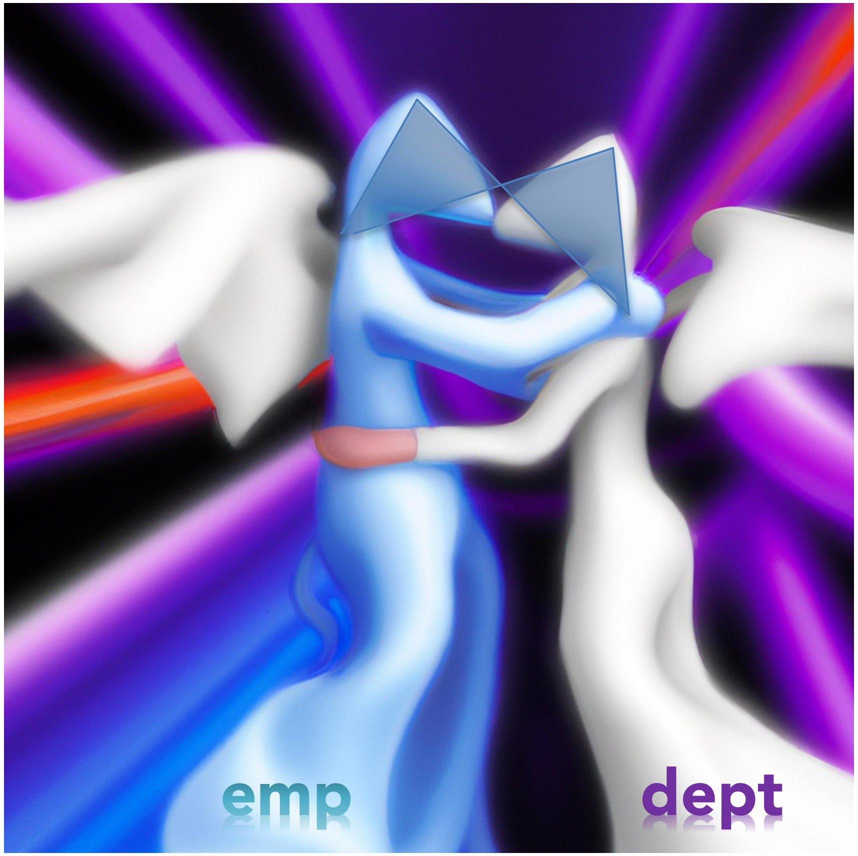
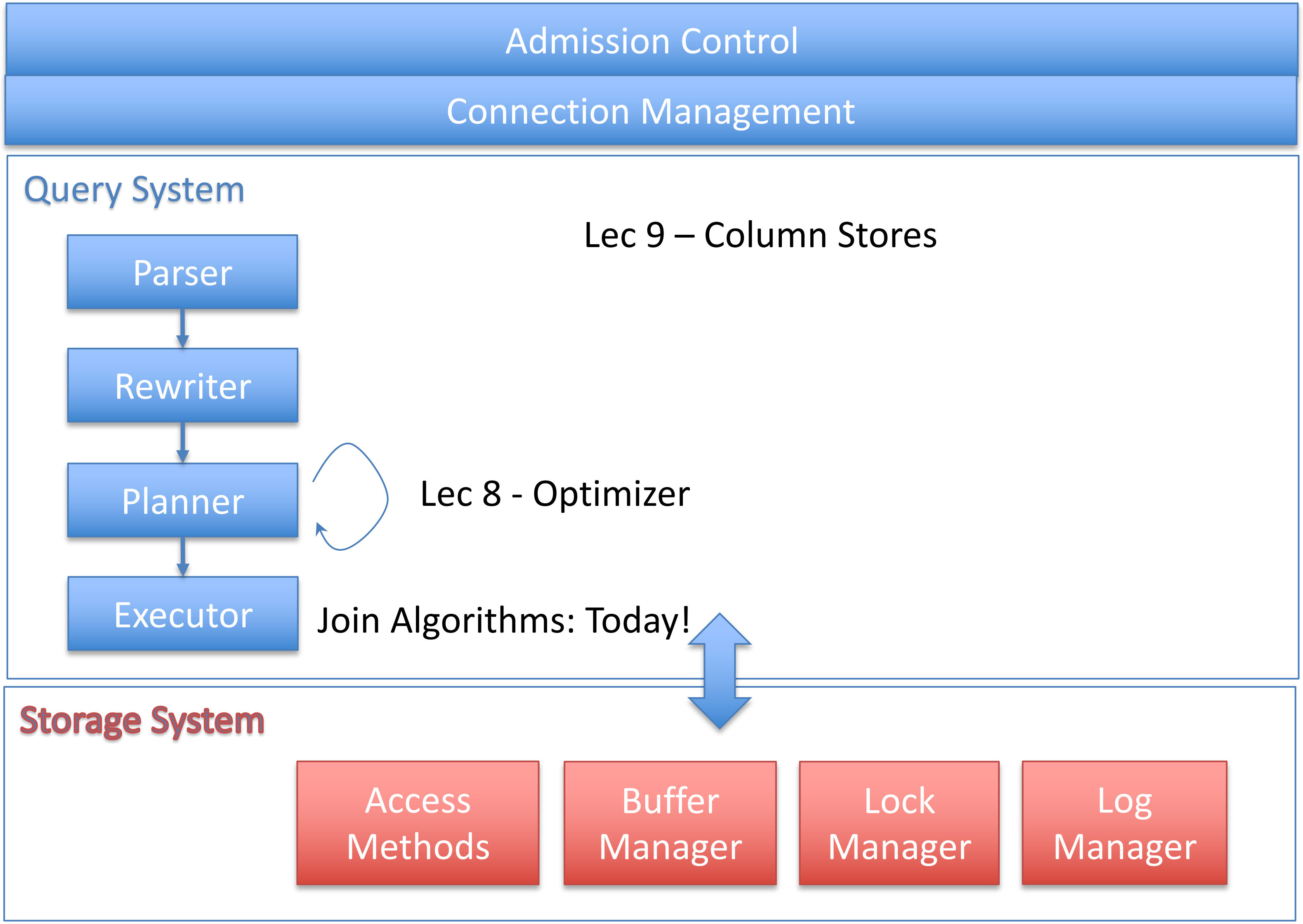
6.5830 Lecture 7



Join Algorithms

September 27, 2023

Plan for Next Few Lectures



Admission Control

Connection Management

Query System

Parser

Rewriter

Planner

Executor

Lec 9 – Column Stores

Lec 8 - Optimizer

Join Algorithms: Today!

Storage System

Access Methods

Buffer Manager

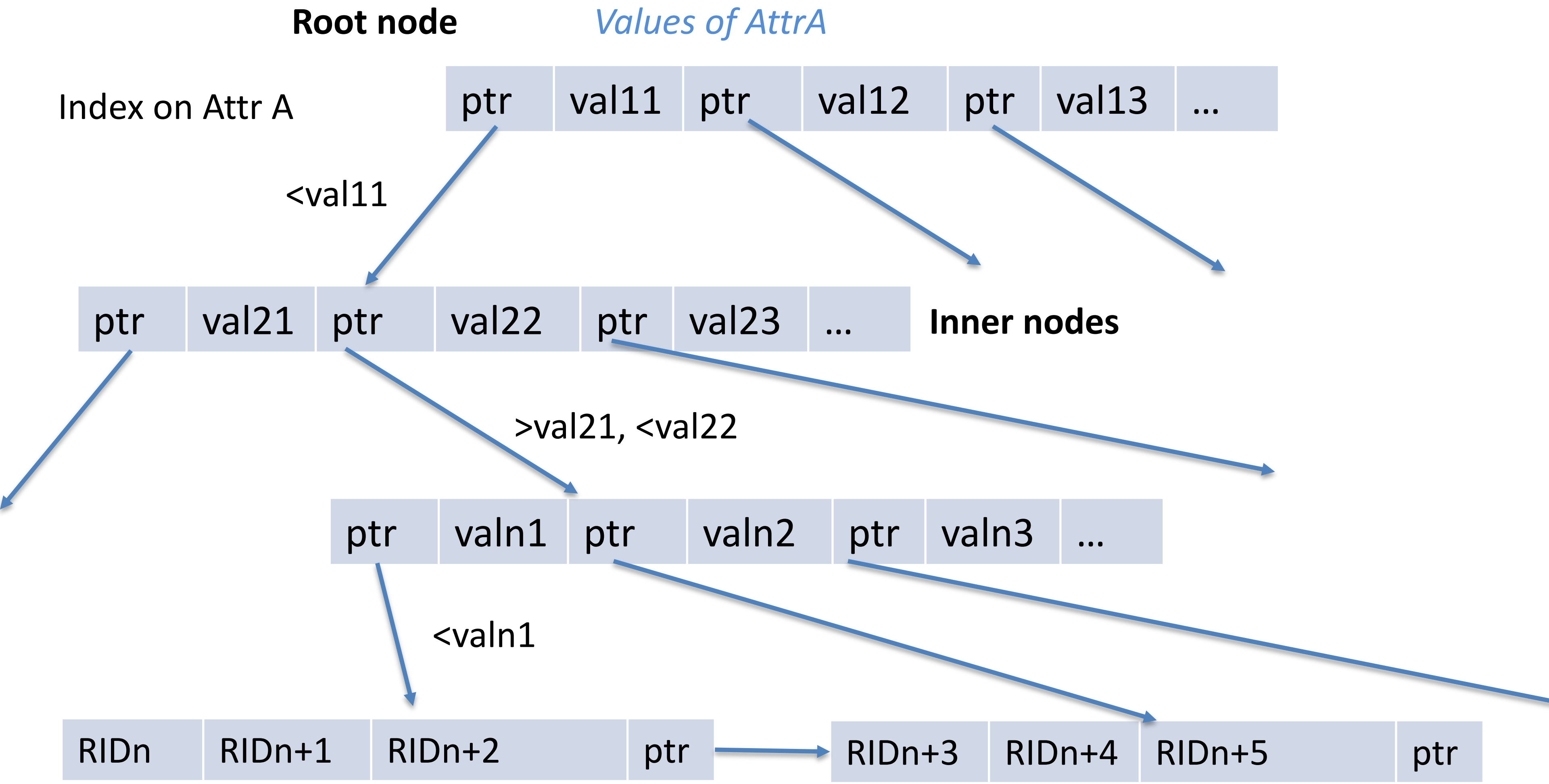
Lock Manager

Log Manager

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 ← next time
- Many alternatives: e.g., R-trees ← next time
- Each has different performance tradeoffs

B+Trees



Root node

Values of AttrA

Index on Attr A

ptr | val11 | ptr | val12 | ptr | val13 | ...

<val11

ptr | val21 | ptr | val22 | ptr | val23 | ...

Inner nodes

>val21, <val22

ptr | valn1 | ptr | valn2 | ptr | valn3 | ...

