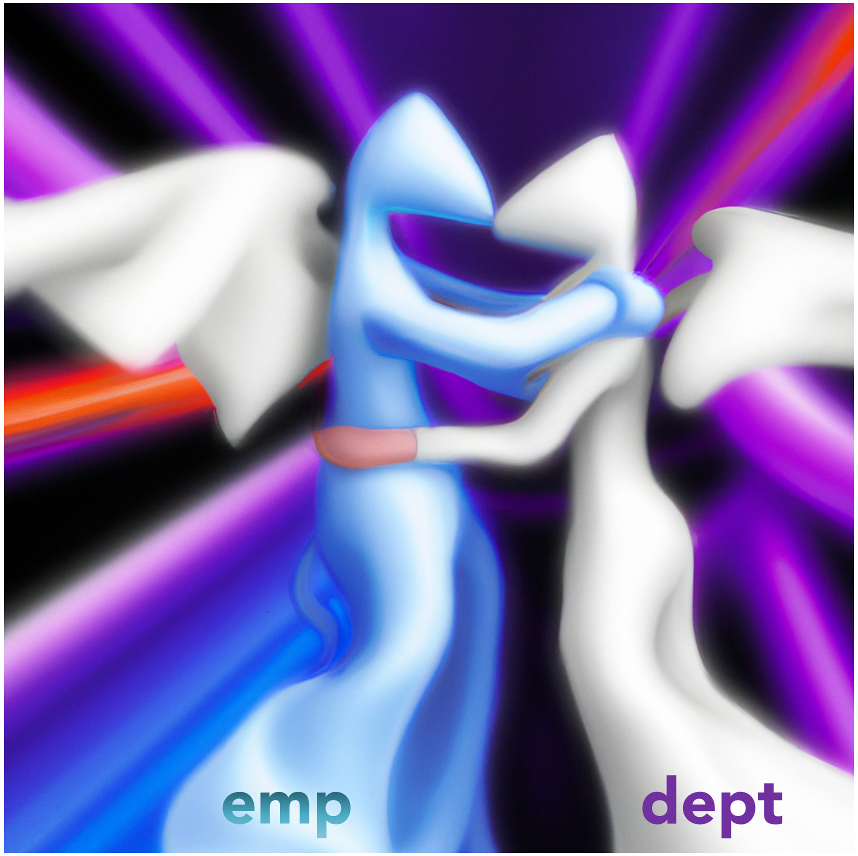
6.5830 Lecture 7



Join Algorithms

September 25, 2024

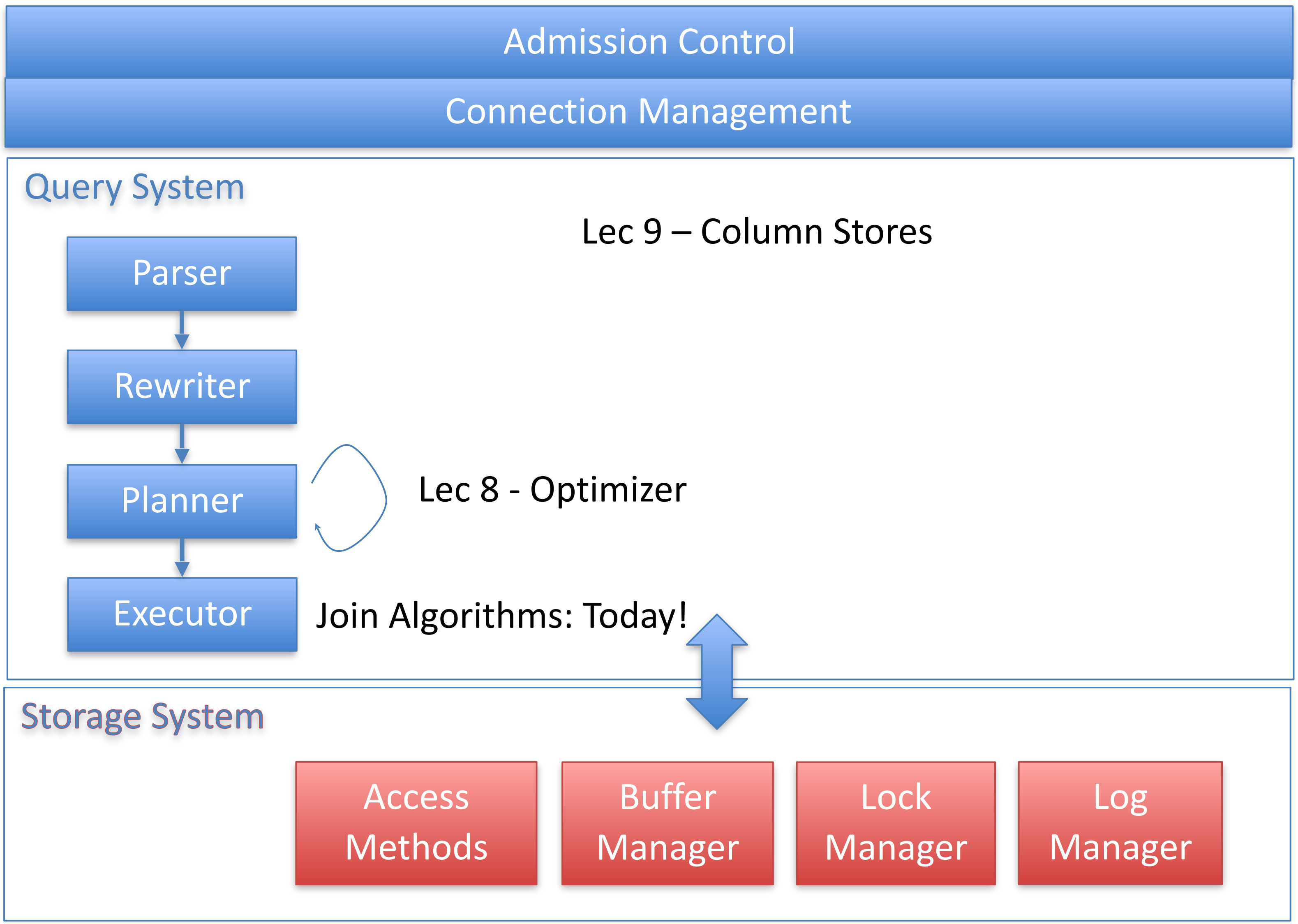
Administration

- Due today: **lab1, group assignment**
- PS2 out later today
- Information about project proposals will be out tomorrow
- Bootcamp + Quiz prep on 10/4 at 4pm in 32-D463

Important date changes:

- New quiz date 10/9, lecture 10 moves to 10/7
- Project proposal 10/11
- PS2 due 10/7

Plan for Next Few Lectures



Last Time: Access Methods

- Access method: way to access the records of the database
- 3 main types:
  - Heap file / heap scan
  - Hash index / index lookup
  - B+Tree index / index lookup / scan
- Many alternatives: e.g., R-trees
- Each has different performance tradeoffs

Indexes Recap

	Heap File	B+Tree	Hash File
Insert	O(1)	$O(\log_B n)$	O(1)
Delete	O(P)	$O(\log_B n)$	O(1)
Scan	O(P)	$O(\log_B n + R)$	-- / O(P)
Lookup	O(P)	$O(\log_B n)$	O(1)

n : number of tuples

P : number of pages in file

B : branching factor of B-Tree

R : number of pages in range

B+Trees are Inappropriate For Multi-dimensional Data

- Consider points of the form (x,y) that I want to index
- Suppose I store tuples with key (x,y) in a B+Tree
- Problem: can't look up y's in a particular range without also reading x's

