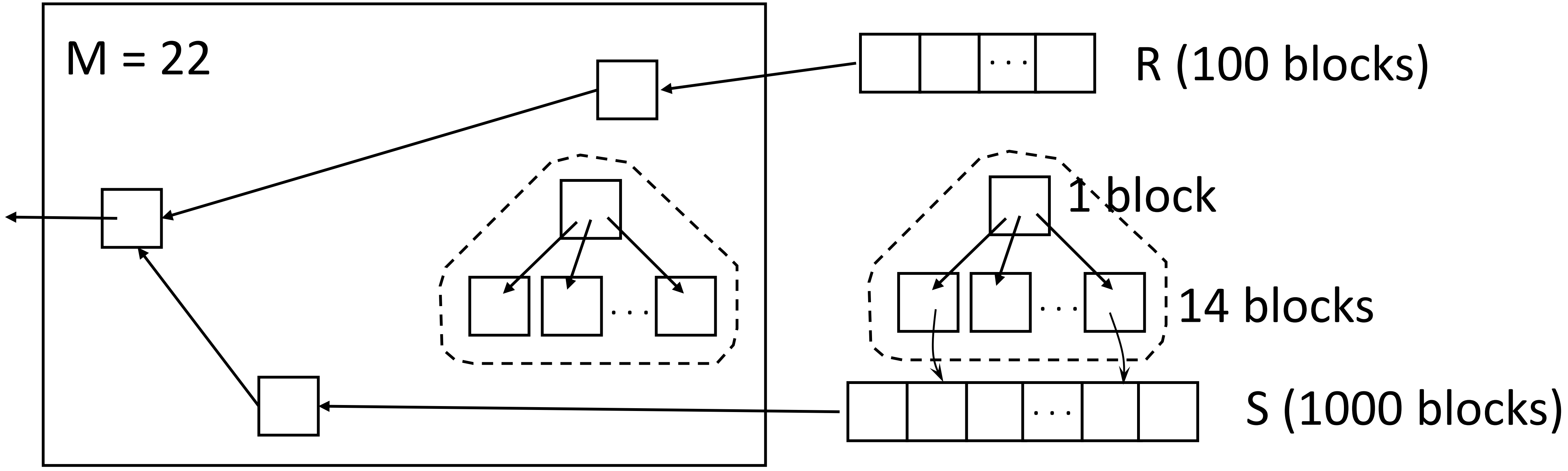
M = 22



- Cost = IOs for (R scan + index look up + tuple read from S)

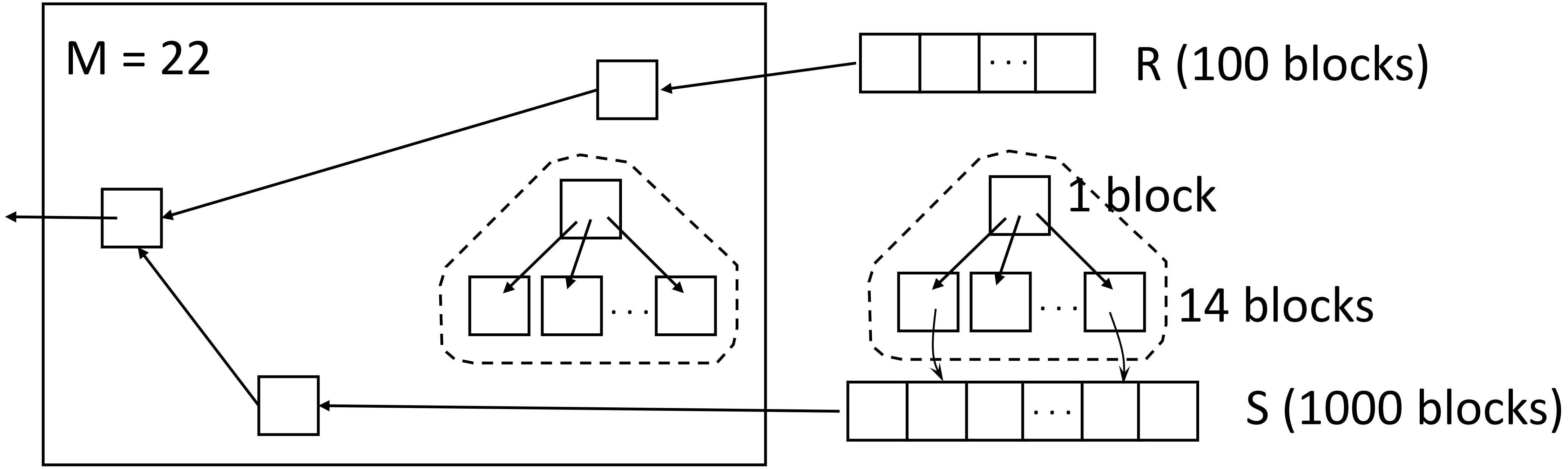
IJ Example (1)