<valn1

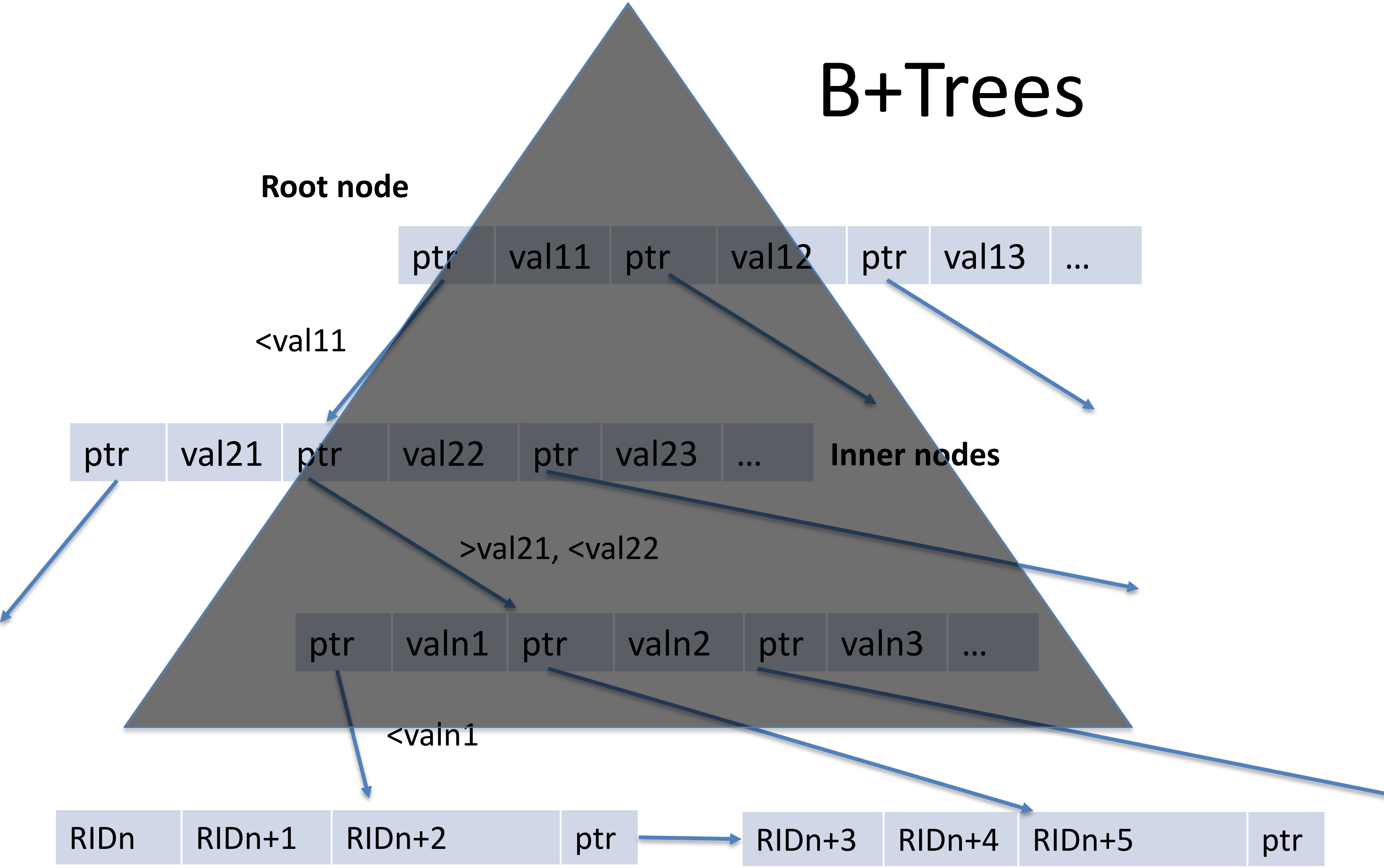
RIDn | RIDn+1 | RIDn+2 | ptr

RIDn+3 | RIDn+4 | RIDn+5 | ptr

RID: Record ID → a reference (pointer) to a record in heap file

Leaf nodes; records in Attr A order, w/ link pointers

B+Trees



Root node

ptr | val11 | ptr | val12 | ptr | val13 | …

<val11

ptr | val21 | ptr | val22 | ptr | val23 | …

Inner nodes

>val21, <val22

ptr | valn1 | ptr | valn2 | ptr | valn3 | …

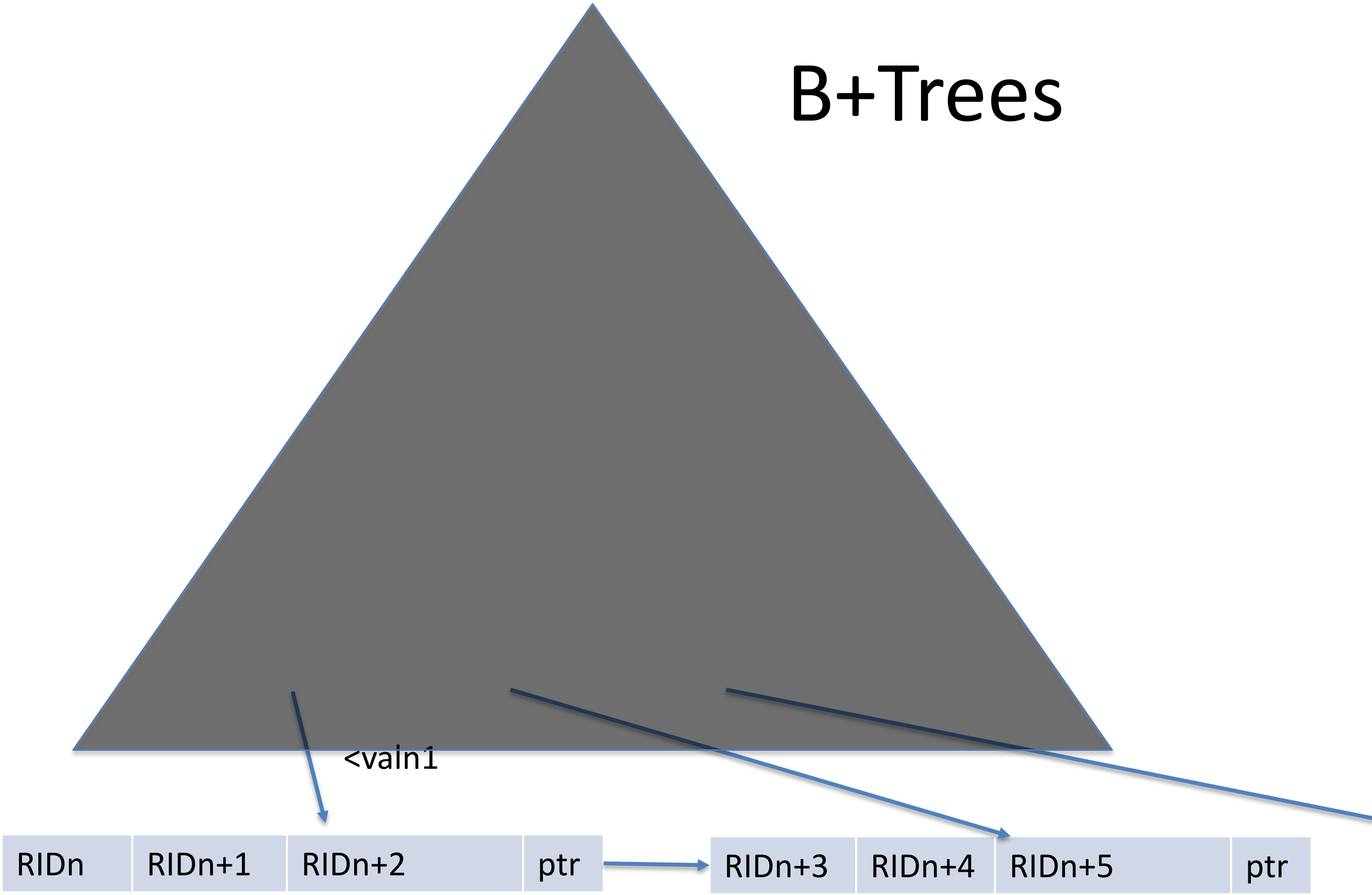
<valn1

RIDn | RIDn+1 | RIDn+2 | ptr

RIDn+3 | RIDn+4 | RIDn+5 | ptr

Leaf nodes; records in Attr A order, w/ link pointers

B+Trees



<valn1

RIDn | RIDn+1 | RIDn+2 | ptr

RIDn+3 | RIDn+4 | RIDn+5 | ptr

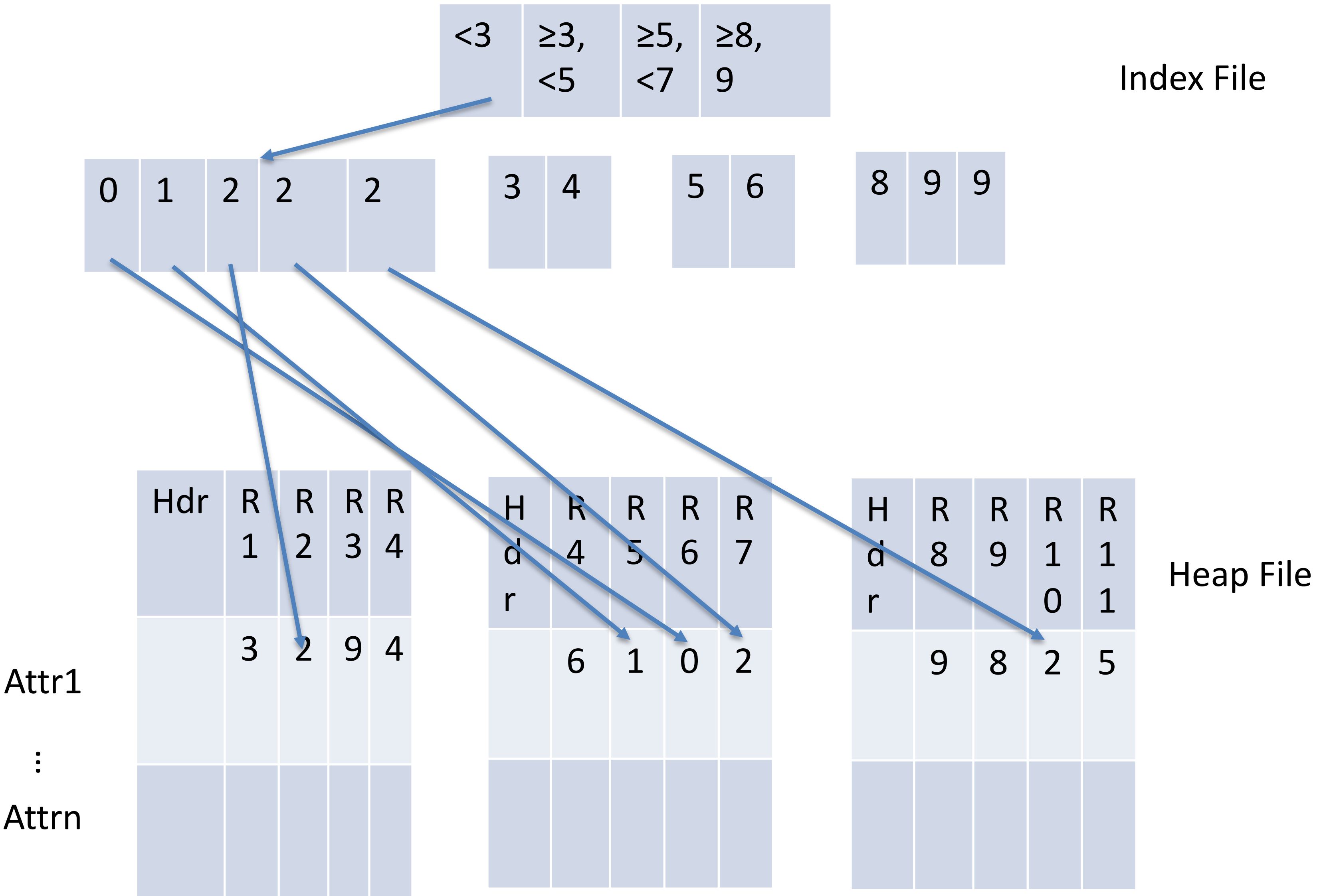
Leaf nodes; records in Attr A order, w/ link pointers

Properties of B+Trees

- Branching factor = B
- Log_B(tuples) levels
- Logarithmic insert/delete/lookup performance
- Support for range scans

- Link pointers
- No data in internal pages
- Balanced (see text “rotation”) algorithms to rebalance on insert/delete
- Fill factor: All nodes except root kept at least 50% full (merge when falls below)
- Clustered / unclustered