- Two multidimension indexes: R-Tree & QuadTree

Example Index with Key = X, Y

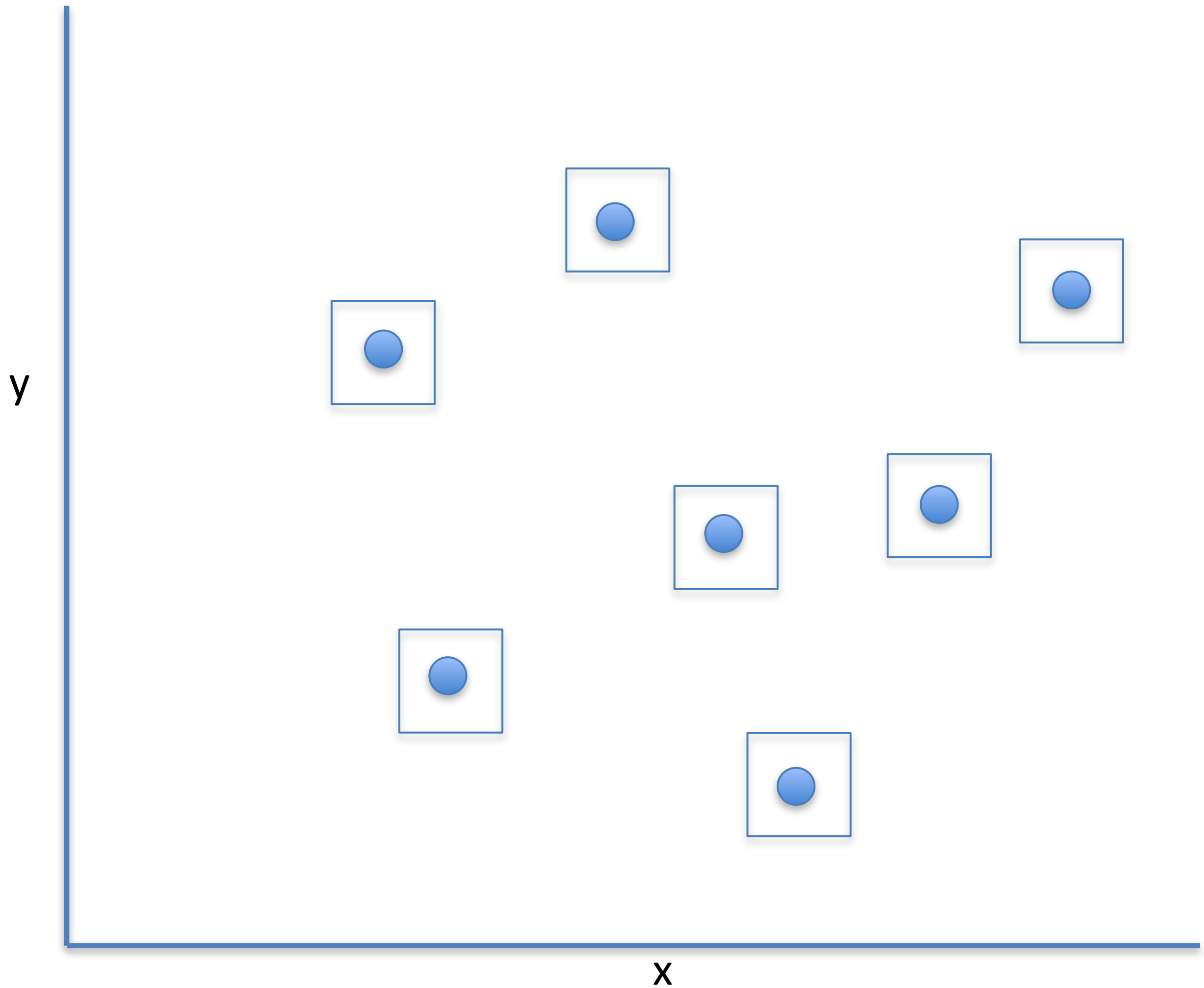
Index sorts data on X, then Y

X	Y
1	2
1	3
1	5
3	12
4	3
4	9
4	11
4	15
5	1
7	1
9	4
9	6
9	7
11	2

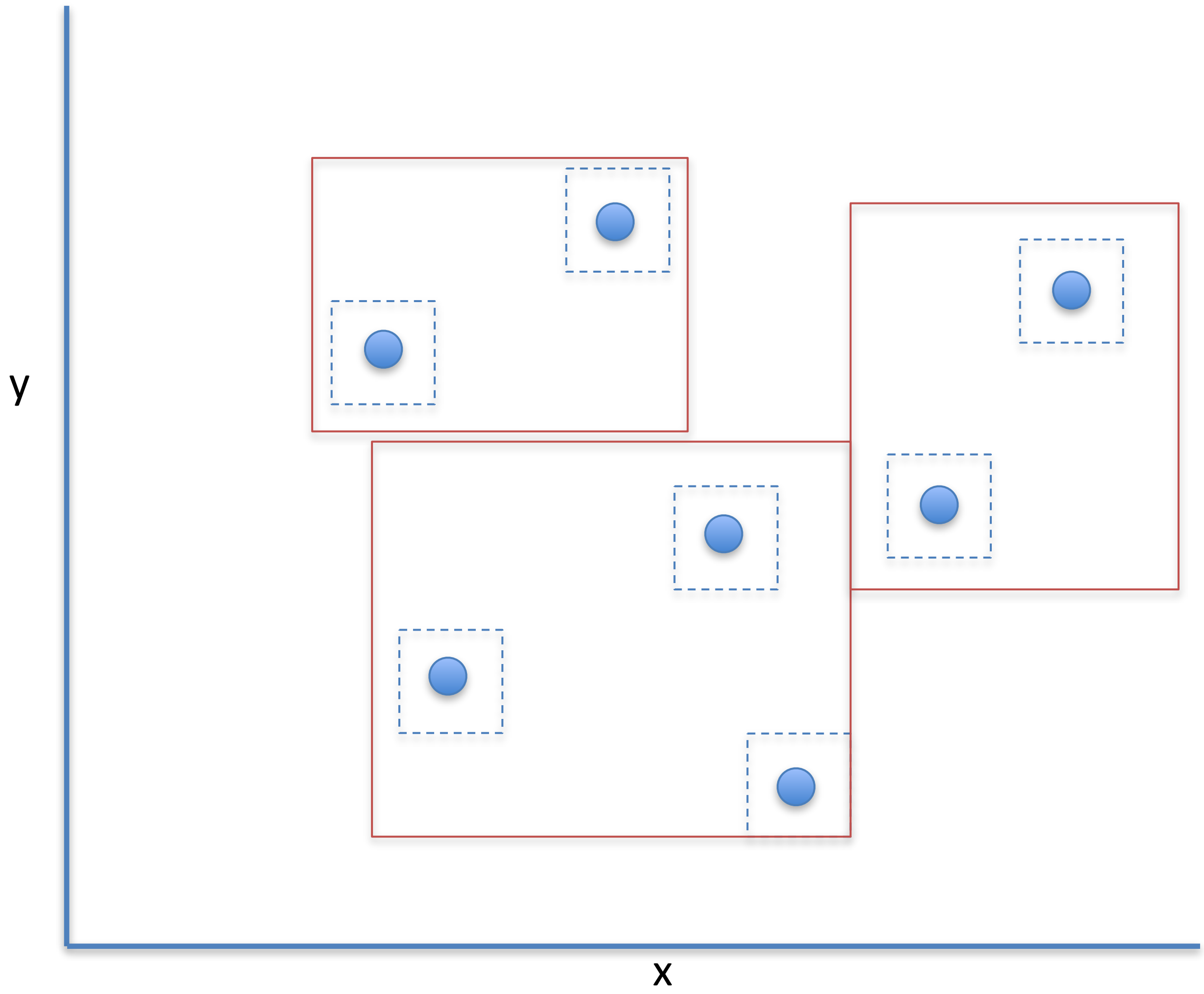
Supports efficient range lookups on X
Allows further filtering on Y, but may be inefficient