- 15 blocks for index
  - 1 root 14 leaf
- On average, 1 matching S tuple per an R tuple
- Q: How many disk IOs? How should we use the memory?



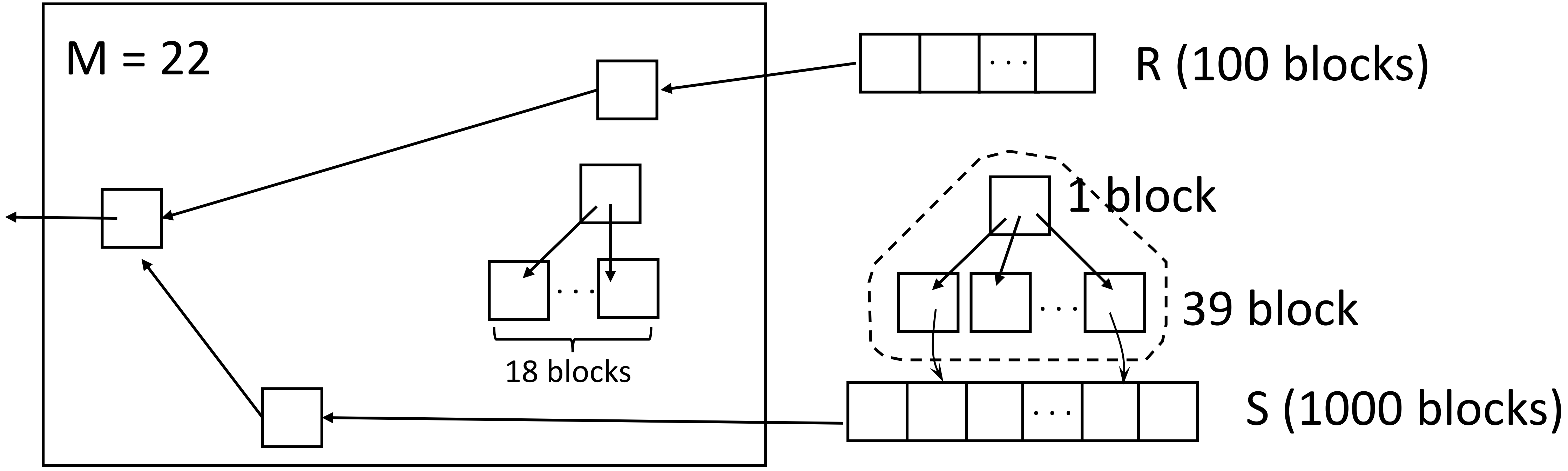
IJ Example (1)

- Cost for R scan:
- Cost for Index look up:
- Cost for read matching S tuple:



