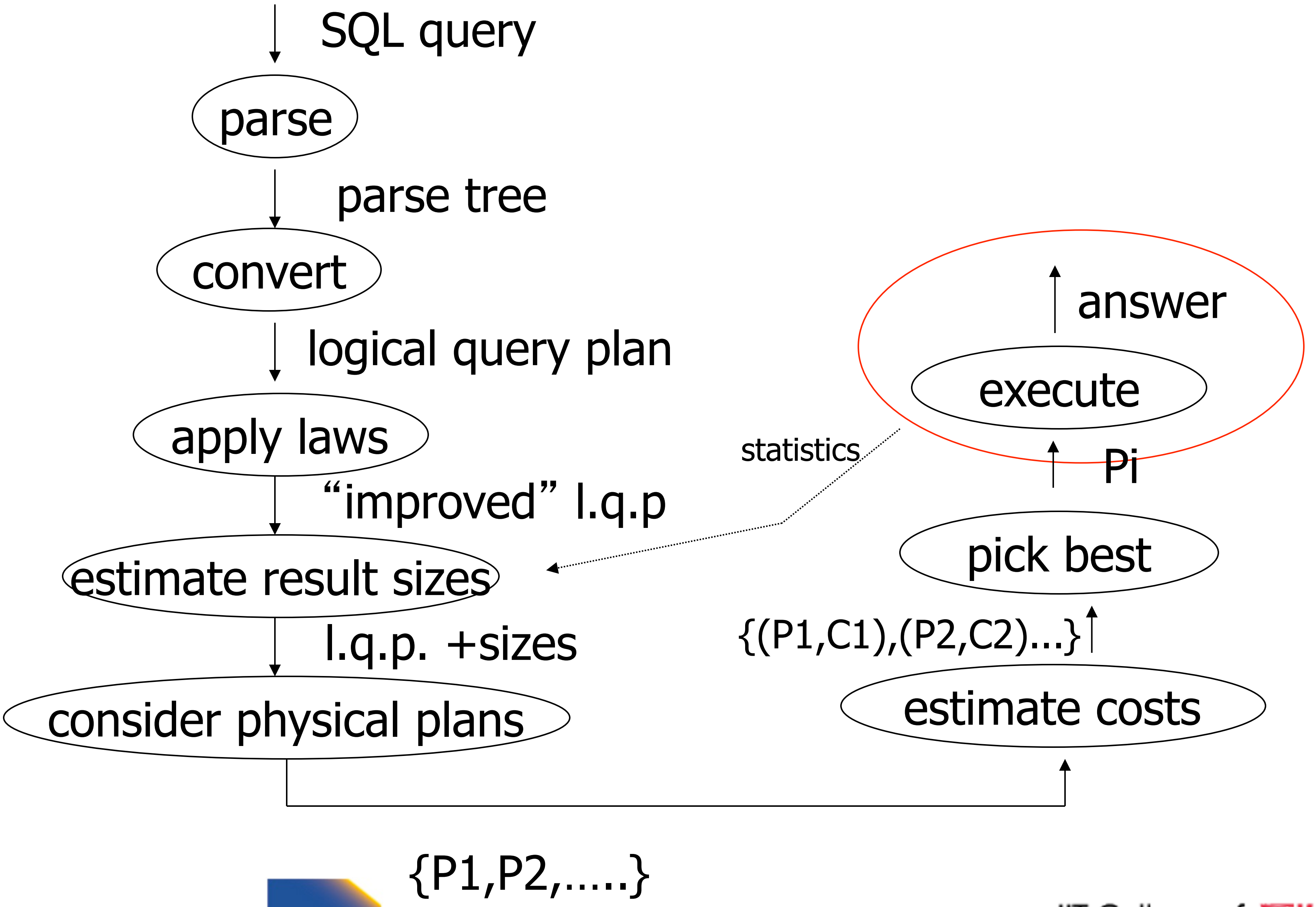
CS 525: Advanced Database Organization

10: Query Execution

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab


SQL query
parse
parse tree
convert
logical query plan
apply laws
“improved” l.q.p
estimate result sizes
l.q.p. +sizes
consider physical plans
{P1,P2,…..}
estimate costs
{(P1,C1),(P2,C2)...}
pick best
Pi
execute
answer
statistics