Unclustered Index



<3 | ≥3, <5 | ≥5, <7 | ≥8, 9

Index File

0 1 2 2 2 | 3 4 | 5 6 | 8 9 9

Hdr	R1	R2	R3	R4
	3	2	9	4

Hdr	R4	R5	R6	R7
	6	1	0	2

Hdr	R8	R9	R10	R11
	9	8	2	5

Heap File

Attr1

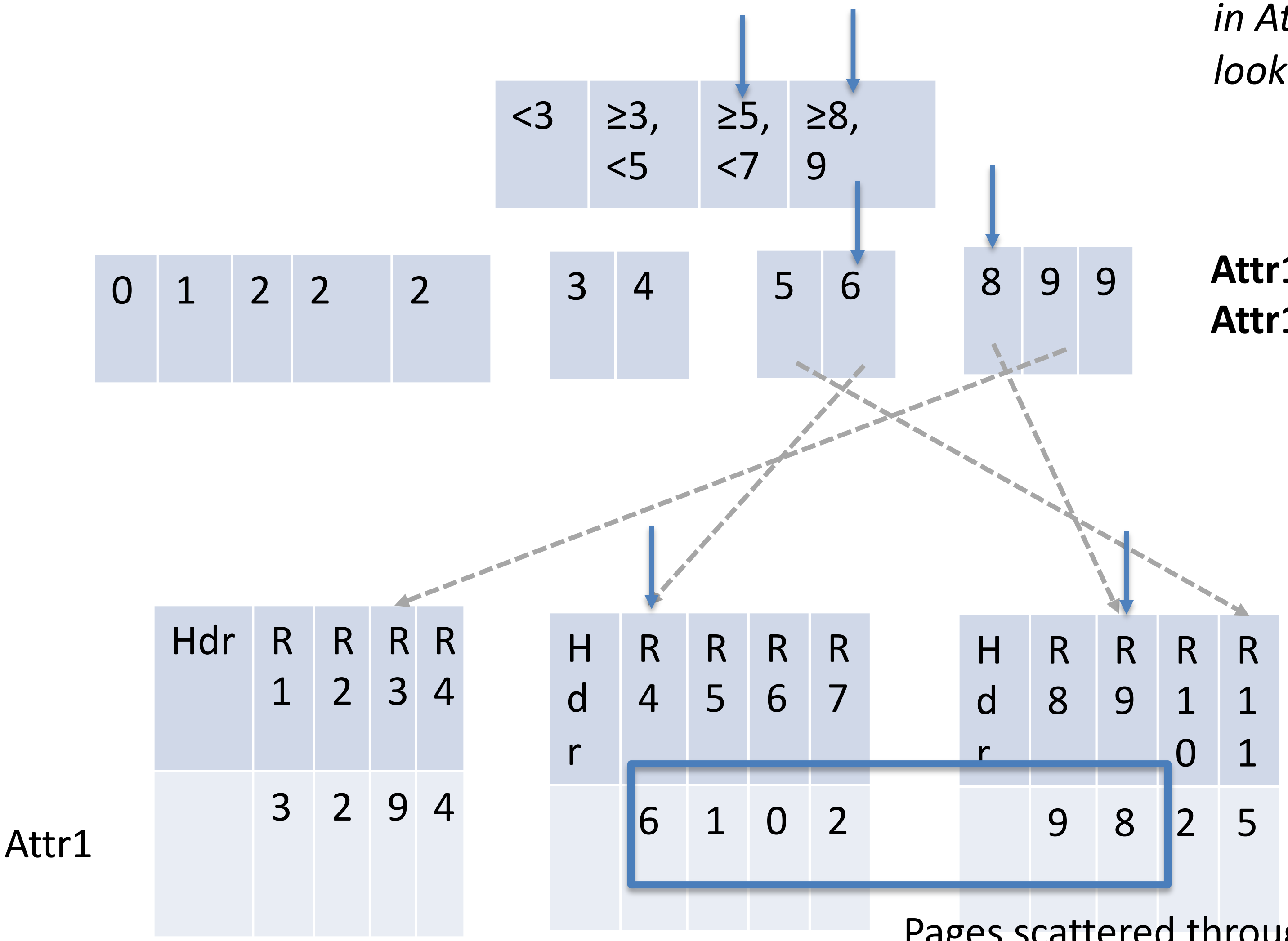
⋮

Attrn

Index Scan

Traverse the records in Attr1 order, or lookup a range

Attr1 >= 6 & Attr1 < 9

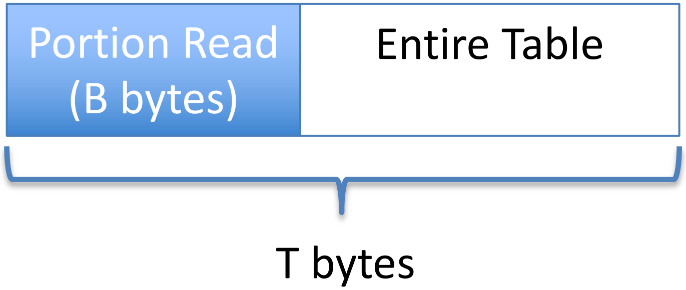


Heap File

Pages scattered throughout heapfile

Note random access! – this is an "unclustered" index

Costs of Random Access



- Consider an SSD with 100 usec “seek” latency, 1 GB/sec BW
- Query accesses B bytes
- R bytes per record, whole table is T bytes
- Seq scan time S = T / 1GB/sec
- Rand access via index time = 100 usec * B/R + B / 1GB/sec
- Suppose R is 100 bytes, T is 10 GB

Time to scan B bytes

Number of records

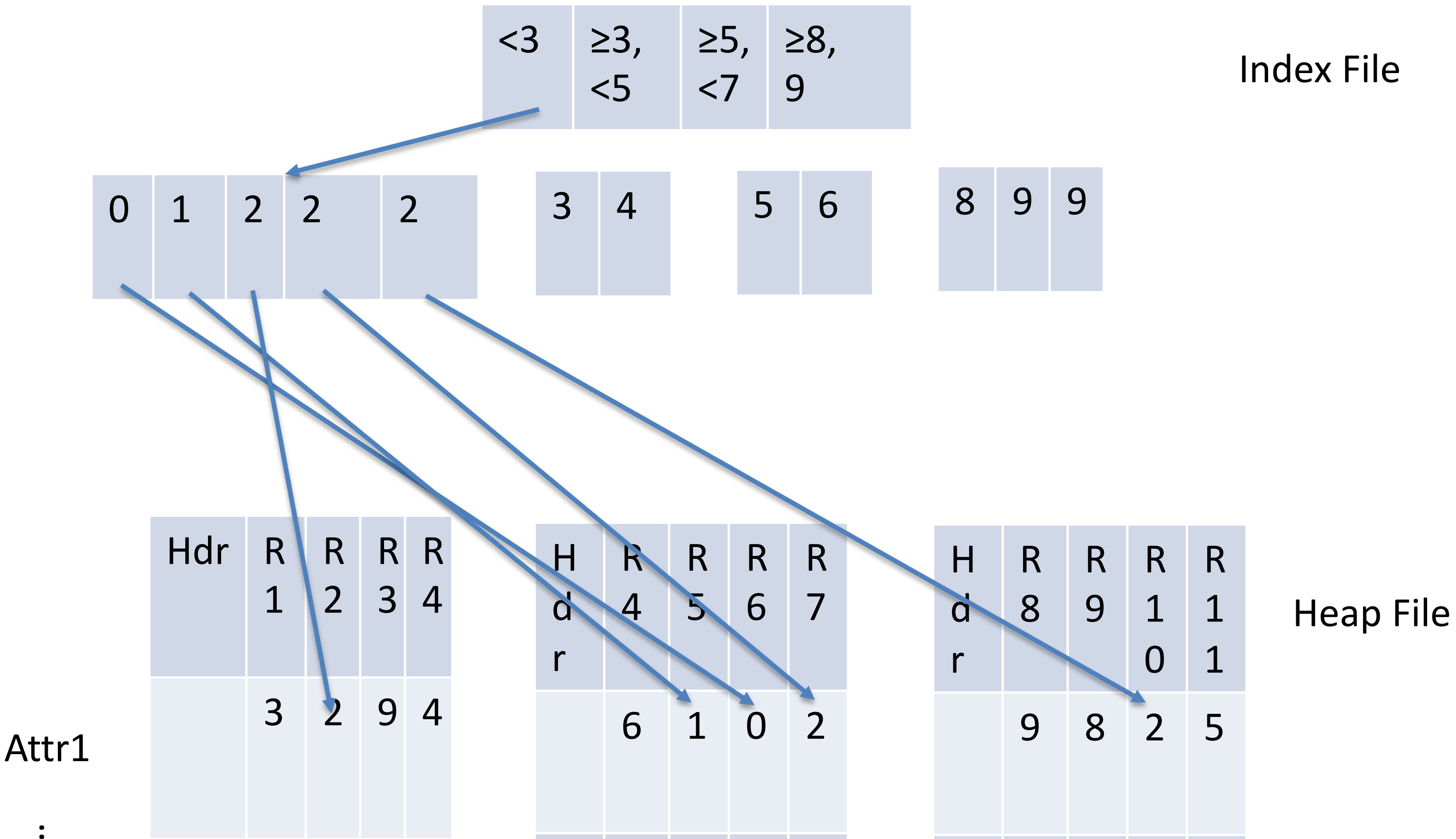
- When is it cheaper to scan than do random lookups via index?

$$100\times10^{-6} * B / 100 + B/1\times10^{9} > 10\times10^{9} / 1\times10^{9}$$
$$1\times10^{-6}B + 1\times10^{-9}B > 10$$
$$B > 9.99\times10^{6}$$

For scans of larger than 10 MB, cheaper to scan entire 10 GB table than to use index

Clustered Index

- Order pages on disk in index order



<3	≥3, <5	≥5, <7	≥8, 9

Index File

0	1	2	2	2

3	4

5	6

8	9	9

	Hdr	R1	R2	R3	R4
Attr1		3	2	9	4
:					