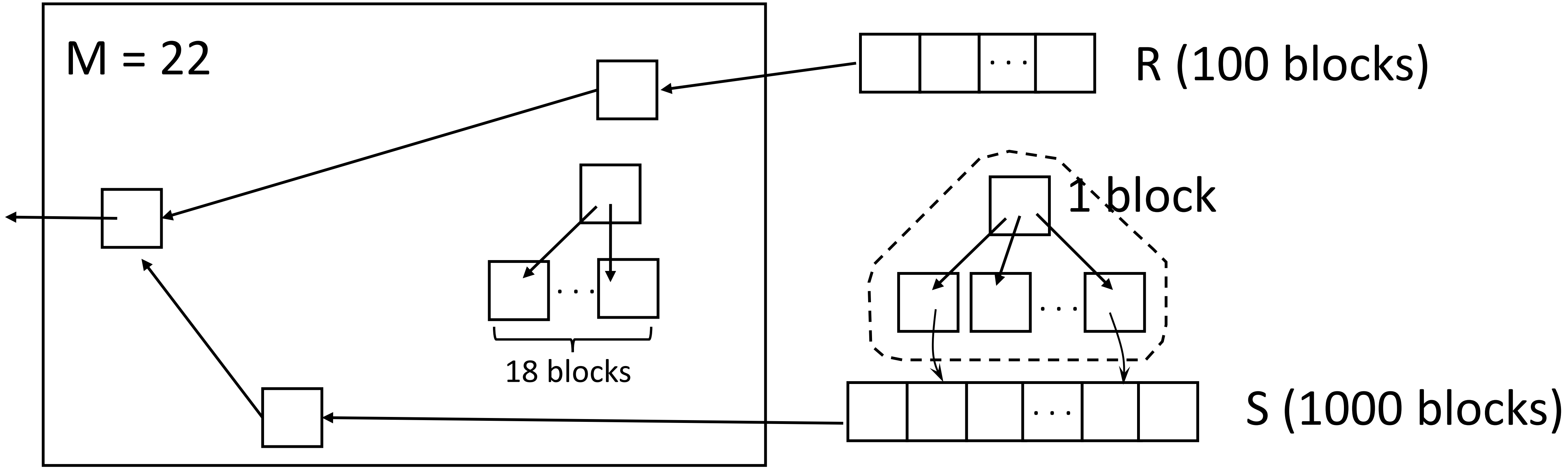
IJ Example (2)

- 40 blocks for index
  - 1 root 39 leaf
- On average, 10 matching S tuple per an R tuple
- Q: How many disk IOs? How should we use the memory?



IJ Example (2)

- Cost for R scan:
- Cost for Index look up:
- Cost for read matching S tuple:



Cost of Join Algorithms

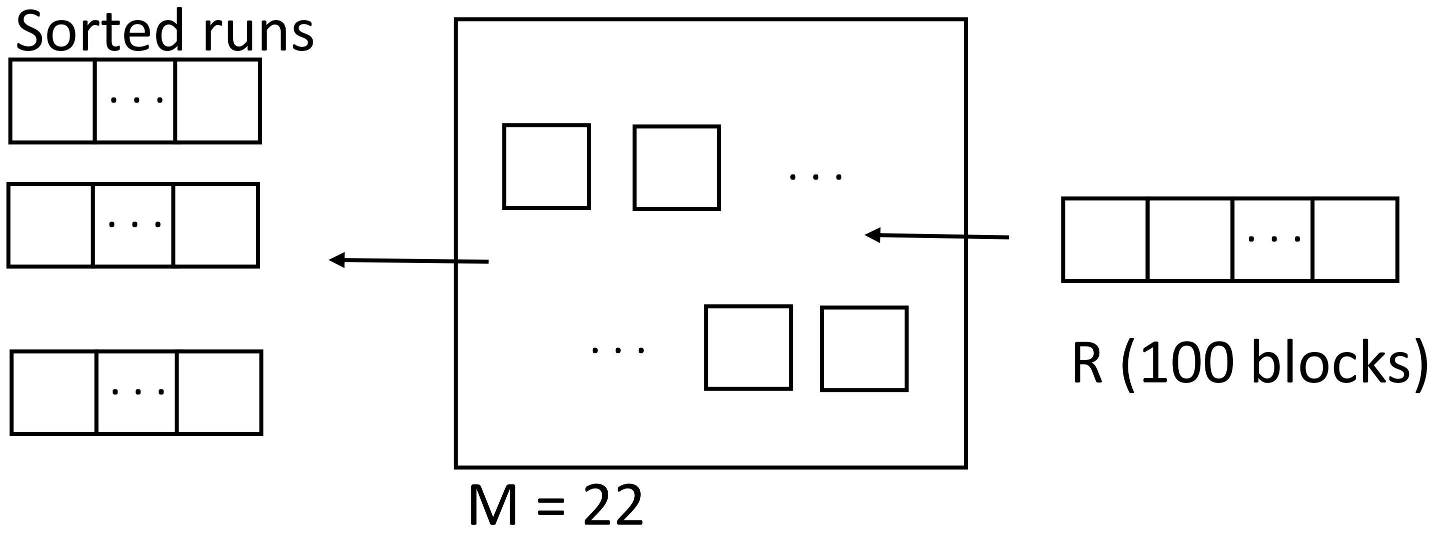
	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