Doesn't support lookups on Y

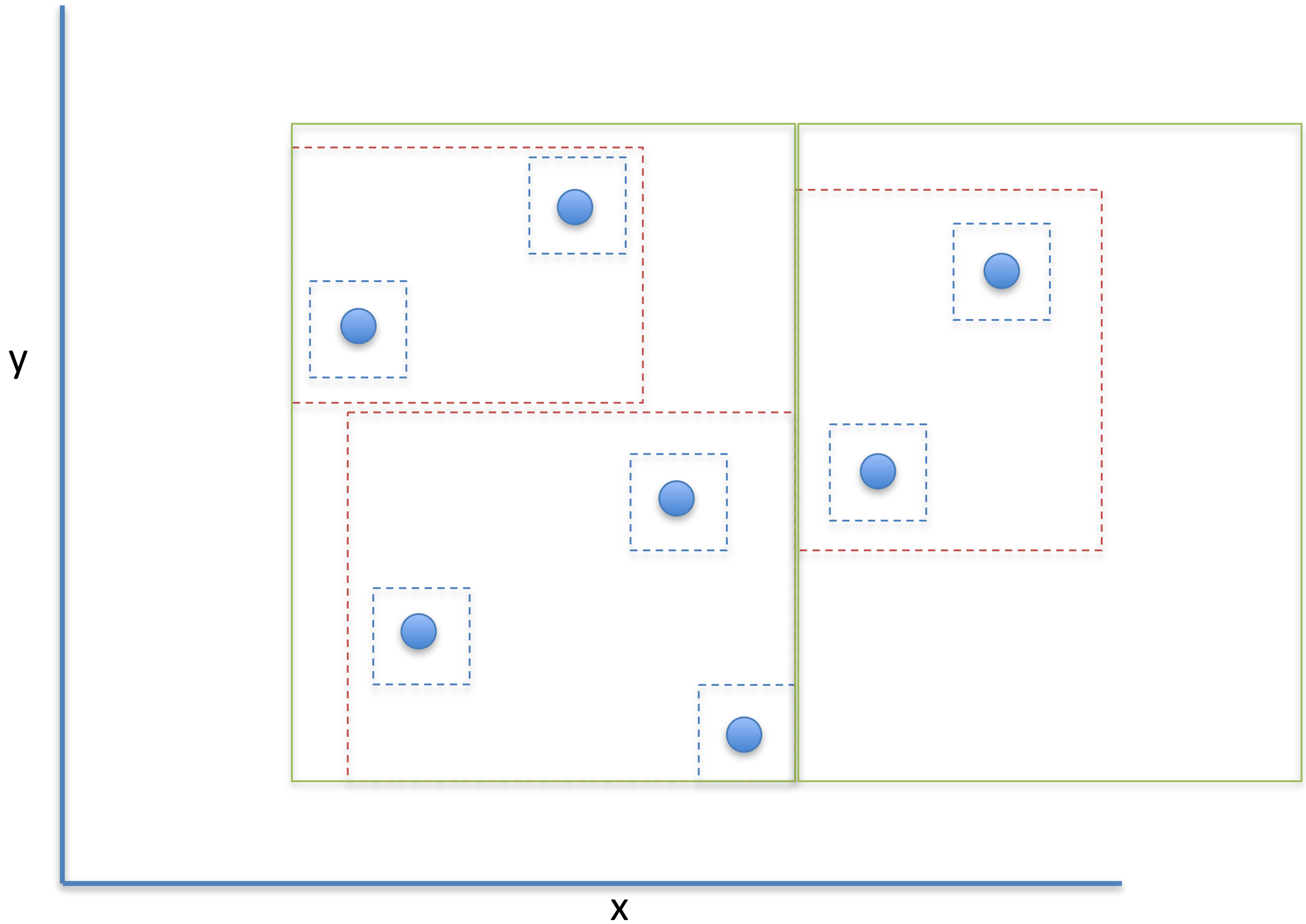
R-Trees / Spatial Indexes

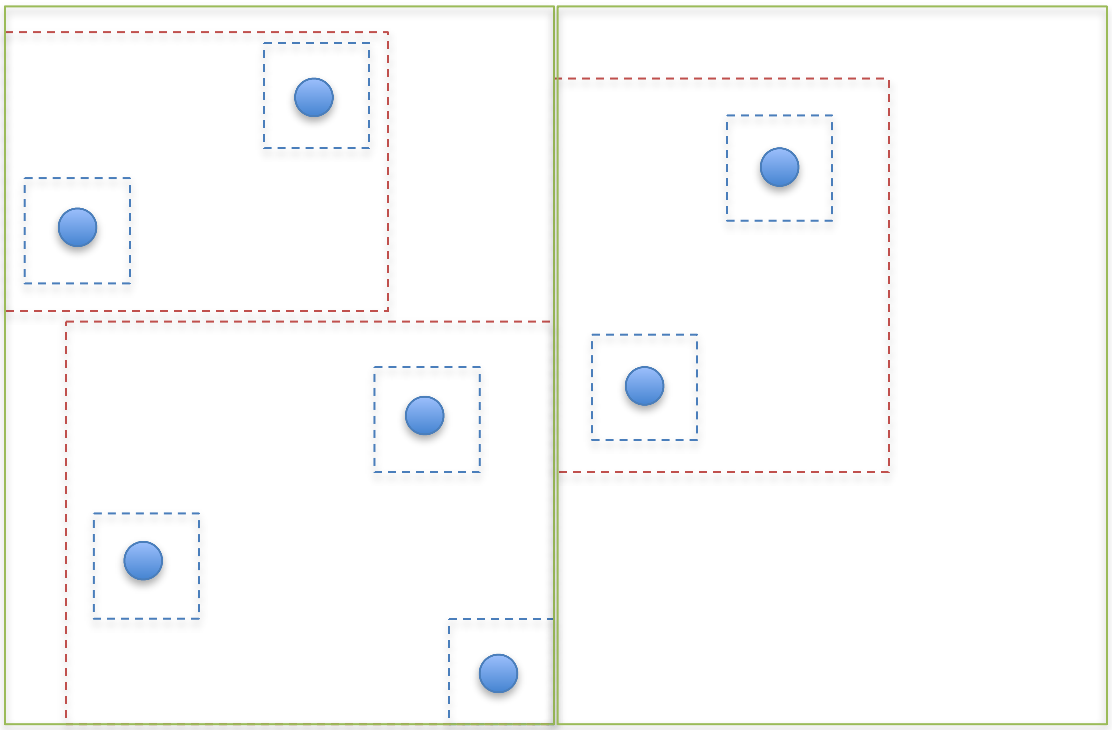


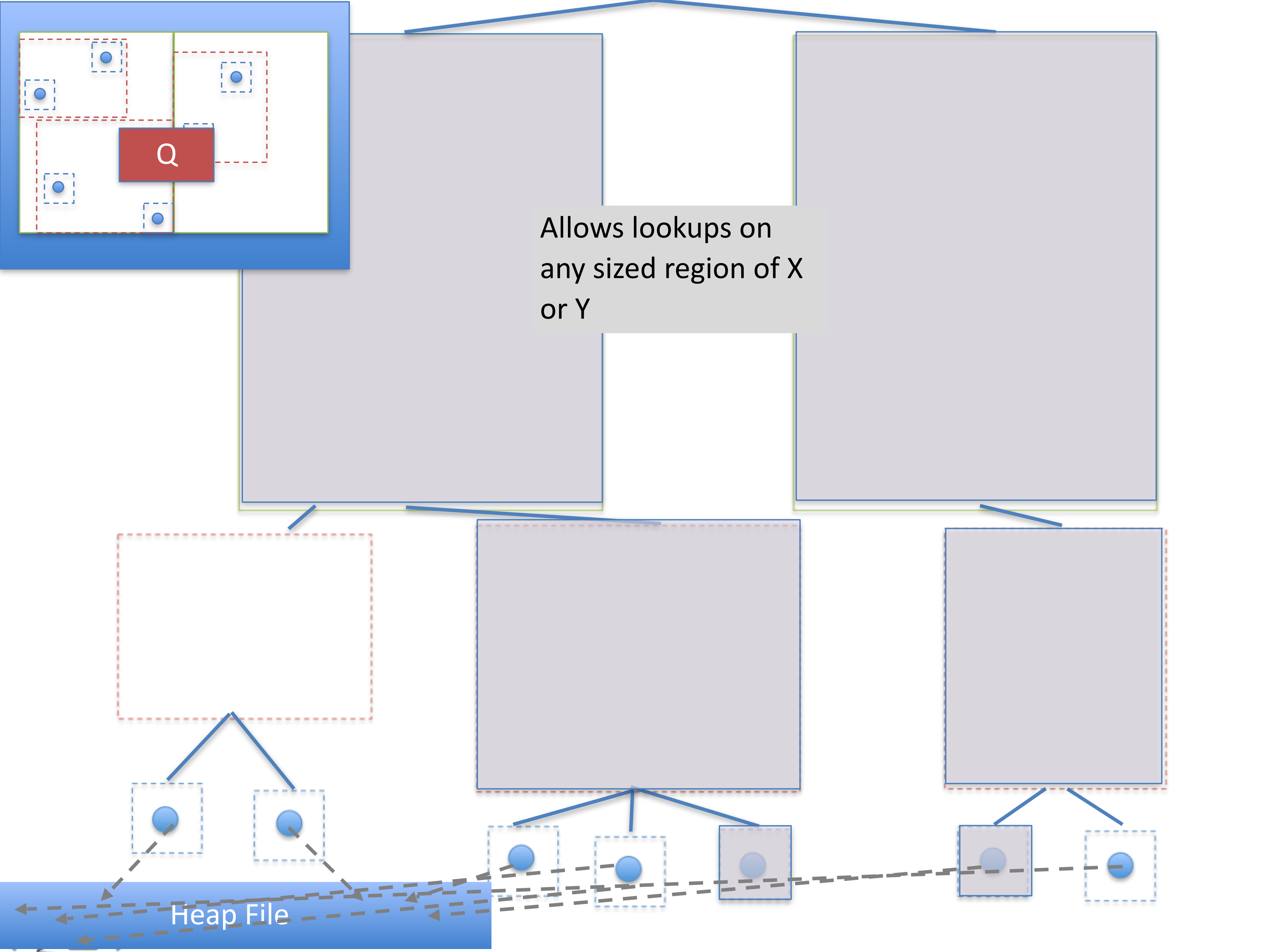
R-Trees / Spatial Indexes



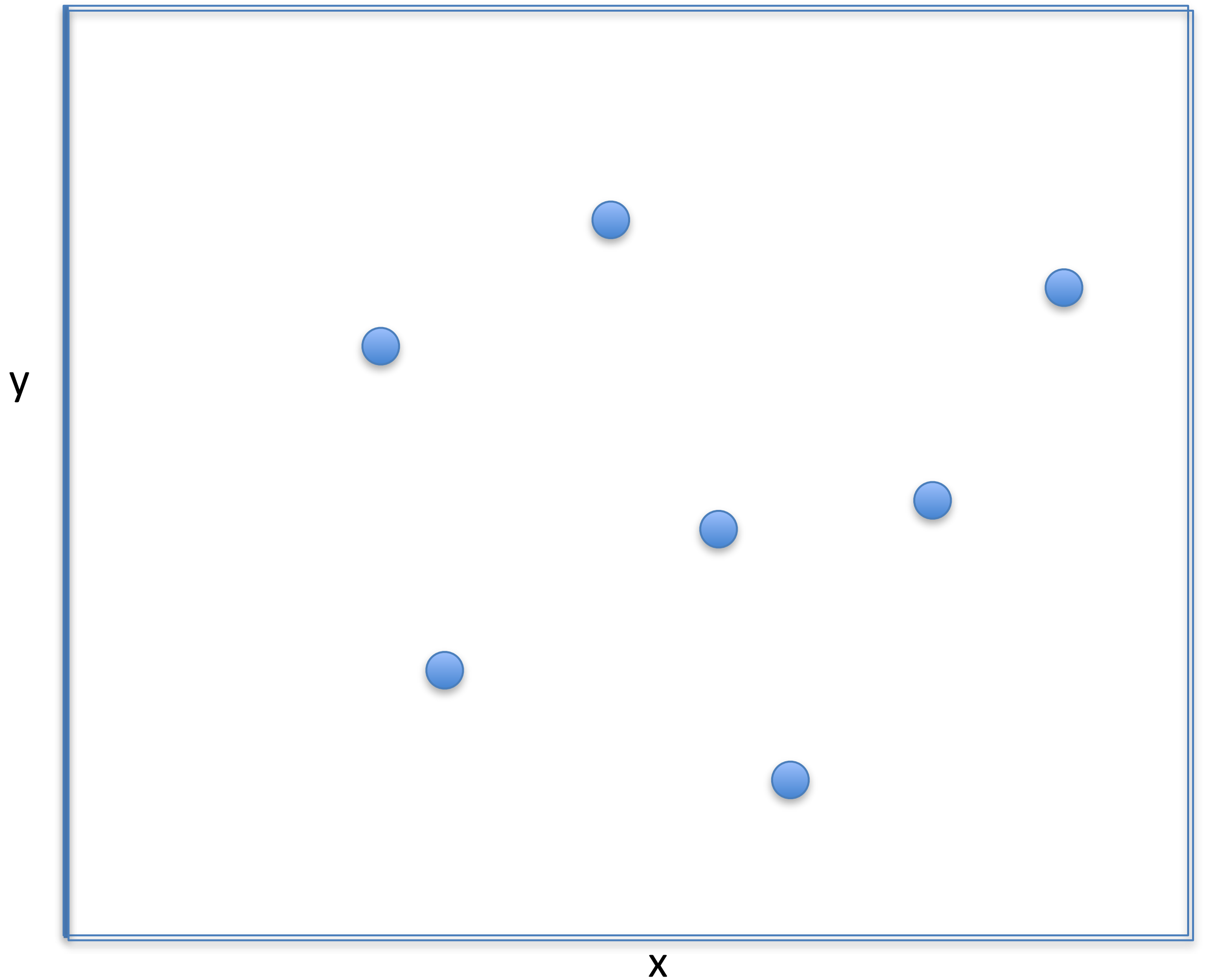
R-Trees / Spatial Indexes



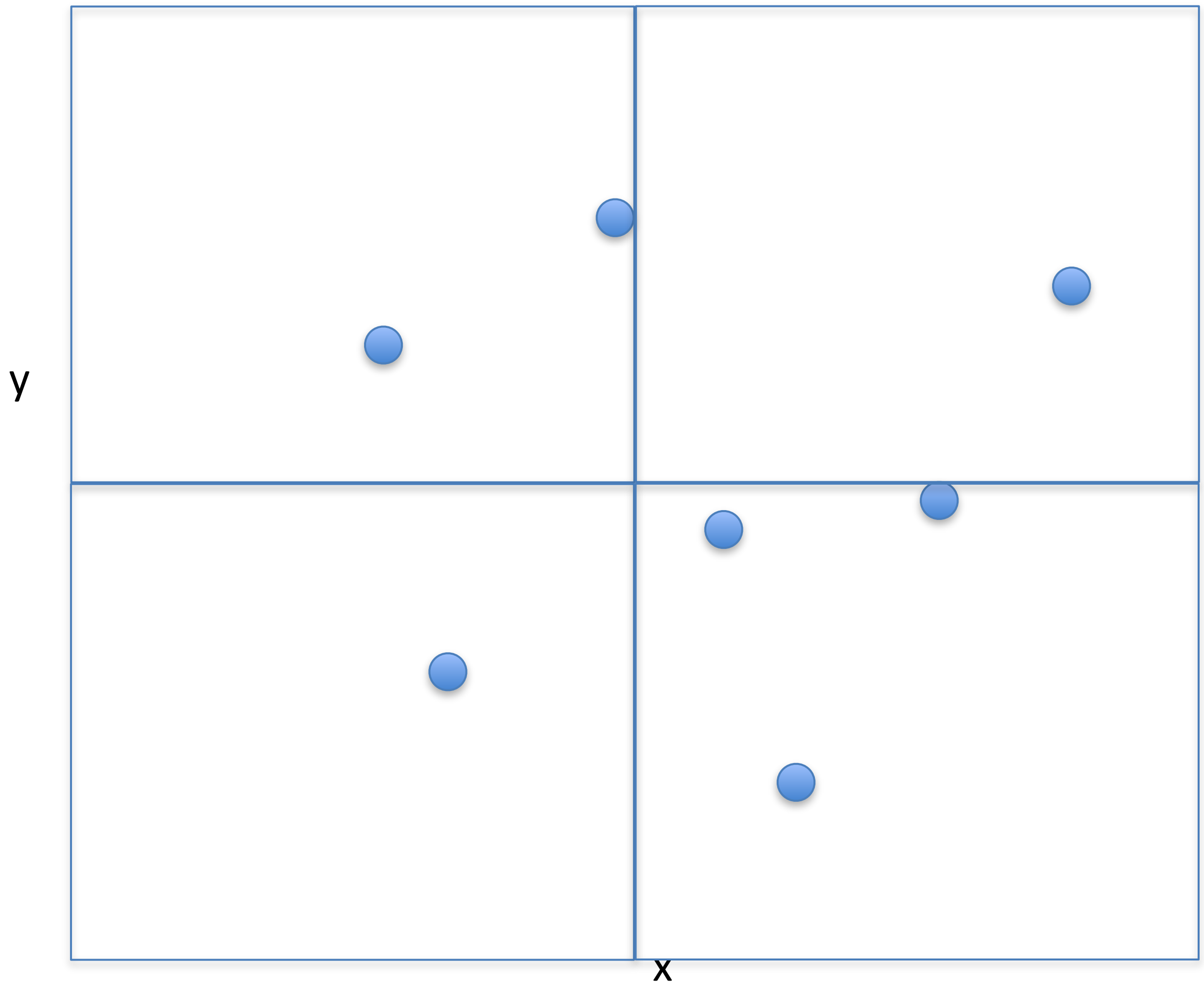



Q
Allows lookups on any sized region of X or Y
Heap File

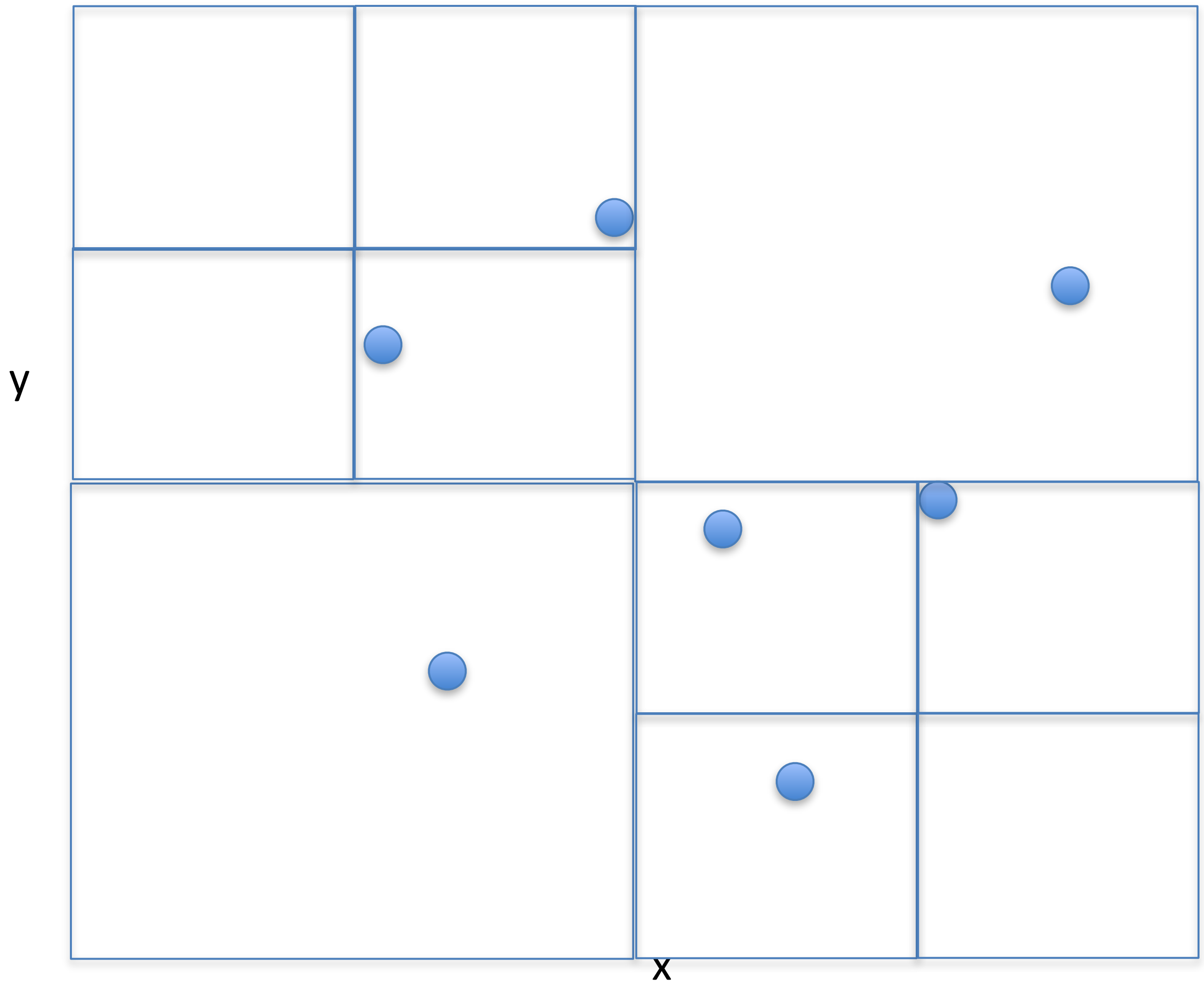
Quad-Tree



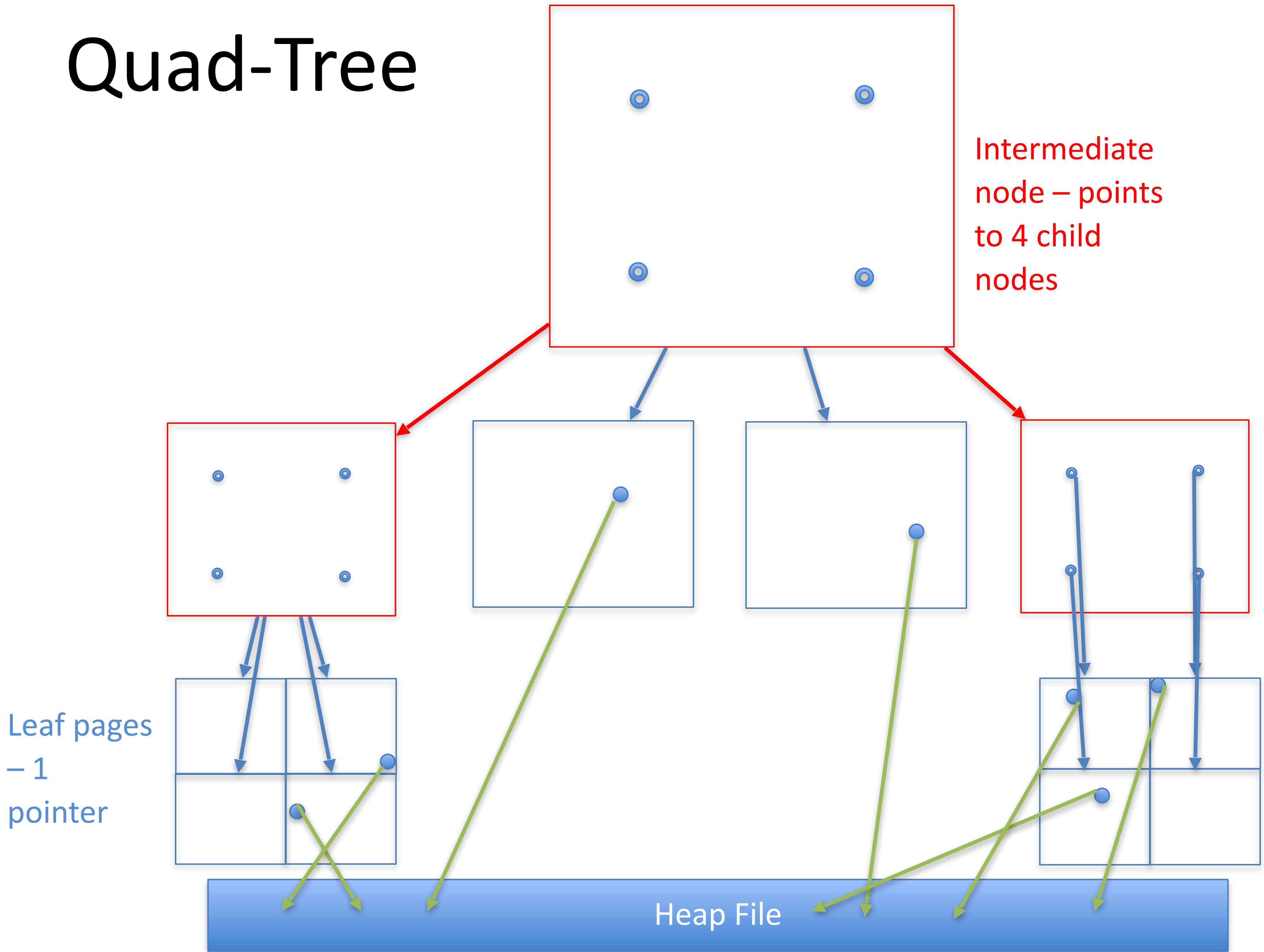
Quad-Tree



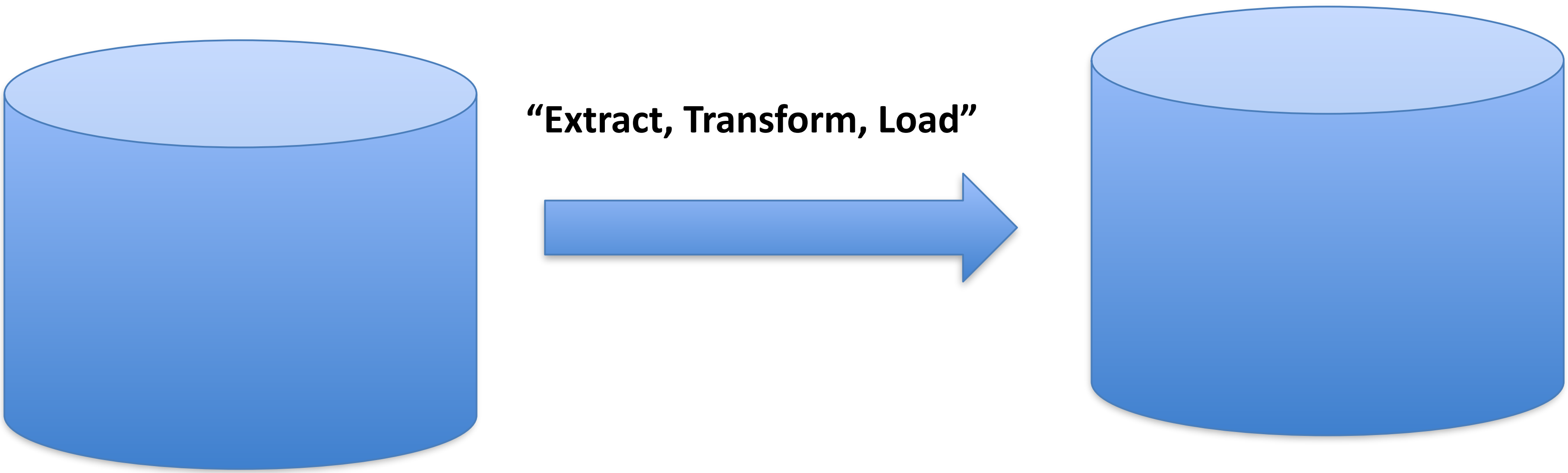
Quad-Tree



Quad-Tree


Intermediate node – points to 4 child nodes
Leaf pages – 1 pointer
Heap File

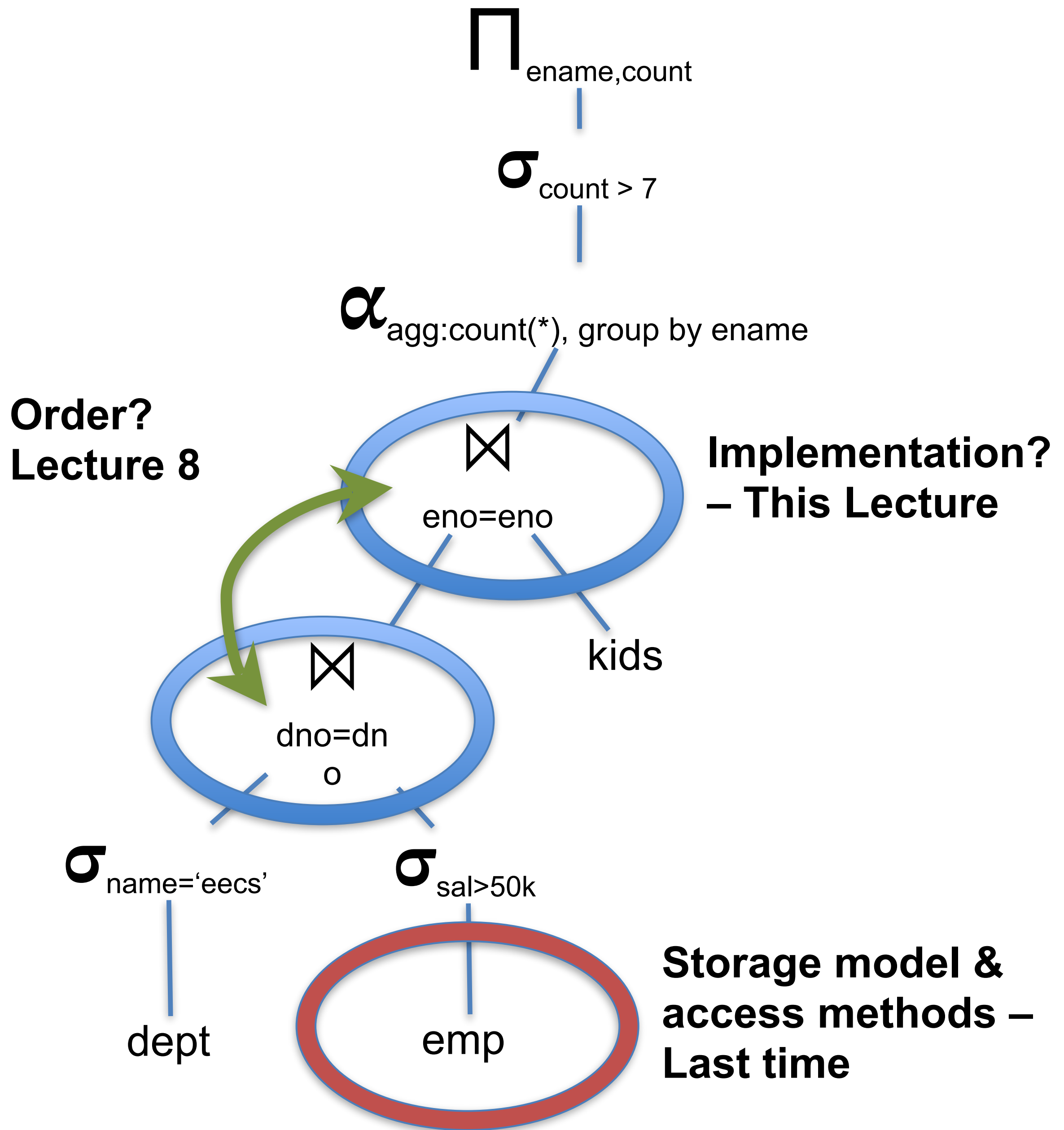
Typical Database Setup



Transactional database
Lots of writes/updates
Reads of individual records

Analytics / Reporting Database
"Warehouse"
Lots of reads of many records
Bulk updates
Typical query touches a few columns