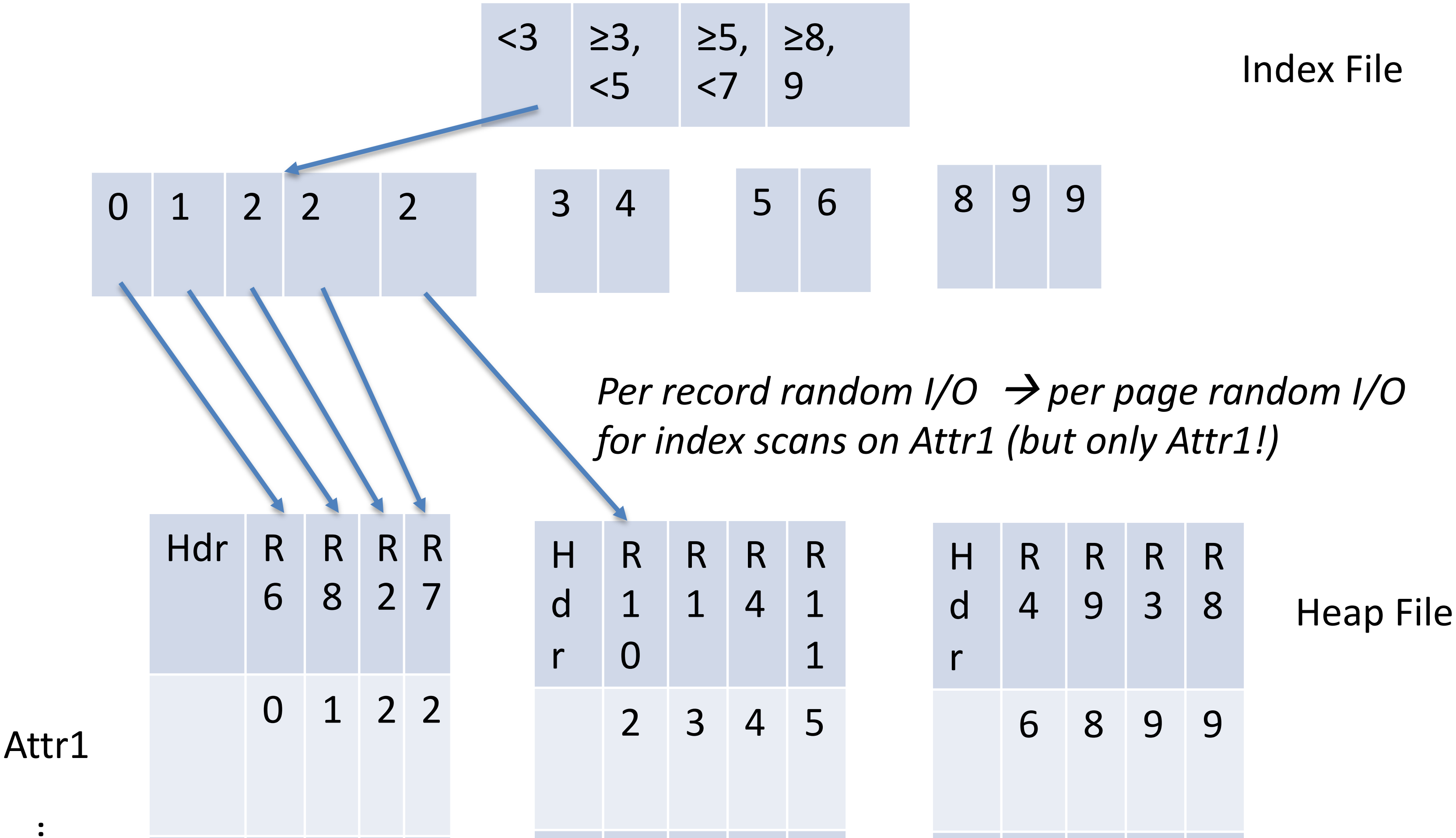
Hdr	R4	R5	R6	R7
	6	1	0	2

Hdr	R8	R9	R10	R11
	9	8	2	5

Heap File

Clustered Index

- Order pages on disk in index order



Per record random I/O → per page random I/O for index scans on Attr1 (but only Attr1!)

Benefit of Clustering

- Consider an SSD with 100 usec latency, 1 GB/sec BW
- Query accesses B bytes, R bytes per record, whole table is T bytes
- **Pages are P bytes**
- Seq scan time S = T / 1GB/sec
- Clustered index access time = 100 usec * B/P~~R~~ + B / 1GB/sec
- Suppose R is 100 bytes, T is 10 GB, **P is 1 MB**

- When is it cheaper to scan than do random lookups via clustered index?

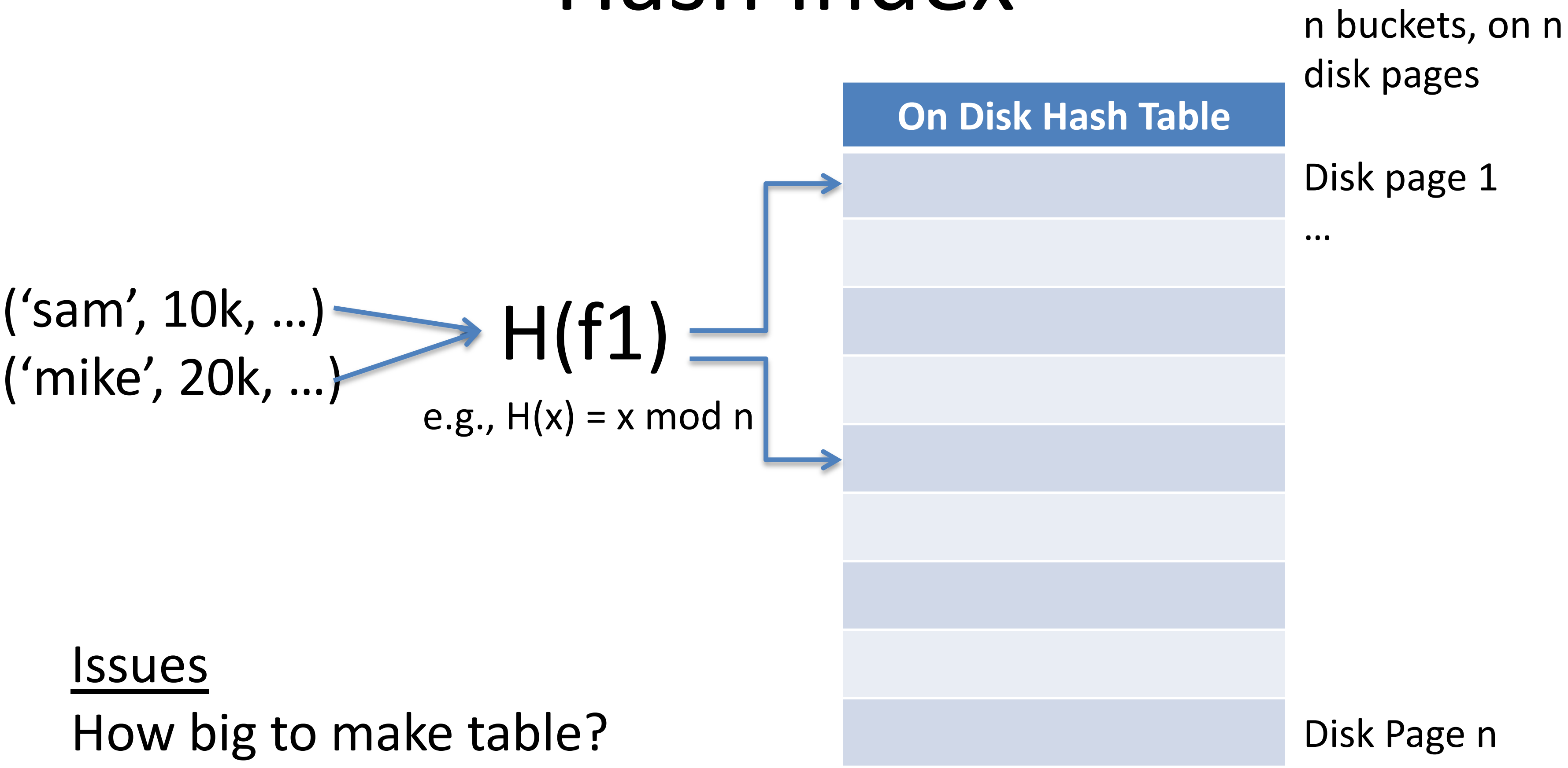
$100 \times 10^{-6} * B / \mathbf{1 \times 10^{6}} + B/1 \times 10^{9} > 10 \times 10^{9} / 1 \times 10^{9}$

$1 \times 10^{-12}B + 1 \times 10^{-9}B > 10$

$B > 9.99 \times 10^{9}$

For scans of larger than 9.9 GB, cheaper to scan entire 10 GB table than to use **clustered** index

Hash Index



Issues

How big to make table?

If we get it wrong, **either**

waste space, **or**

end up with long overflow chains, **or**

have to rehash

Extensible Hashing

- Create a family of hash tables parameterized by k

  $$H_k(x) = H(x) \bmod 2^k$$

- Start with k = 1 (2 hash buckets)
- Use a directory structure to keep track of which bucket (page) each hash value maps to
- When a bucket overflows, increment k (if needed), create a new bucket, rehash keys in overflowing bucket, and update directory

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

- What indexes would you create on `emp` for the following queries (assuming each query is the only query the database runs and emp is really really large)

```
SELECT MAX(sal) FROM emp
SELECT sal FROM emp WHERE id = 1
SELECT name FROM emp WHERE sal > 100k
SELECT name FROM emp WHERE sal > 100k AND dept = 2
```

(A) BTree, Btree, None, Hash
(B) BTree, Hash, BTree, none
(C) None, Hash, BTree, BTree
(D) BTree, Hash, BTree, BTree

Study Break

- What indexes would you create for the following queries (assuming each query is the only query the database runs)

```
SELECT MAX(sal) FROM emp
```

B+Tree on emp.sal

```
SELECT sal FROM emp WHERE id = 1
```

Hash index on emp.id

```
SELECT name FROM emp WHERE sal > 100k
```

B+Tree on emp.sal (maybe)

```
SELECT name FROM emp WHERE sal > 100k AND dept = 2
```

B+tree on emp.sal (maybe), Hash on dept.dno (maybe)

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)