SMJ: Cost of Sorting

- Sort-Merge Join
  1. Sort stage: Sort R and S
  2. Join stage: Join sorted R and S
- Q: How many disk IOs during sort stage?

SMJ: Cost of Sorting

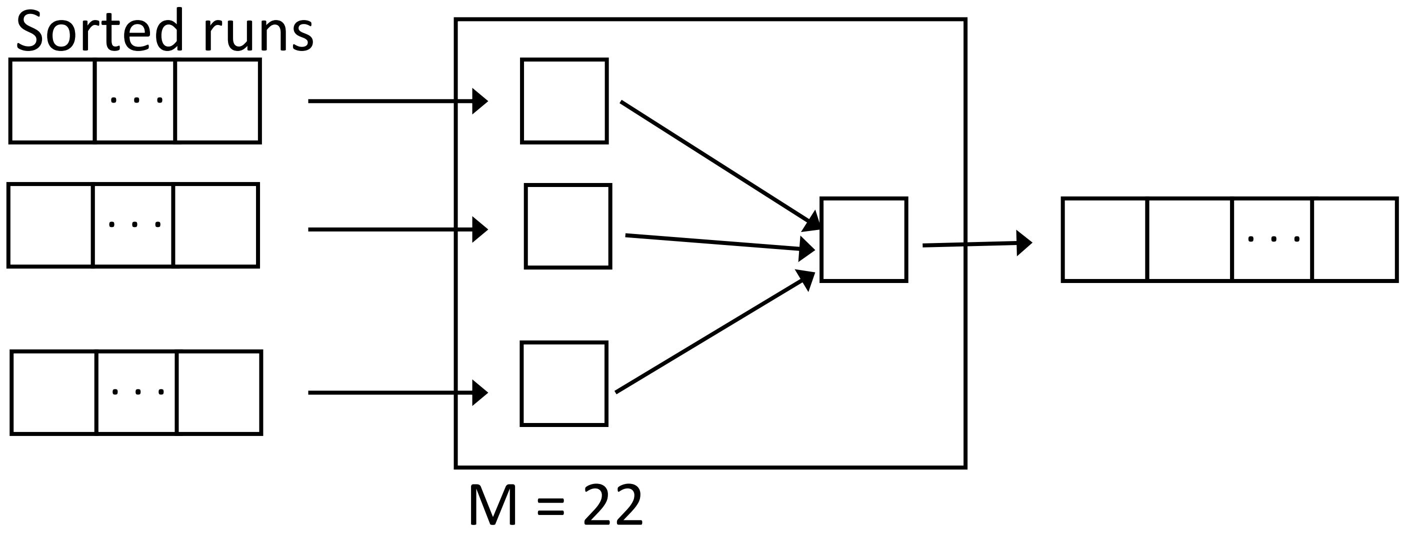
- Q: How can we sort R?