Plan questions



Join Algorithms

- Nested loops (NL)
- Blocked nested loops
- Index nested loops (INL)
- When tables fit in memory
  - Hash (only 1 needs to fit)
  - Sort merge (both must fit)
- When tables don’t fit into memory
  - Blocked hash join
  - External sort merge
  - Simple hash
  - Grace hash

Notation

Evaluating Join(S,R,predicate)

Assume R is always the smaller table

{S} – number of records in S

|S| – number of pages of S

Memory of size M pages

Nested Loops

```
for s in S:
    for r in R
        if pred(s,r):
            output s join r
```

Inner vs outer matters, if only one relation fits in memory

{S} * {R} comparisons in either case

https://clicker.mit.edu/6.5830/

	CPU Complexity	I/O Complexity
(A)	{R} x S log {S}	|S| + |R|
(B)	{R} x {S}	|S| + |S| x |R|
(C)	{R} x {S}	|S| + {S} x |R|
(D)	|S| x |R|	|S| + |R|
(E)	{R} x {S}	|S| + |R|
(F)	{R} x |S|	|S| + |S| x |R|

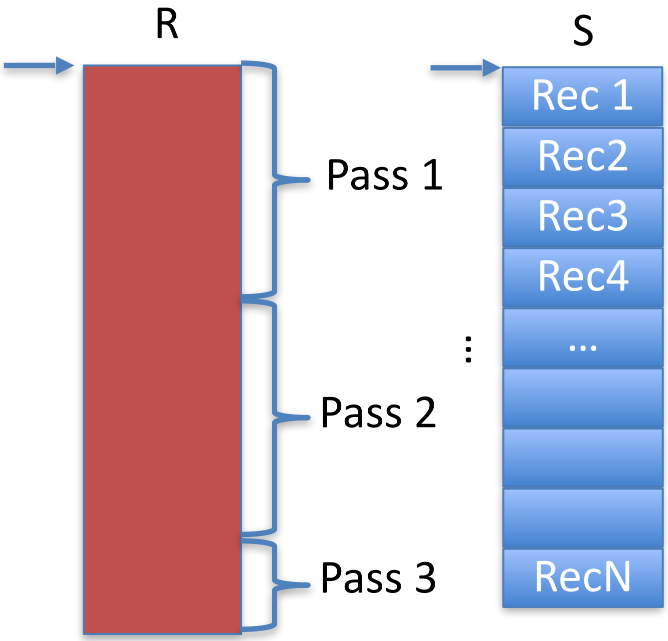
Select all possible correct solutions

Basic Join Summary

	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't

Block Nested Loops

```
B = block size (< M)
while (not at end of R):
    R' = read B records from R
    for s in S:
        for r in R':
            if pred(s,r):
                output s join r
```



Inner vs outer matters; {S} * {R} comparisons, but {R}/B passes over S

https://clicker.mit.edu/6.5830/

	CPU Complexity	I/O Complexity
(A)	{R/M} x {S}	
(B)	{R} x {S/M}	
(C)	{R} x {S}	
(D)	{R} x {S}	

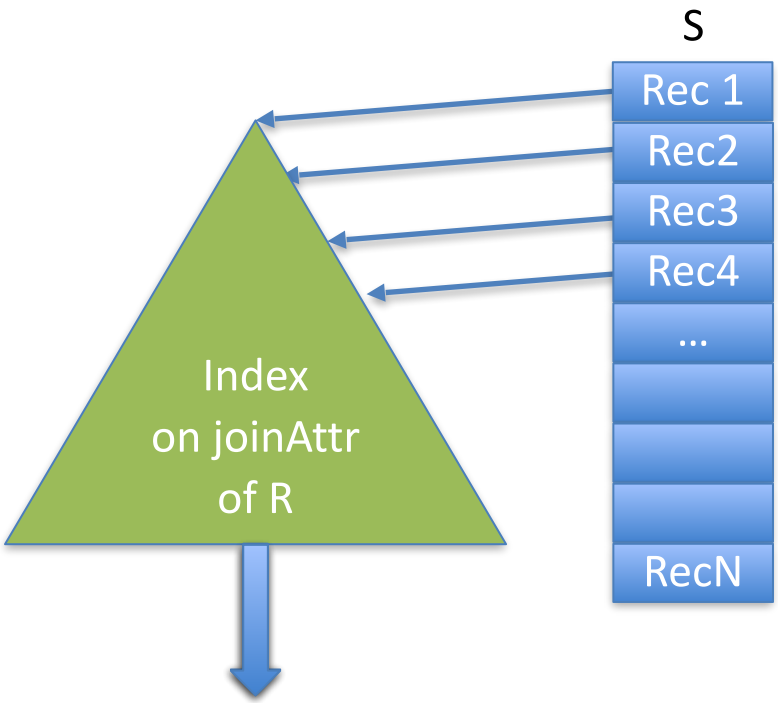
Basic Join Summary

	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	{R} x {S}		Better to partition R (fewer passes)

Index Nested Loops

- Assume Index I on Join Attribute of R

```
for s in S:
    for r in lookup s.joinAttr in I:
        output s join r
```