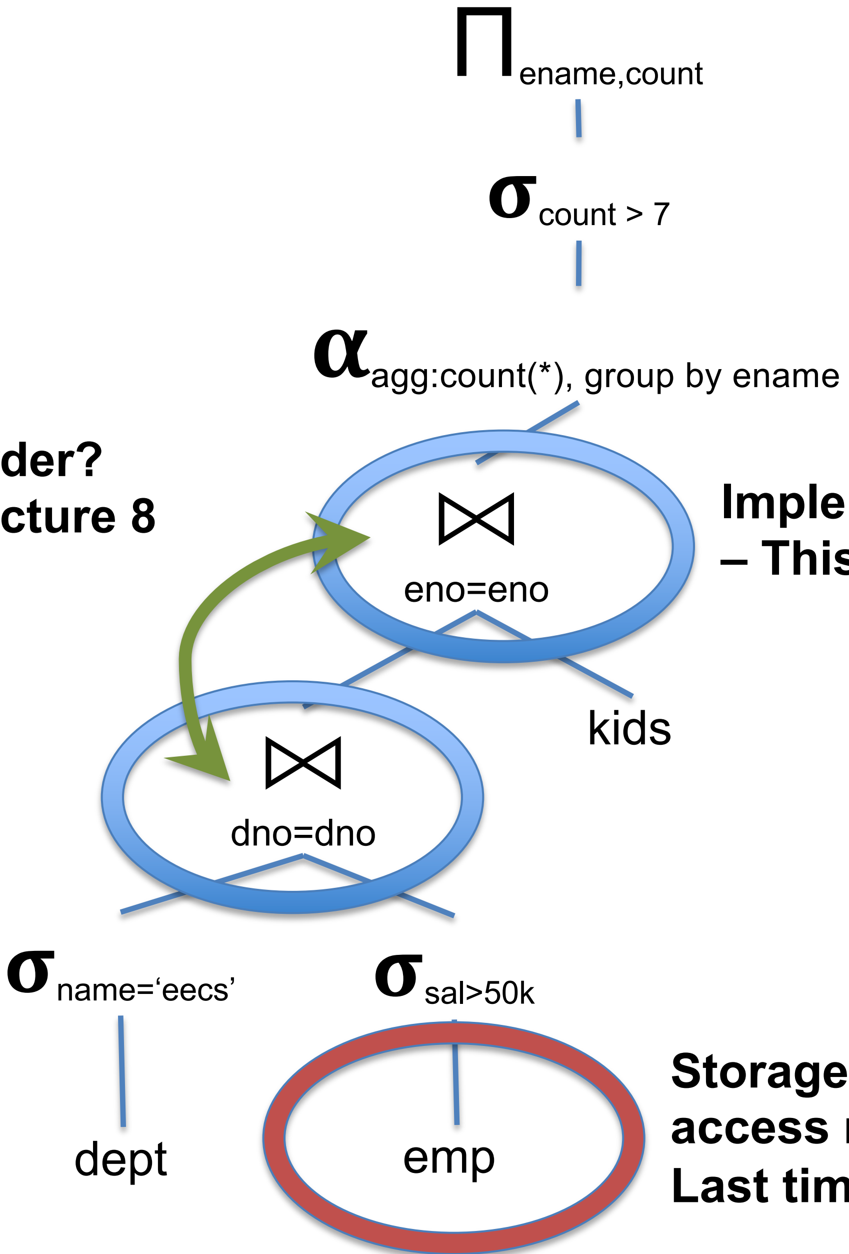
n : number of tuples

P : number of pages in file

B : branching factor of B-Tree

R : number of pages in range

Plan questions



Order?
Lecture 8

Implementation?
– This Lecture

Storage model &
access methods –
Last time

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

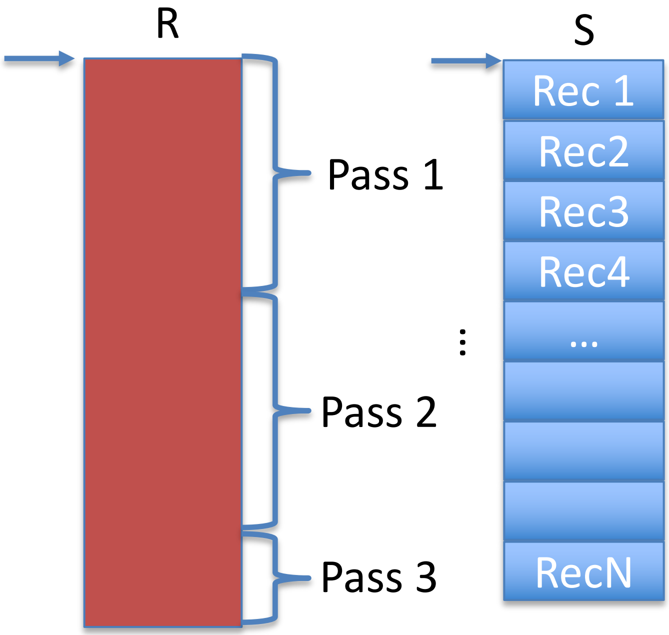
{S} * {R} comparisons in either case

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

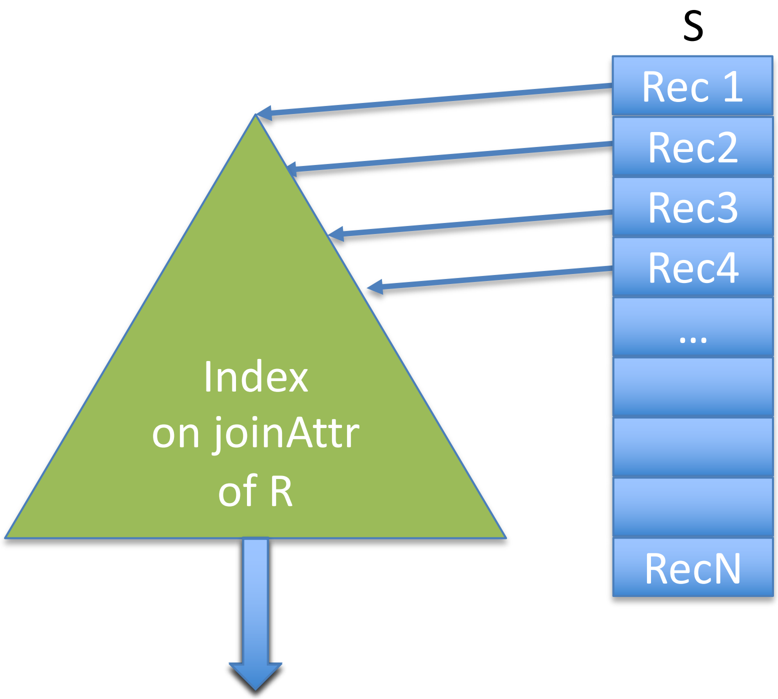
Basic Join Summary

	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}	$\lceil \lvert R \rvert / M \rceil \times \lvert S \rvert$ *Here we use M not B*	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

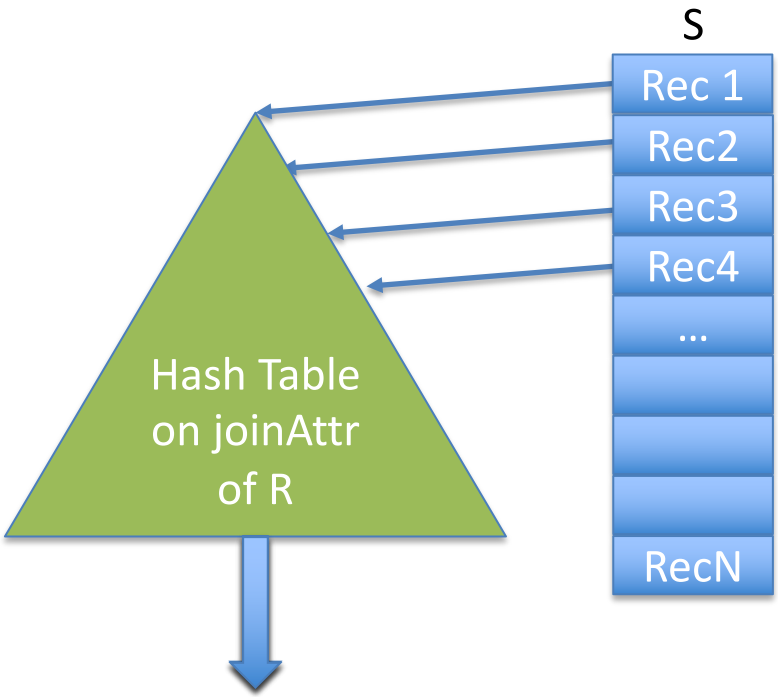
	CPU Complexity	I/O Complexity	Notes
Nested loops	{R} x {S}	|S| + {S}|R| *R doesn't fit in memory* |S| + |R| *R fits in memory*	Choice of inner / outer matters when R fits in memory and S doesn't
Blocked nested loops	{R} x {S}	$\lceil \lvert R \rvert / M \rceil \times \lvert S \rvert$ + |R|	Better to partition R (fewer passes)
Index nested loops	{R} x D *D is tree depth, < ~5*	{R} x D *I/O random unless D 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



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}	$\lceil \lvert R \rvert / M \rceil \times \lvert S \rvert$ + |R|	Better to partition R (fewer passes)
Index nested loops	{S} x D *D is tree depth, < ~5*	{S} x D *I/O random unless D sorted & index clustered on join attr*	Assuming index on R.
Hash join	{R} + {S}	|R| + |S|	Both tables must fit in 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

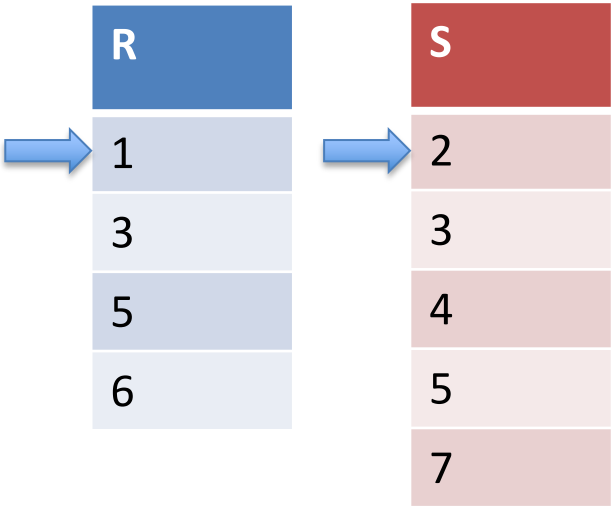
Basic Join Summary

	CPU Complexity	**I/O Complexity**	**Notes**
Nested loops	{R} x {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} x {S}	$\lceil \lvert R\rvert / M \rceil \times \lvert S\rvert + \lvert R\rvert$	Better to partition R (fewer passes)
Index nested loops	{R} x D *D is tree depth, < ~5*	{R} x D *I/O random unless D 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	$\{R\} + (\lceil \lvert R\rvert / M \rceil \times \{S\})$	$\lceil \lvert R\rvert / M \rceil \times \lvert S\rvert + \lvert R\rvert$	

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
```



R
1
3
5
6

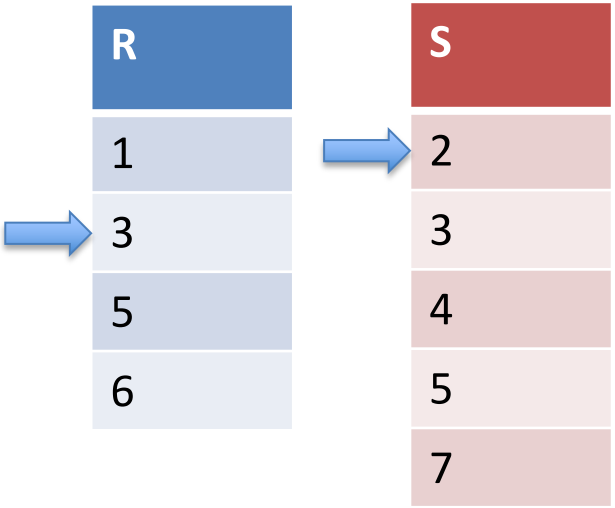
S
2
3
4
5
7

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
```



R
1
3
5
6

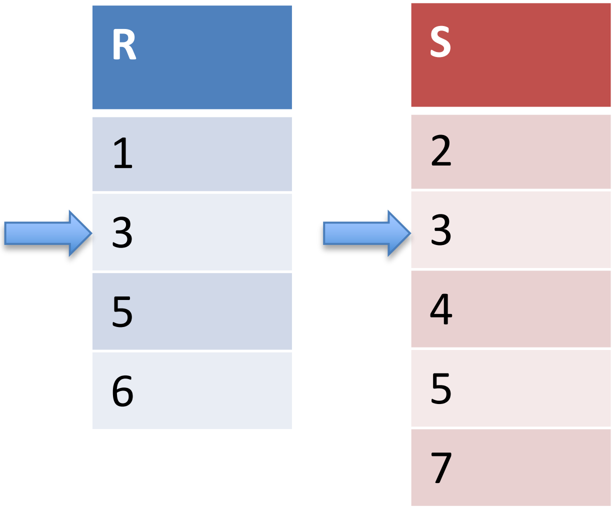
S
2
3
4
5
7

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
```



R
1
3
5
6

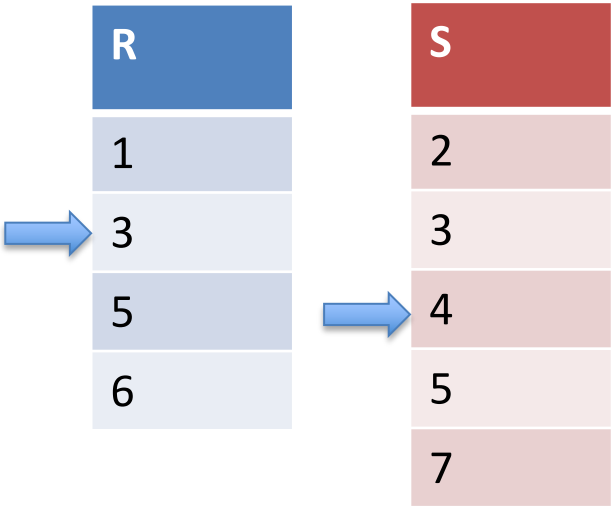
S
2
3
4
5
7

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
```