- Q: How many blocks can we sort in each batch?
  - Do we need to allocate one block for output?
- Q: How many sorted runs?

SMJ: Cost of Sorting

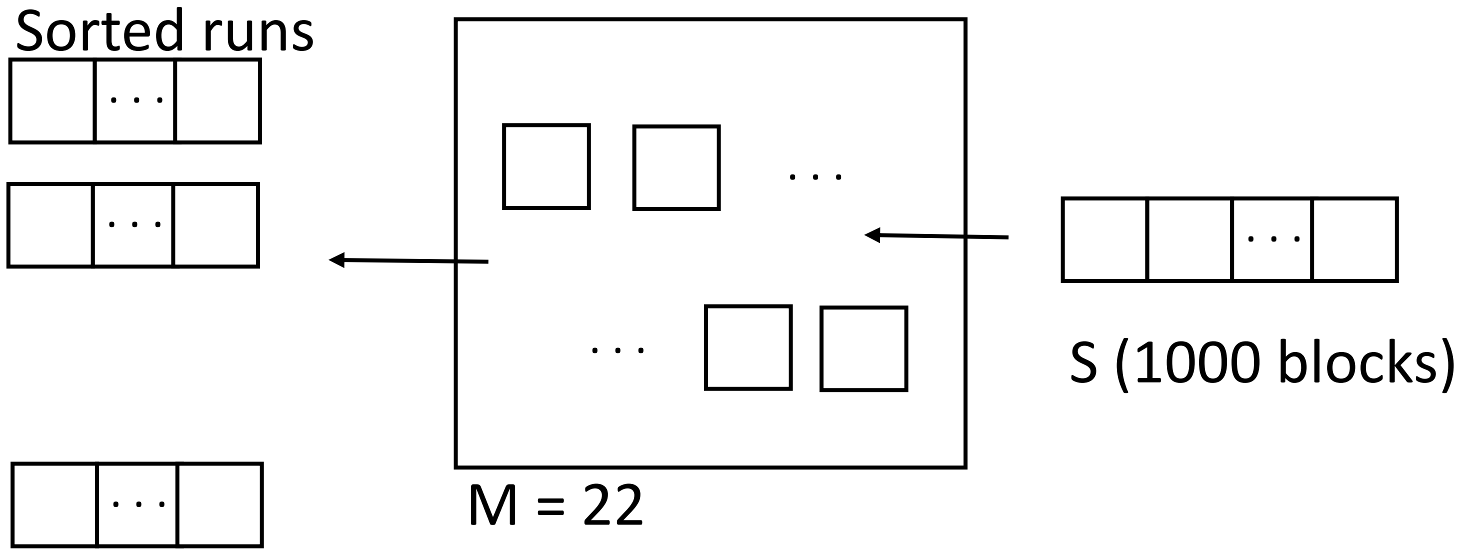
- Q: What to do with sorted runs?



- Q: How many disk IOs during the "merge step" of sort?
- Q: Total IOs for sorting R?