Inner vs outer matters; {S} lookups

Inner is always indexed attribute

Note that index lookups are random, unless S is ordered on join attribute and index is clustered on join attribute

Basic Join Summary

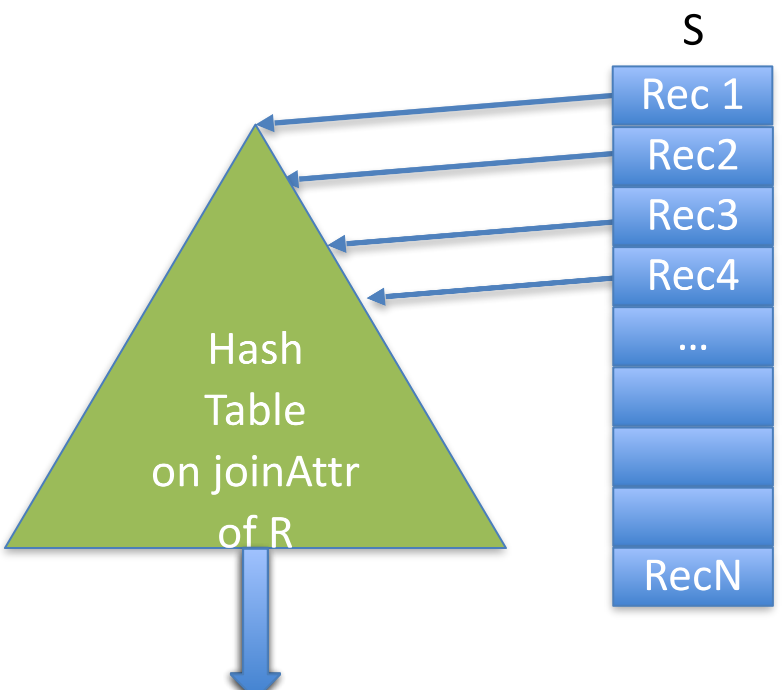
	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	{R} x {S}	+ |R|	Better to partition R (fewer passes)
Index nested loops	{R} x D *D is tree depth, < ~5*	{R} x D *I/O random unless R sorted & index clustered on join attr*	Assuming index on S.

(In Memory) Hash Join

- Essentially the same as index nested loops, with in-memory hash "index" built on the fly
- Build hash table T on join attribute of R

```
T = build hash table on joinAttr of R
for s in S:
    for r in lookup s.joinAttr in T:
        output s join r
```

Inner vs outer matters; {S} lookups, requires memory to hold hash table on R



https://clicker.mit.edu/6.5830/

	CPU Complexity	I/O Complexity
(A)	{R} * {S}	|R| + 2 * |S|
(B)	{R} + {S}	|R| * |S|
(C)	{R} * {S}	|R| + |S|
(D)	{R} + {S}	|R| + |S|

R is the inner table and fits into main memory

Basic Join Summary

	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	{R} x {S}	+ |R|	Better to partition R (fewer passes)
Index nested loops	{R} x D *D is tree depth, < ~5*	{R} x D *I/O random unless R sorted & index clustered on join attr*	Assuming index on S.
Hash join	{R} + {S}	|R| + |S|	R must fit into memory

Blocked Hash

- Similar to block nested loops
- Iteratively:
  - Build hash table on chunk of R so that hash table fits in memory
  - Probe (lookup in) with all of S
  - Repeat with next chunk of R

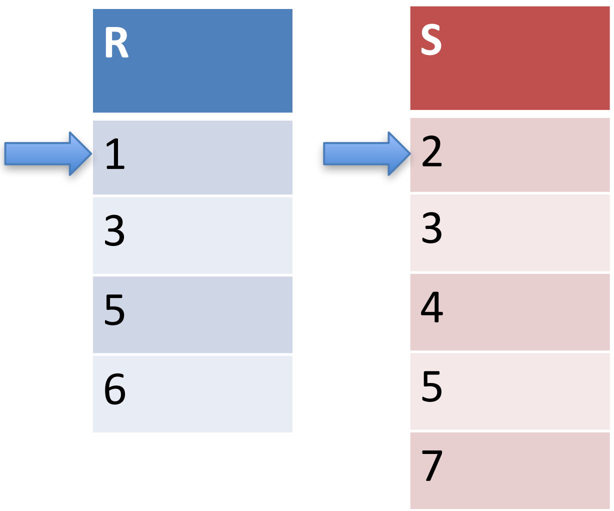
Basic Join Summary

	CPU Complexity	I/O Complexity	Notes
Nested loops	$\{R\} \times \{S\}$	$\lvert S \rvert + \{S\}\lvert R \rvert$ *R doesn't fit in memory* $\lvert S \rvert + \lvert R \rvert$ *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	$\{R\} \times \{S\}$	$+ \lvert R \rvert$	Better to partition R (fewer passes)
Index nested loops	$\{R\} \times D$ *D is tree depth, $< \sim 5$*	$\{R\} \times D$ *I/O random unless R sorted & index clustered on join attr*	Assuming index on S.
Hash join	$\{R\} + \{S\}$	$\lvert R \rvert + \lvert S \rvert$	Both tables must fit in memory
Blocked hash join		$+ \lvert R \rvert$	

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```

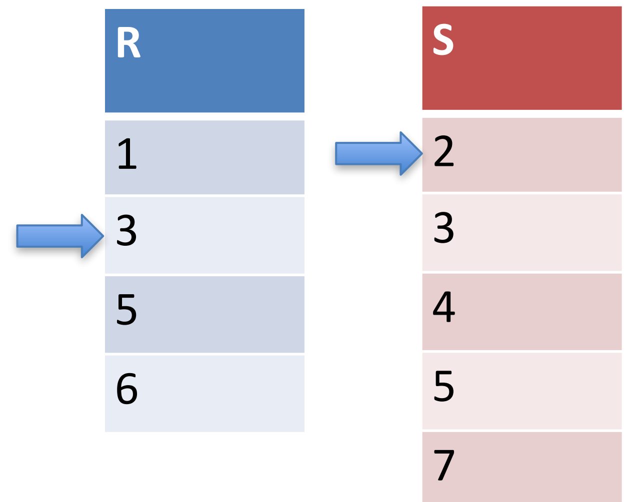


Output:

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```

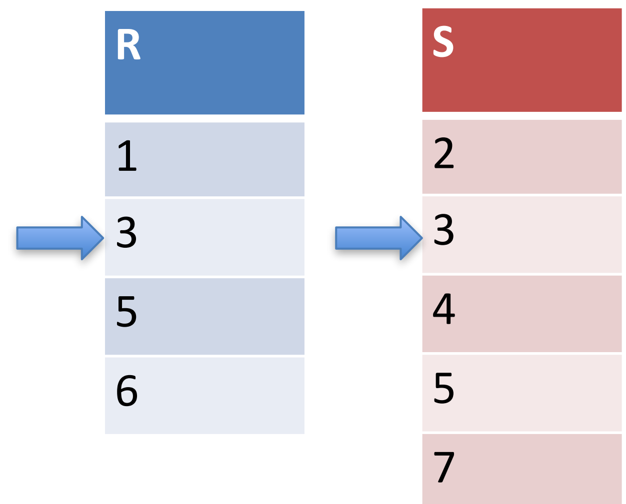


Output:

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```

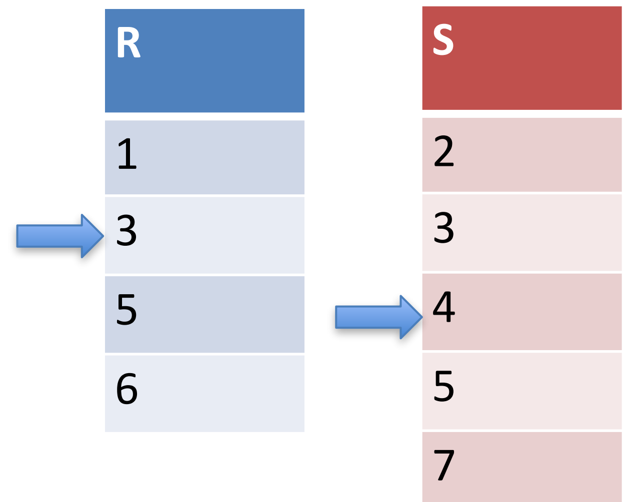


Output: 3

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```



Output: 3

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```

R
1
3
5
6

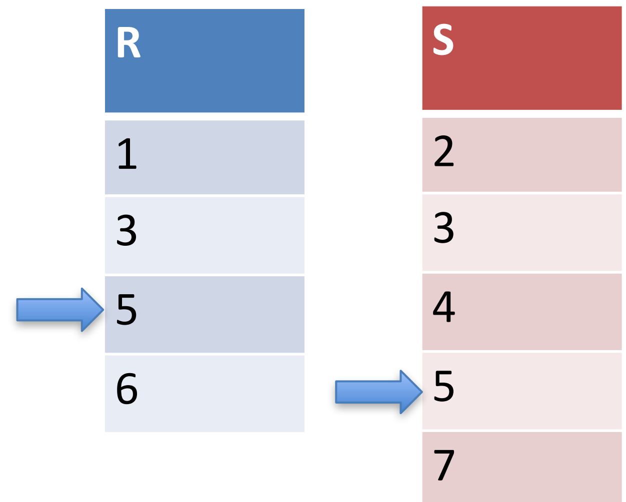
S
2
3
4
5
7

Output: 3

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```



Output: 3, 5

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```

R
1
3
5
6

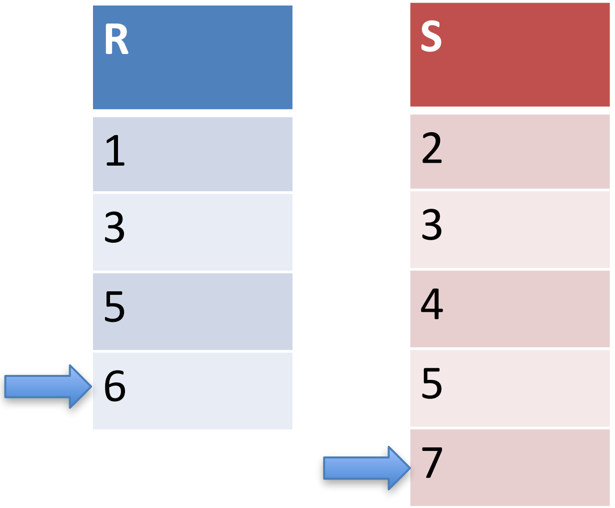
S
2
3
4
5
7

Output: 3, 5

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```



Output: 3, 5

Sort Merge Join

- Sort both S and R (or use index on each to traverse in order)
- Merge (no shared duplicates)

```
while (i < {R} and j < {S}):
    if (R[i].joinAttr == S[j].joinAttr):
        output R[i] join S[j]
    if (R[i].joinAttr < S[j].joinAttr):
        i = i + 1
    else:
        j = j + 1
```

R
1
3
5
6

S
2
3
4
5
7

Output: 3, 5

Note that output is sorted!