R
1
3
5
6

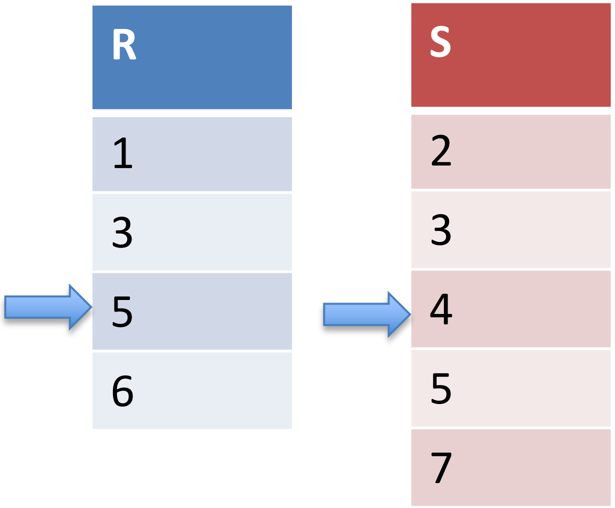
S
2
3
4
5
7

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
```



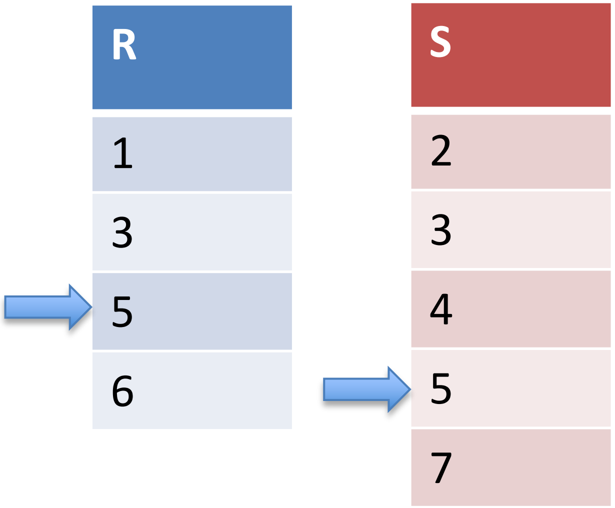
R	S
1	2
3	3
5	4
6	5
	7

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
```



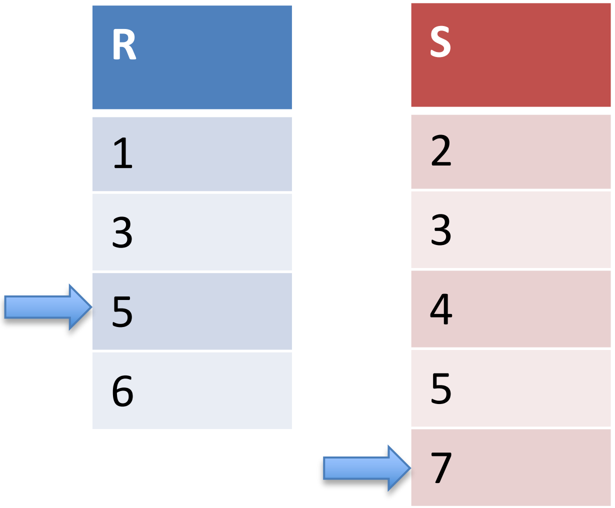
R	S
1	2
3	3
5	4
6	5
	7

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
```

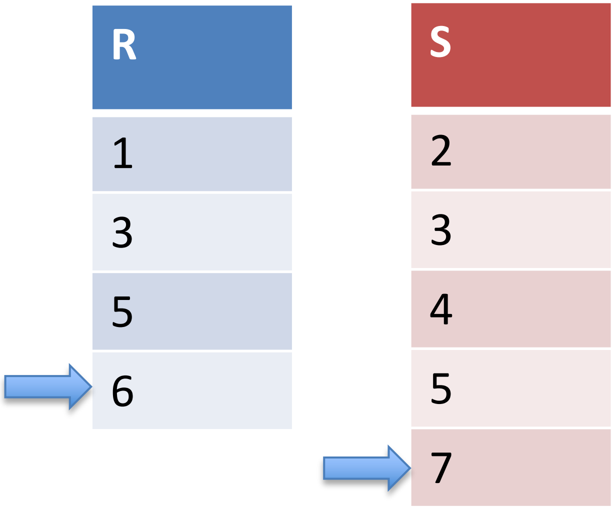


Output: 3, 5

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
```