SMJ: Cost of Sorting

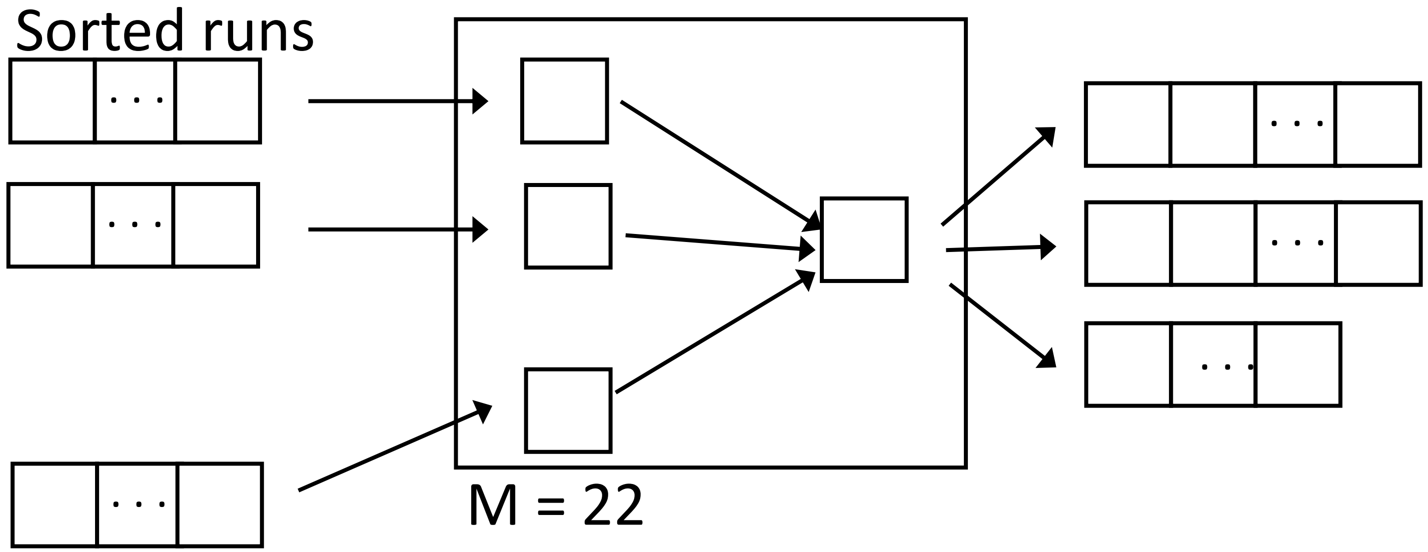
- Q: How can we sort S?



- Q: How many sorted runs are produced from S?

SMJ: Cost of Sorting

- Q: How many sorted runs can we merge at a time?



- Q: What to do with the produced sorted runs?

SMJ: Cost of Sorting

- Q: How many ”merging” steps are needed to sort S?
  - 1 initial sorting
  - 2 merging steps of sorted runs
  - 2,000 disk IO’s per each sorting/merging step
  - 6,000 total disk IO’s to sort S table
- In general, to sort R of b_R blocks with M memory buffers, we need
  - 1 initial sorting
  - $\left\lceil \log_{M-1}(\frac{b_R}{M}) \right\rceil$ subsequent merging stages
  - 2 b_R disk IO’s per each sorting/merging stage
  - In total, $2b_R\left(\left\lceil \log_{M-1}(\frac{b_R}{M}) \right\rceil + 1\right)$ disk IO’s are needed

Cost of Join Algorithms

	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ		
SMJ		
HJ		
IJ		

Cost of Join Algorithms

	Cost (M=22, b_R =100, b_S=1000)	Formula ($b_R < b_S$)
NLJ	5,100	$b_R + \left\lceil \frac{b_R}{M-2} \right\rceil b_S$
SMJ	7,500 (if unsorted) 1,100 (if sorted)	$2b_R \left(\left\lceil \log_{M-1}(\frac{b_R}{M}) \right\rceil + 1 \right)$+ $2b_R \left(\left\lceil \log_{M-1}(\frac{b_R}{M}) \right\rceil + 1 \right)$+$(b_R$+$b_S)$
HJ	3,300	$2(b_R$+$b_S) \left\lceil \log_{\mathrm{M}-1} \frac{b_R}{M-2} \right\rceil + (b_R$+$b_S)$
IJ	1,115 – 10,640	$b_R + \vert R \vert (C + J)$ C: index lookup cost, J: # matching S tuples per R tuple