Handling Duplicates

- What is desired output?

4 copies!

(5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

(5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

  4 copies!

  (5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

(5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

(5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

Output: 5

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

  4 copies!

  (5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

Output: 5, 5

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

(5,5),(5,5),(5,5),(5,5)

R
1
5
5
6

S
2
3
5
5
7

Output: 5, 5

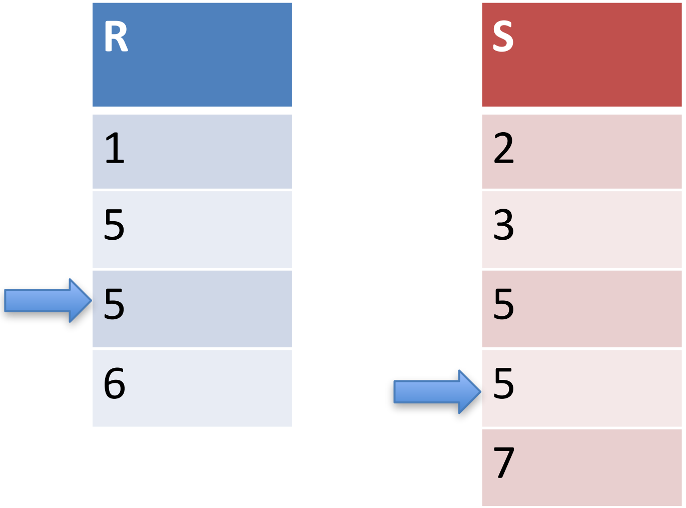
- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

  4 copies!

  (5,5),(5,5),(5,5),(5,5)



Output: 5, 5, 5

- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

  4 copies!

  (5,5),(5,5),(5,5),(5,5)

R	S
1	2
5	3
5	5
6	5
	7

Output: 5, 5, 5, 5

- Solution: count run lengths in S and R, emit cross product of repeated runs

Psuedocode for Duplicates

```
while (i < {R} and j < {S}):
  if R[i].joinAttr == S[j].joinAttr:
    rLen = getRunLen(R,i)
    sLen = getRunLen(S,j)
    emitRun(R,S,i,j,rLen,sLen)
    i = i + rLen
    j = j + sLen
  elif R[i].joinAttr < S[j].joinAttr:
    i = i + 1
  else:
    j = j + 1
```

```
def getRunLen(v,i):
  runLen = 1
  while (i < len(v)-1):
    i = i + 1
    if v[i] == v[i-1]:
      runLen = runLen + 1
    else:
      break
  return runLen
```

```
def emitRun(R,S,r,s,rLen,sLen):
  for i in range(r,r+rLen):
    for j in range(s,s+sLen):
      output R[i] join S[j]
```

Basic Join Summary

	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	{R} x {S}	+ |R|	Better to partition R (fewer passes)
Index nested loops	{R} x D *D is tree depth, < ~5*	{R} x D *I/O random unless R sorted & index clustered on join attr*	Assuming index on S.
Hash join	{R} + {S}	|R| + |S|	Both tables must fit in memory
Blocked hash join		+ |R|	
Sort merge join	{R}log{R} + {S}log{S} + {S} + {R}	|R| + |S|	Assumes both tables fit in memory; If already sorted, can avoid logn step

Study Break

- When would you prefer sort-merge over hash join?
- When would you prefer index-nested-loops join over hash join?

Join Processing in Database Systems with Large Main Memories

LEONARD D. SHAPIRO
North Dakota State University

“External” Sort Merge Join

Equi-join of two tables S & R

|S| = Pages in S; {S} = Tuples in S

|S| ≥ |R|

M pages of memory; M > sqrt(|S|)

Algorithm:

- Partition S and R into memory sized sorted runs, write out to disk
- Merge all runs simultaneously

Total I/O cost: Read |R| and |S| twice, write once

3(|R| + |S|) I/Os

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

If each run is M pages and M > sqrt(|S|), then there are at most

$$|S|/sqrt(|S|) = sqrt(|S|)$$

runs of S

So if |R| = |S|, we actually need M to be 2 x sqrt(|S|)

[handwavy argument in paper for why it's only sqrt(|S|)]

R1

S1

OUTPUT

1					
3					
4	14	11	7	12	15

Need enough memory to keep 1 page of each run in memory at a time

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2 ←	8 ←	4 ←
3 ←	9	7 ←	3	9	6
4	14	11	7	12	15

OUTPUT

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ⇐	1	2	8 ⇐	4 ⇐
3 ⇐	9	7 ⇐	3 ⇐	9	6
4	14	11	7	12	15

OUTPUT

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT
(3,3)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4 ←
3	9	7 ←	3 ←	9	6
4 ←	14	11	7	12	15

OUTPUT
(3,3)
(4,4)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ⇐	1	2	8 ⇐	4 ⇐
3	9	7 ⇐	3	9	6
4 ⇐	14	11	7 ⇐	12	15

OUTPUT
(3,3)
(4,4)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4
3	9	7 ←	3	9	6 ←
4 ←	14	11	7 ←	12	15

OUTPUT
(3,3)
(4,4)
(6,6)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4   R2 = 6,9,14   R3 = 1,7,11

S1 = 2,3,7   S2 = 8,9,12   S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ←	1	2	8 ←	4
3	9	7 ←	3	9	6 ←
4 ←	14	11	7 ←	12	15

…

OUTPUT
(3,3)
(4,4)
(6,6)
(7,7)

Output in sorted order!

Simple "External" Hash

Idea: Avoid repeated passes over S in blocked hash

Algorithm:

Given hash function H(x) → [0,…,P-1] *(e.g., x mod P)*

where P is number of partitions

for i in [0,…,P-1]:

for each r in R:

if H(r)=i, add r to in memory hash

otherwise, write r back to disk in R'

for each s in S:

if H(s)=i, lookup s in hash, output matches

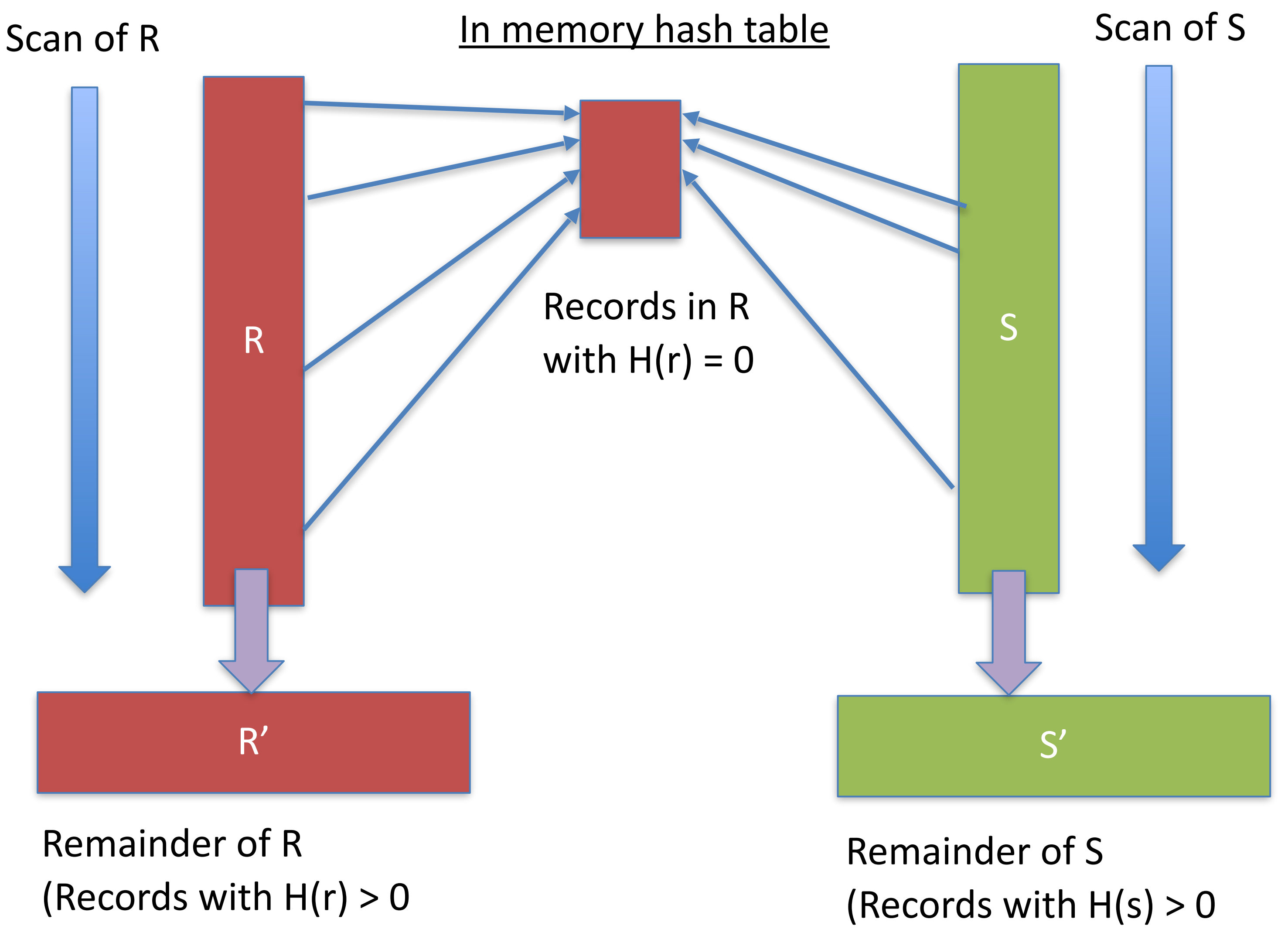
otherwise, write s back to disk in S'

replace R with R', S with S'