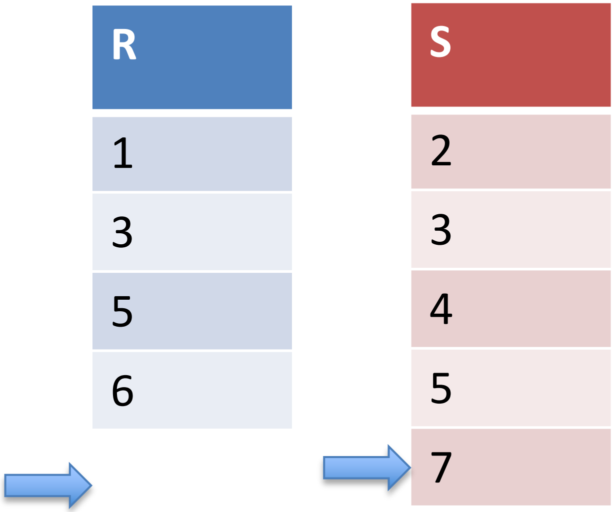
R	S
1	2
3	3
5	4
6	5
	7

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
```



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

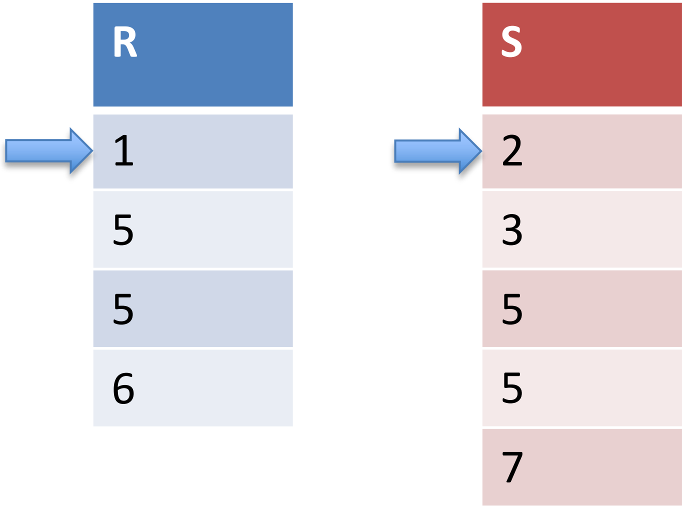
- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

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



R	S
1	2
5	3
5	5
6	5
	7

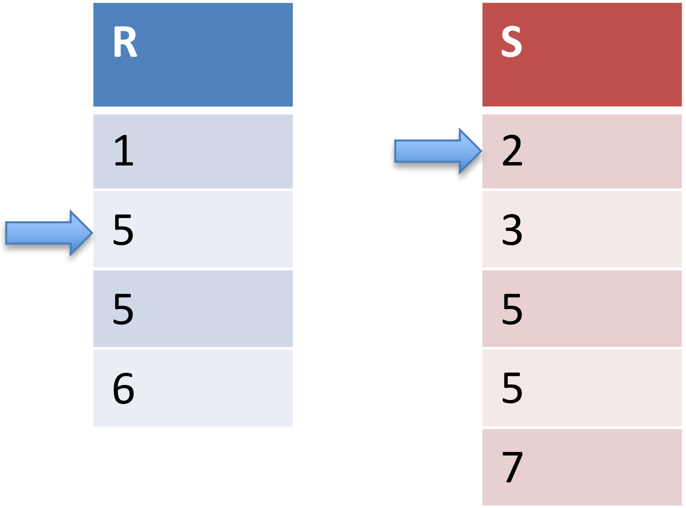
- Solution: count run lengths in S and R, emit cross product of repeated runs

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

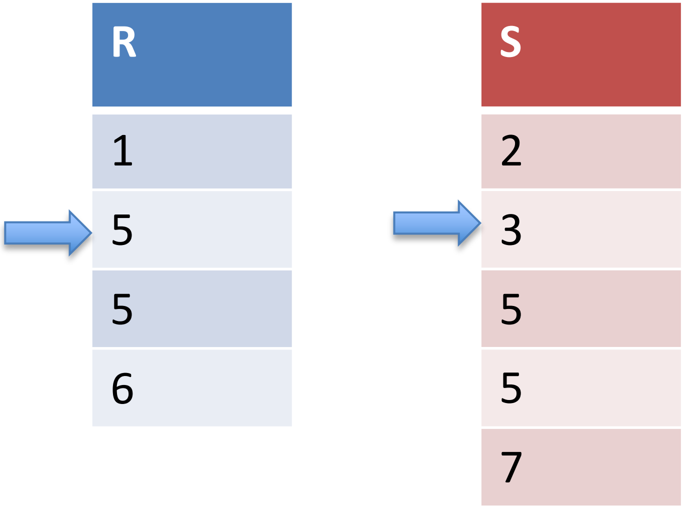
- Solution: count run lengths in S and R, emit cross product of repeated runs

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

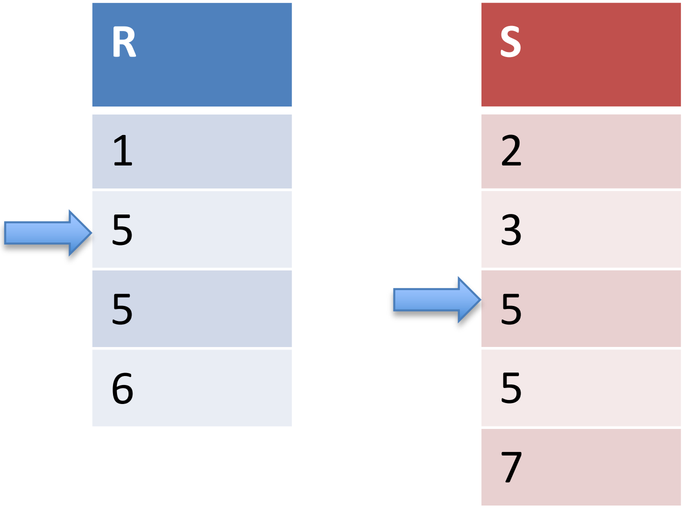
- Solution: count run lengths in S and R, emit cross product of repeated runs

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

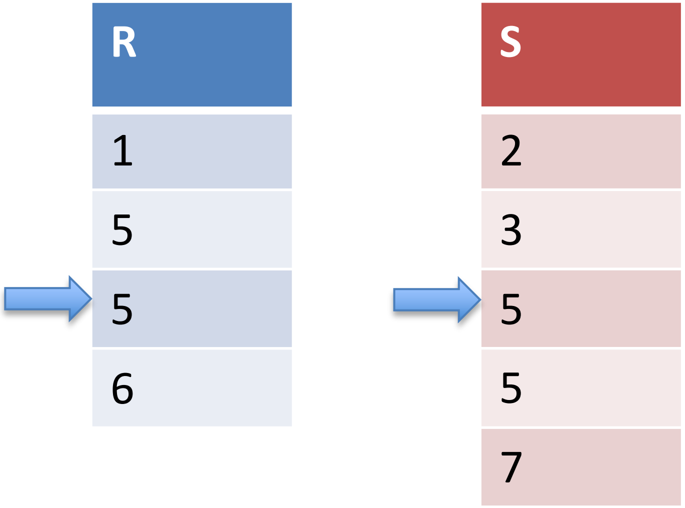
- Solution: count run lengths in S and R, emit cross product of repeated runs

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

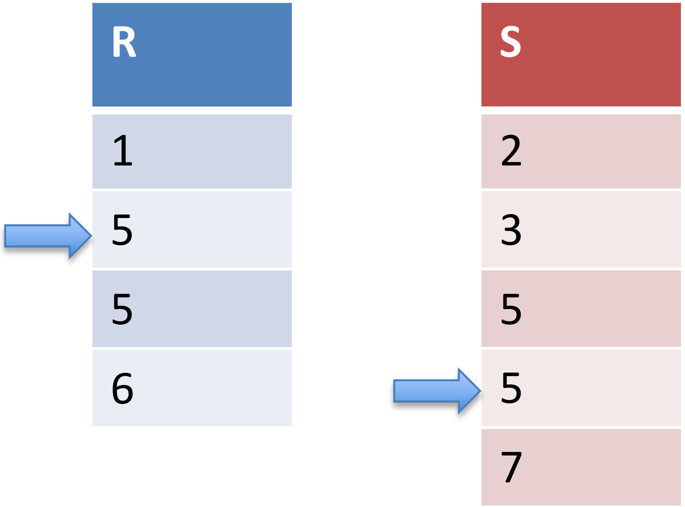
- Solution: count run lengths in S and R, emit cross product of repeated runs

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

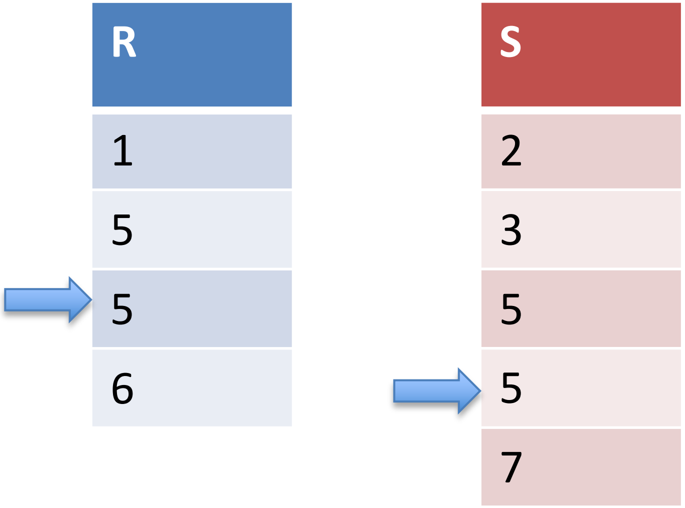
- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

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



R
1
5
5
6

S
2
3
5
5
7

Output: 5, 5, 5

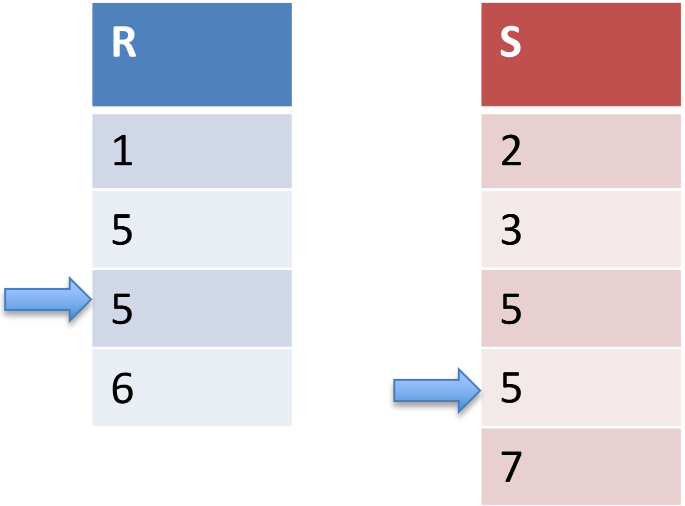
- Solution: count run lengths in S and R, emit cross product of repeated runs

Handling Duplicates

- What is desired output?

4 copies!

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



R
1
5
5
6

S
2
3
5
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	$\{R\} \times \{S\}$	$\lvert S\rvert + \{S\}\lvert R\rvert$ *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	$\{R\} \times \{S\}$	$\lceil \lvert R\rvert / M \rceil \times \lvert S\rvert + \lvert R\rvert$	Better to partition R (fewer passes)
Index nested loops	$\{R\} \times D$ *D is tree depth, < ~5*	$\{R\} \times D$ *I/O random unless D 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	$\{R\} + (\lceil \lvert R\rvert / M \rceil \times \{S\})$	$\lceil \lvert R\rvert / M \rceil \times \lvert S\rvert + \lvert R\rvert$	
Sort merge join	$\{R\}\log\{R\} + \{S\}\log\{S\} + \{S\} + \{R\}$	$\lvert R\rvert + \lvert S\rvert$	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

GPT 4: What is some of the top computer science research out of North Dakota State?