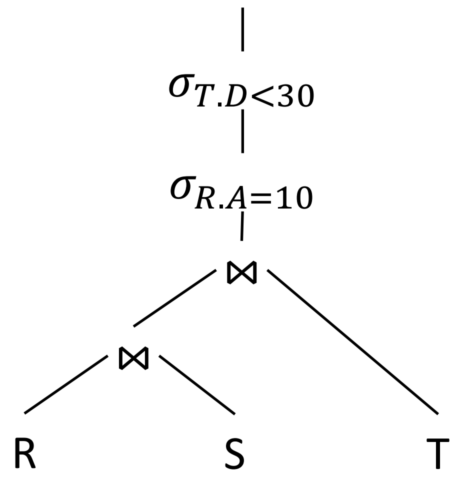
Summary of Joins

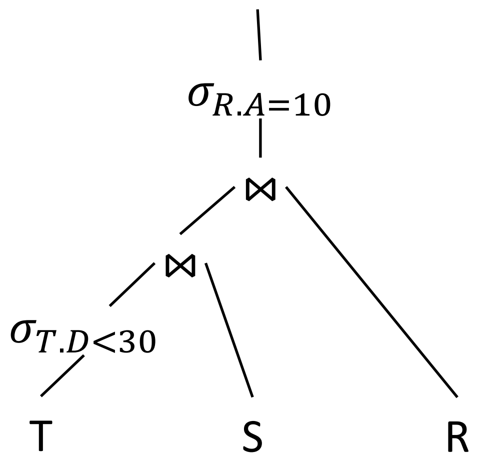
- Nested-loop join is OK for “small” relations (relative to memory size)
- Hash join is usually the best for equi-join
  - If tables have not been sorted and with no index
  - Consider merge join if tables have been sorted
  - Consider index join if index exists
- To pick the best, DBMS needs to maintain data statistics

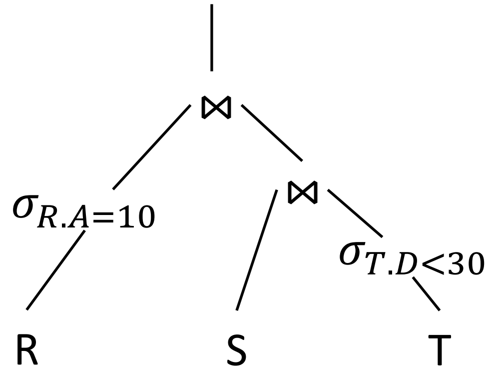
Query Optimization

- R(A, B)   S(B,C)   T (C,D):

  SELECT * FROM R, S, T
  WHERE R.B = S.B AND S.C = T.C AND R.A = 10 and T.D < 30
- Q: How can we process the above query?







Query Optimization

- Q: Focusing on just $R \bowtie S \bowtie T$, how many different ways?

- In general, for n way joins, $\frac{\left(2(n-1)\right)!}{(n-1)!}$ ways
    - For $n = 10$, $\frac{18!}{9!} = 17 \times 10^9$ different ways!!!

Query Optimization

- In reality, picking the very best is too difficult
- DBMS tries to avoid "obvious mistakes" using a number of heuristics to examine only those plans that are likely to be good
  - Put the smallest table on the left
  - "Left-deep" tree
  - Push selection as deep as possible
  - …
- For 90% of queries, DBMS picks a good query execution plan
  - To optimize the remaining 10%, companies pay big money to database consultants

Looking at Query Plan

- Many systems allow users to look at query plan
  - No SQL standard
  - Different systems use different syntax
- Examples
  - My SQL, PostgreSQL: EXPLAIN SELECT …
  - Oracle: EXPLAIN PLAN FOR SELECT …
  - MS SQL Server: SET SHOWPLAN_TEXT ON

Statistics Collection for DBMS

- “Cost-based optimizer”:
  - DBMS uses statistics on tables/indexes to pick the best query execution plan
  - Keeping correct stats is *very important.* Without correct stats, DBMS may do stupid things
- Oracle
  - ANALYZE TABLE <table> COMPUTE STATISTCS
  - ANALYZE TABLE <table> ESTIMATE STATISTICS ---- cheaper than COMPUTE
- DB2
  - RUN ON TABLE <userid>.<table> AND INDEXES ALL
- MySQL does not have a cost-based optimizer
  - Rule-based optimizer: Use simple heuristics only without looking at the actual data