Pass 0

Illustration

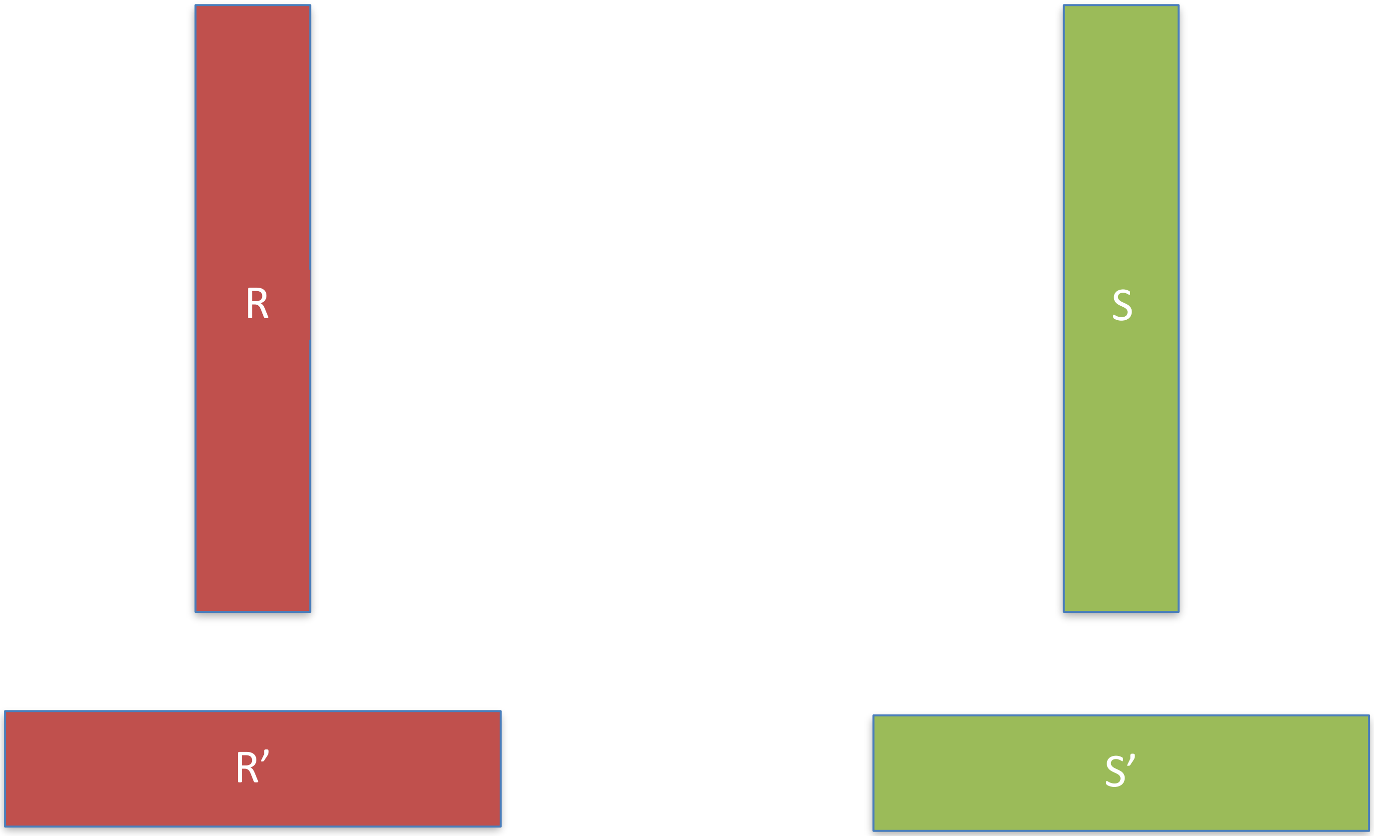
Hash function in 0…P



Pass 0

Illustration

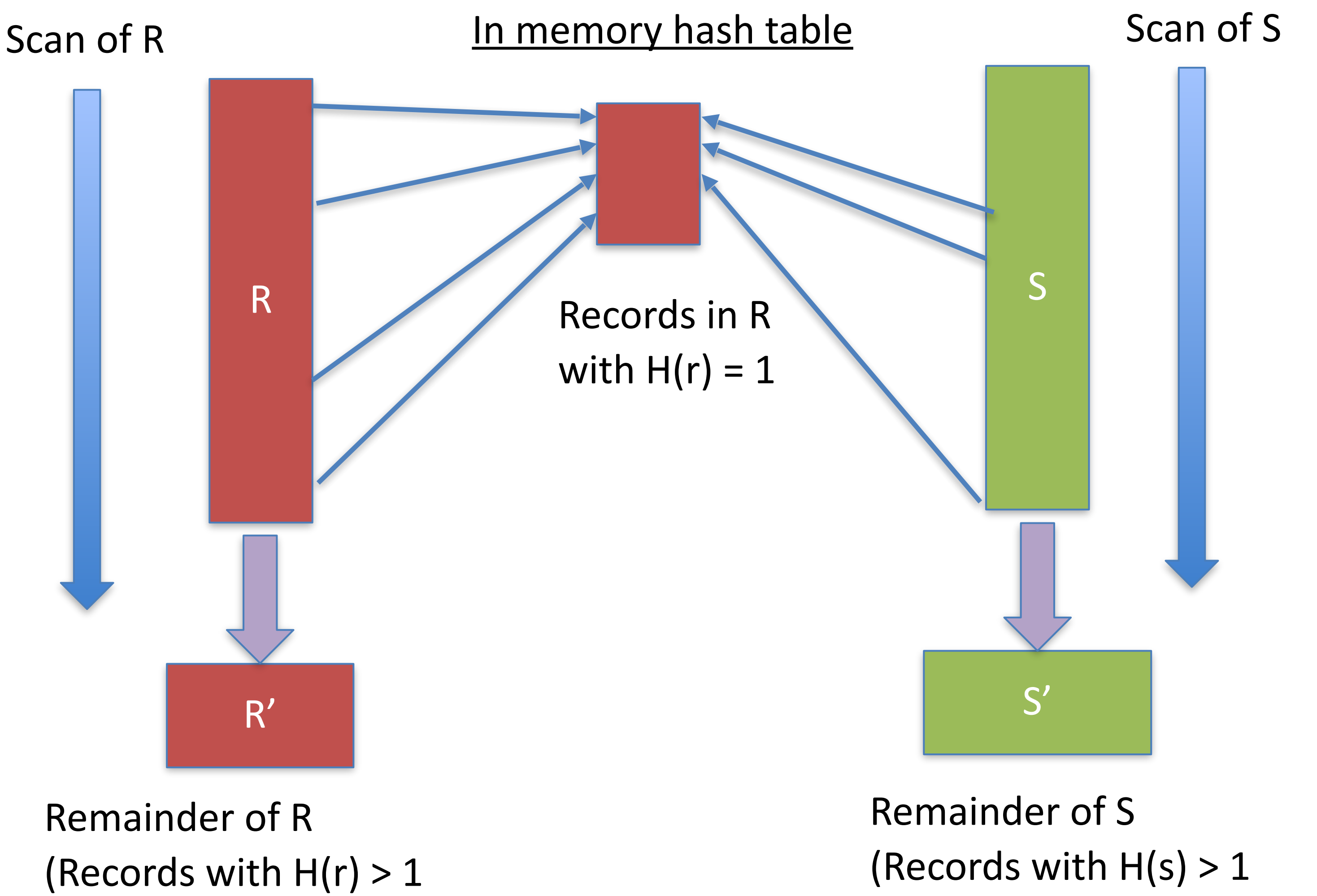
Hash function in 0…P



Pass 1

Illustration

Hash function in 0…P



Repeat for P passes

https://clicker.mit.edu/6.5830/

	I/O Complexity
(A)	
(B)	
(C)	
(D)	

N = number of partitions

Simple Hash I/O Analysis

Suppose P=2, and hash uniformly maps tuples to partitions

Read |R| + |S|

Write 1/2 (|R| + |S|)

Read 1/2 (|R| + |S|)

} 2 (|R| + |S|)

P=3

Read |R| + |S|

Write 2/3 (|R| + |S|)

Read 2/3 (|R| + |S|)

Write 1/3 (|R| + |S|)

Read 1/3 (|R| + |S|)

} 3 (|R| + |S|)

P=4

|R| + |S| + 2 * (3/4 (|R| + |S|)) + 2 * (2/4 (|R| + |S|)) + 2 * (1/4 (|R| + |S|))

= 4 (|R| + |S|)

➔ P = n ; n * (|R| + |S|) I/Os

Grace Hash

Can we avoid rewriting some records many times?

Algorithm:

Partition:

Suppose we have P partitions, and H(x) → [0…P-1]

Choose P = |S| / M ➔ P ≤ sqrt(|S|) *//may need to leave a little slop for imperfect hashing*

Allocate P 1-page output buffers, and P output files for R

For each r in R:

Write r into buffer H(r)

If buffer full, append to file H(r)

Allocate P output files for S

For each s in S:

Write s into buffer H(s)

if buffer full, append to file H(s)

Need one page of RAM for each of P partitions

Since
M > sqrt(|S|) and
P ≤ sqrt(|S|), all is well

Join:

For i in [0,…,P-1]

Read file i of R, build hash table (memory *should* hold this)

Scan file i of S, probing into hash table and outputting matches

Total I/O cost: Read |R| and |S| once, write once, read back once more

3(|R| + |S|) I/Os

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2

P output buffers

F0	F1	F2

P output files

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
		5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
	4	5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5
6		

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5
6		

F0	F1	F2

Need to flush R0 to F0!

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
	4	5

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
		14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
	1	14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
	1	14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9		5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	

F0	F1	F2
3	4	5
6	1	14

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	11

F0	F1	F2
3	4	5
6	1	14

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2

F0	F1	F2
3	4	5
6	1	14
9	7	11

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
→ 15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9
6,6
7,7
4,4

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6
7,7
4,4

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Hybrid

- Acts like simple for small tables, grace for large tables
- Suppose we have M = $\sqrt{|R|}$ + E
  - E is additional memory beyond the minimum
- Make the first partition size E, and join as in simple
- For remaining partitions write out as in grace
- Repeat with S, joining first partition on the fly, and writing out remaining partitions as in grace
- Join remaining partitions as in grace

External Join Summary

Notation: P partitions / passes over data; assuming hash is O(1)

Sort-Merge	Simple Hash	Grace Hash
I/O: 3 (|R| + |S|) CPU: O(P x {S}/P log {S}/P)	I/O: P (|R| + |S|) CPU: O({R} + {S})	I/O: 3 (|R| + |S|) CPU: O({R} + {S})

Grace hash is generally a safe bet, unless memory is close to size of tables, in which case simple can be preferable

Extra cost of sorting makes sort merge unattractive unless there is a way to access tables in sorted order (e.g., a clustered index), or a need to output data in sorted order (e.g., for a subsequent ORDER BY)

Many fancier versions exist, e.g., using modern sorting techniques (radix or counting sort), parallel cores, etc