As of my last training data up to January 2022, specific research details about North Dakota State University (NDSU) might not be as prominent on a global scale as some of the larger research-focused universities like MIT or Stanford. However, NDSU is a reputable institution and has research contributions across various fields, including computer science.

“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

R1

S1

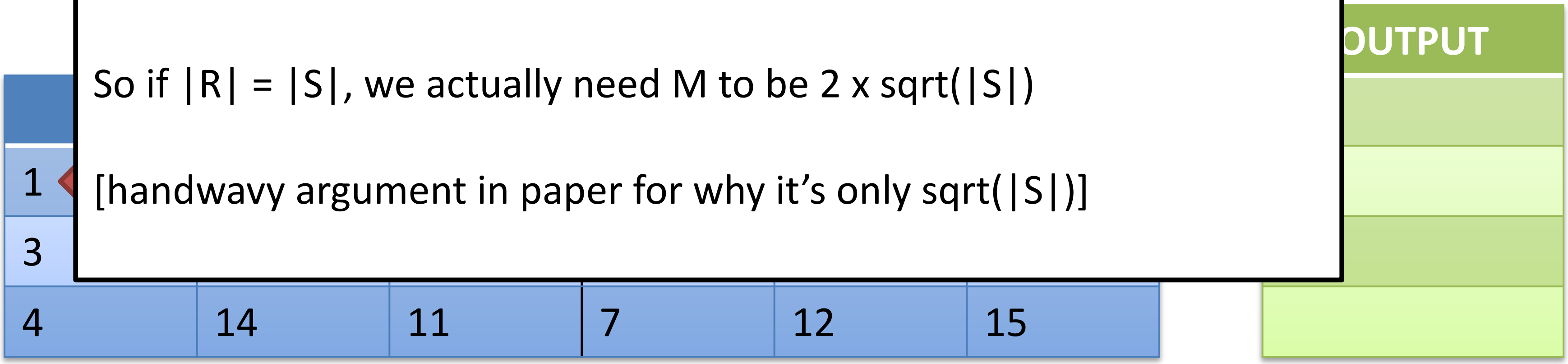
If each run is M pages and M > sqrt(|S|), then there are at most

|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|)]



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

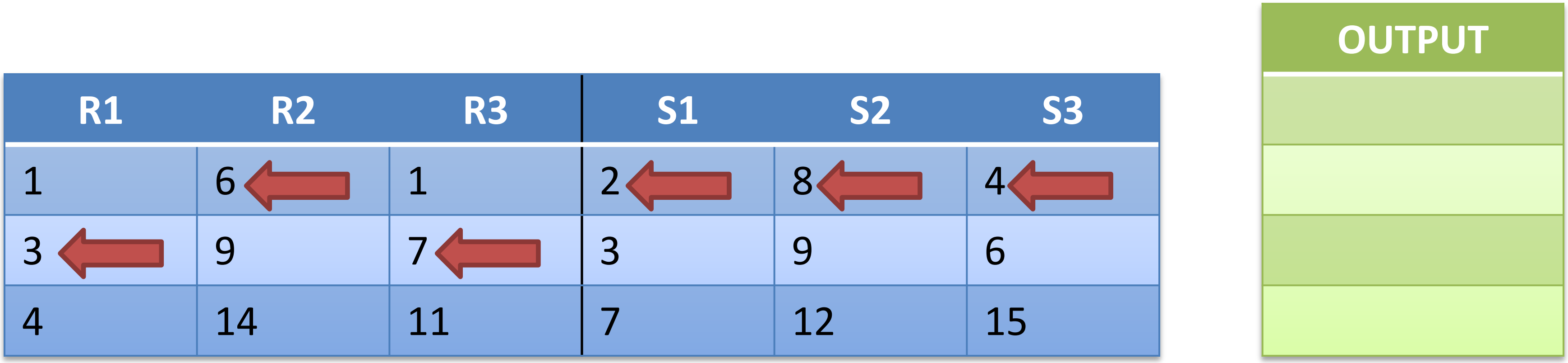
Example

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

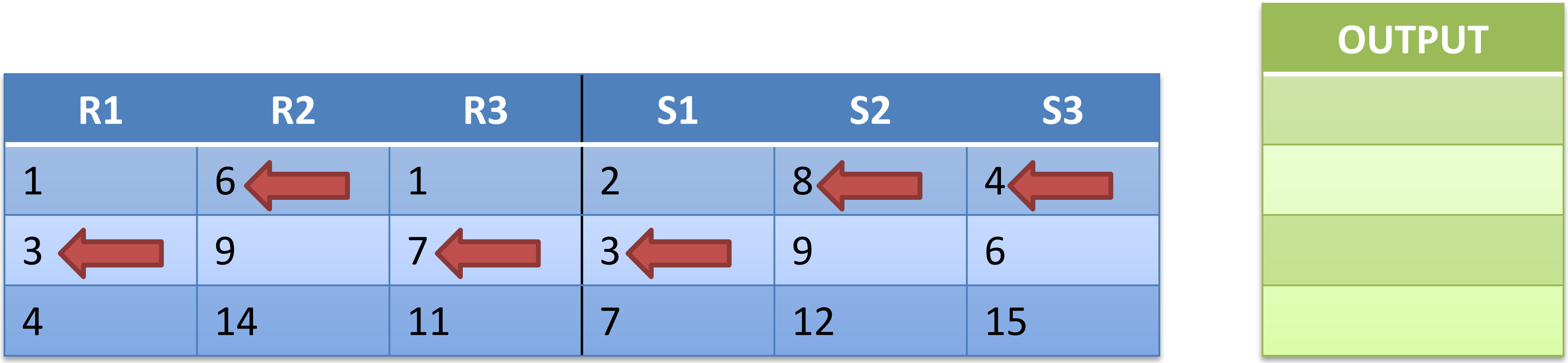
Example

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

S1 = 2,3,7

R2 = 6,9,14

S2 = 8,9,12

R3 = 1,7,11

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

S1 = 2,3,7

R2 = 6,9,14

S2 = 8,9,12

R3 = 1,7,11

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)

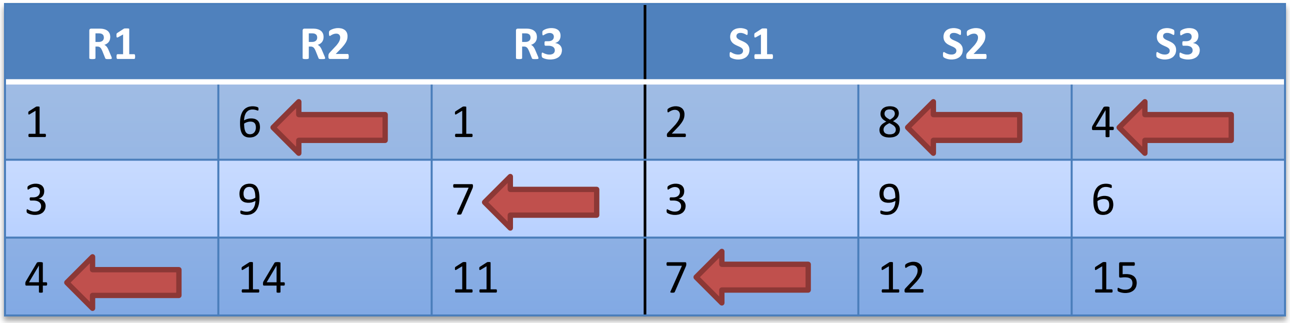
Example

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

S1 = 2,3,7

R2 = 6,9,14

S2 = 8,9,12

R3 = 1,7,11

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

S1 = 2,3,7

R2 = 6,9,14

S2 = 8,9,12

R3 = 1,7,11

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

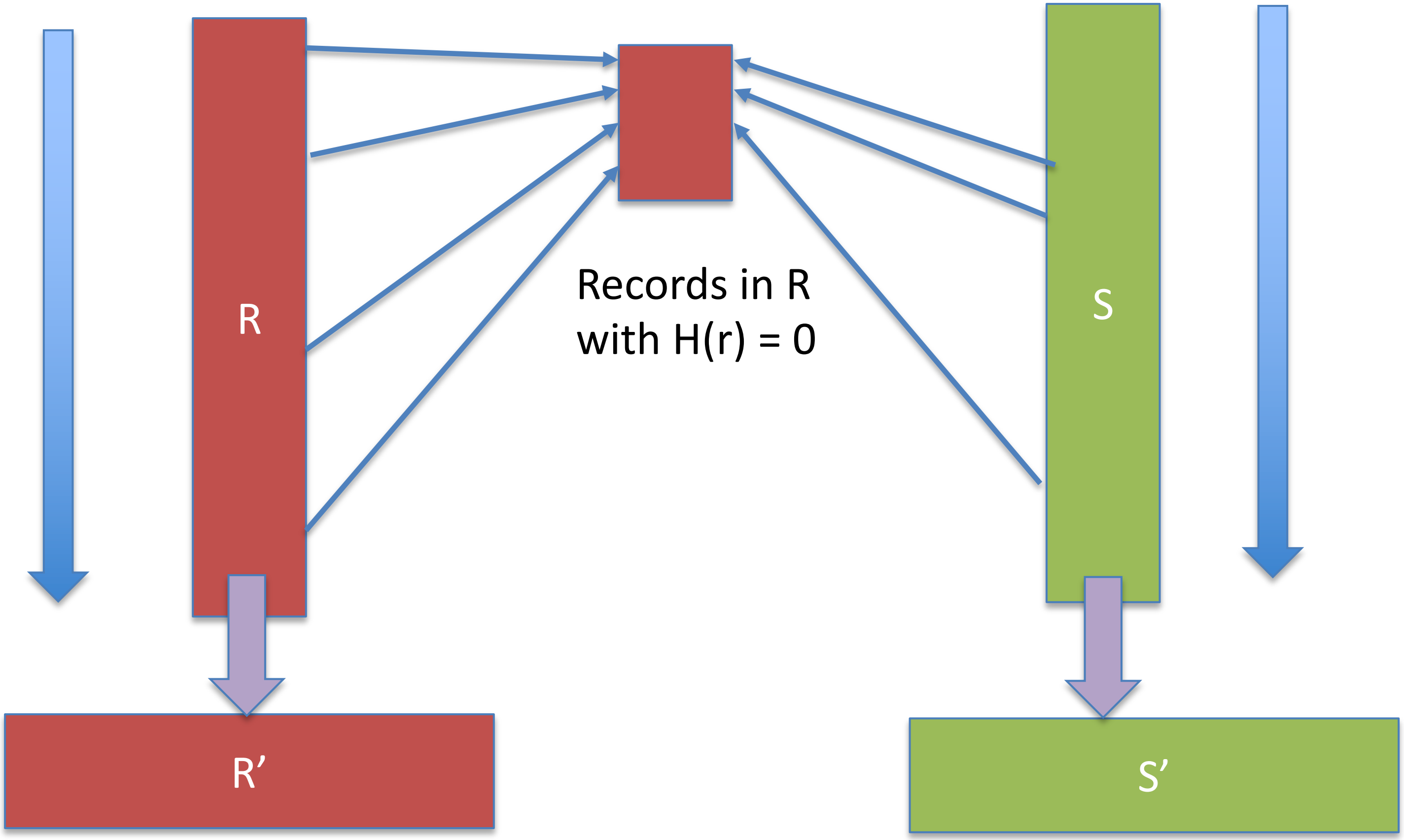
Illustration

Hash function in
0…P

Scan of R

In memory hash table

Scan of S



Records in R
with H(r) = 0

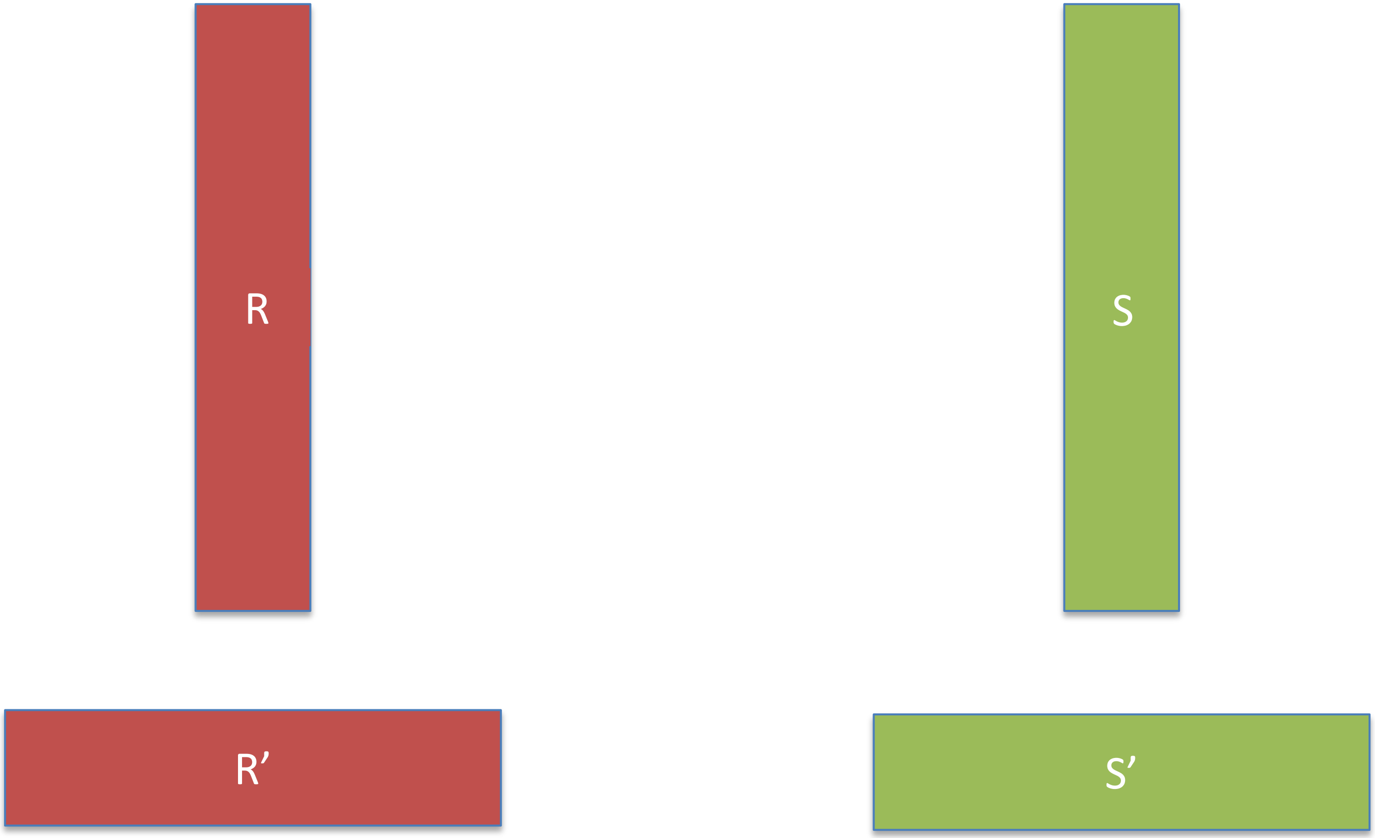
Remainder of R
(Records with H(r) > 0

Remainder of S
(Records with H(s) > 0

Pass 0

Illustration

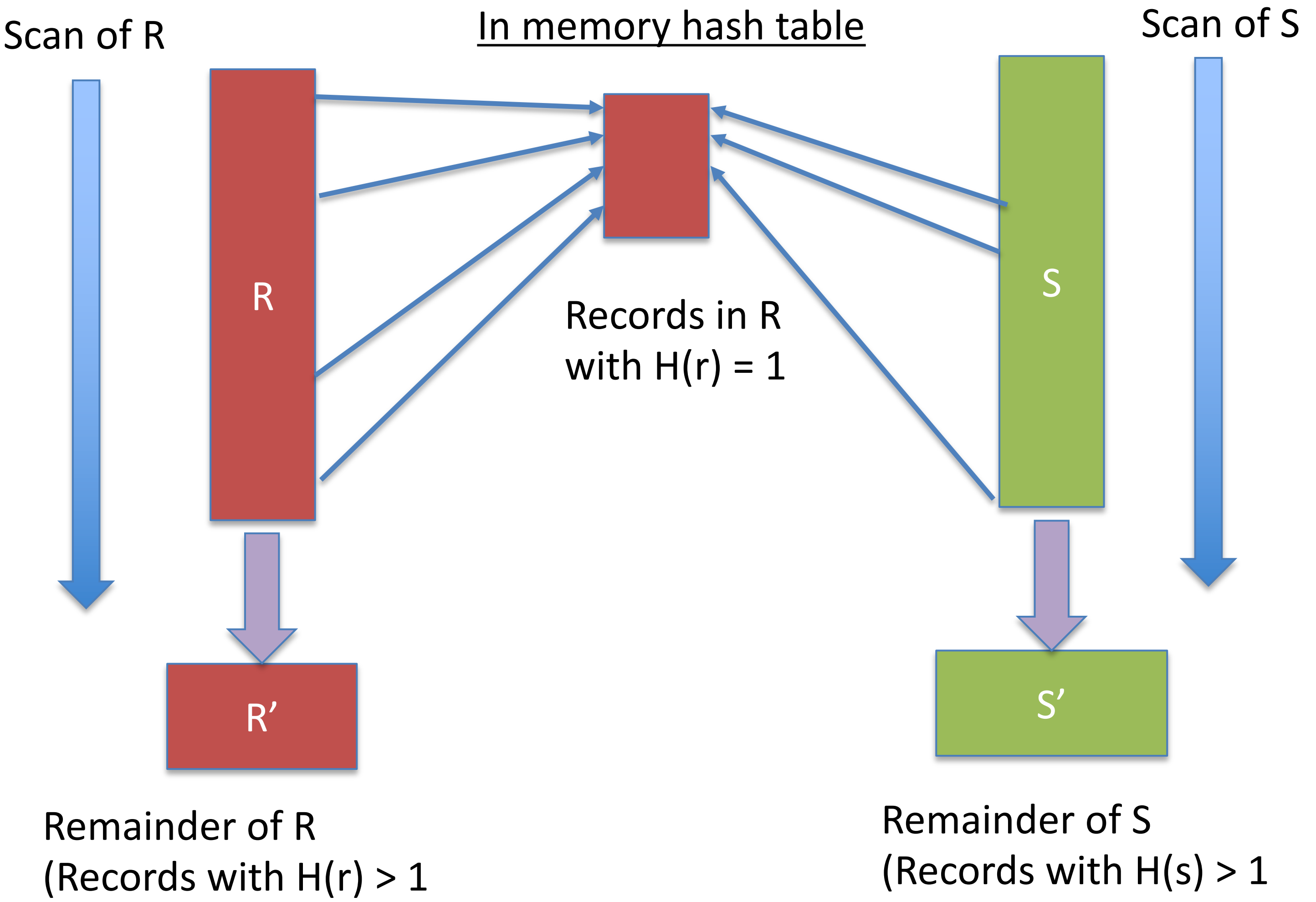
Hash function in 0…P



Pass 1

Illustration

Hash function in 0…P



In memory hash table

Scan of R

Scan of S

R

S

Records in R with H(r) = 1

R’

S’

Remainder of R (Records with H(r) > 1

Remainder of S (Records with H(s) > 1

Repeat for P passes

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)
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
```

Need one page of RAM for each of P partitions

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

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

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

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

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		

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