Query Execution

- Here only:
  - – how to implement operators
  - – what are the costs of implementations
  - – how to implement queries
    - Data flow between operators
- Next part:
  - – How to choose good plan

Execution Plan

- A tree (DAG) of physical operators that implement a query
- May use indices
- May create temporary relations
- May create indices on the fly
- May use auxiliary operations such as sorting

How to estimate costs

- If everything fits into memory
  - Standard computational complexity
- If not
  - Assume fixed memory available for buffering pages
  - Count I/O operations
  - Real systems combine this with CPU estimations

Estimating IOs:

- Count # of disk blocks that must be read (or written) to execute query plan

To estimate costs, we may have additional parameters:

B(R) = # of blocks containing R tuples

f(R) = max # of tuples of R per block

M = # memory blocks available

To estimate costs, we may have additional parameters:

B(R) = # of blocks containing R tuples

f(R) = max # of tuples of R per block

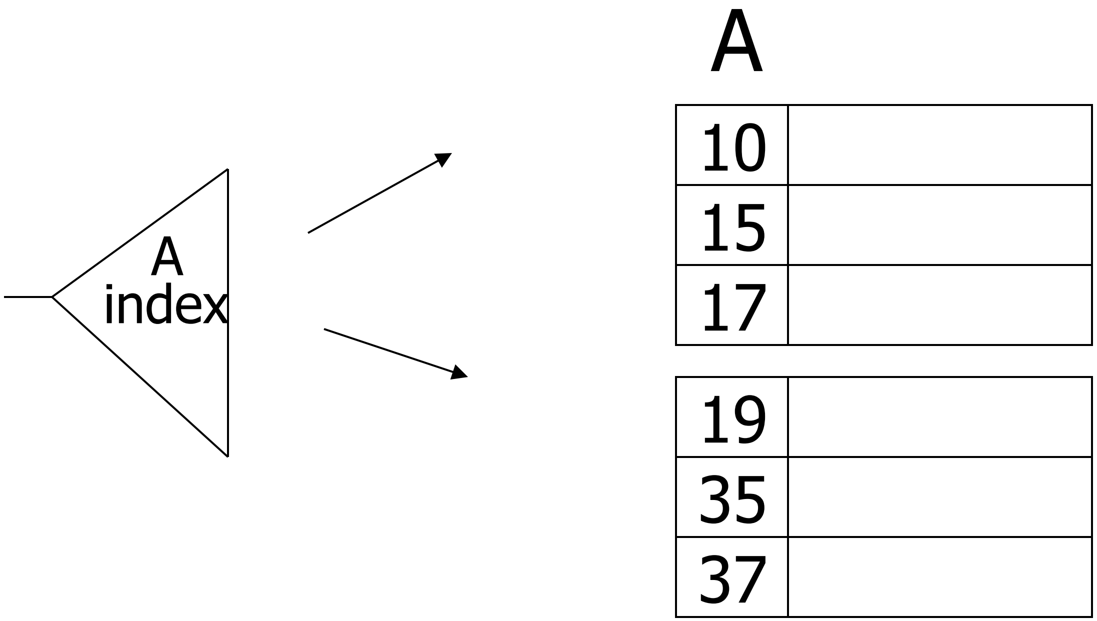
M = # memory blocks available

HT(i) = # levels in index i

LB(i) = # of leaf blocks in index i

Clustered index

Index that allows tuples to be read in an order that corresponds to physical order



Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Operator Profiles

- Algorithm
- In-memory complexity: e.g., $O(n^2)$
- Memory requirements
  - Runtime based on available memory
- #I/O if operation needs to go to disk
- Disk space needed
- Prerequisites
  - Conditions under which the operator can be applied

Execution Strategies

- Compiled
  - Translate into C/C++/Assembler code
  - Compile, link, and execute code
- Interpreted
  - Generic operator implementations
  - Generic executor
    - Interprets query plan

Virtual Machine Approach

- Implement virtual machine of low-level DBMS operations
- Compile query into machine-code for that machine

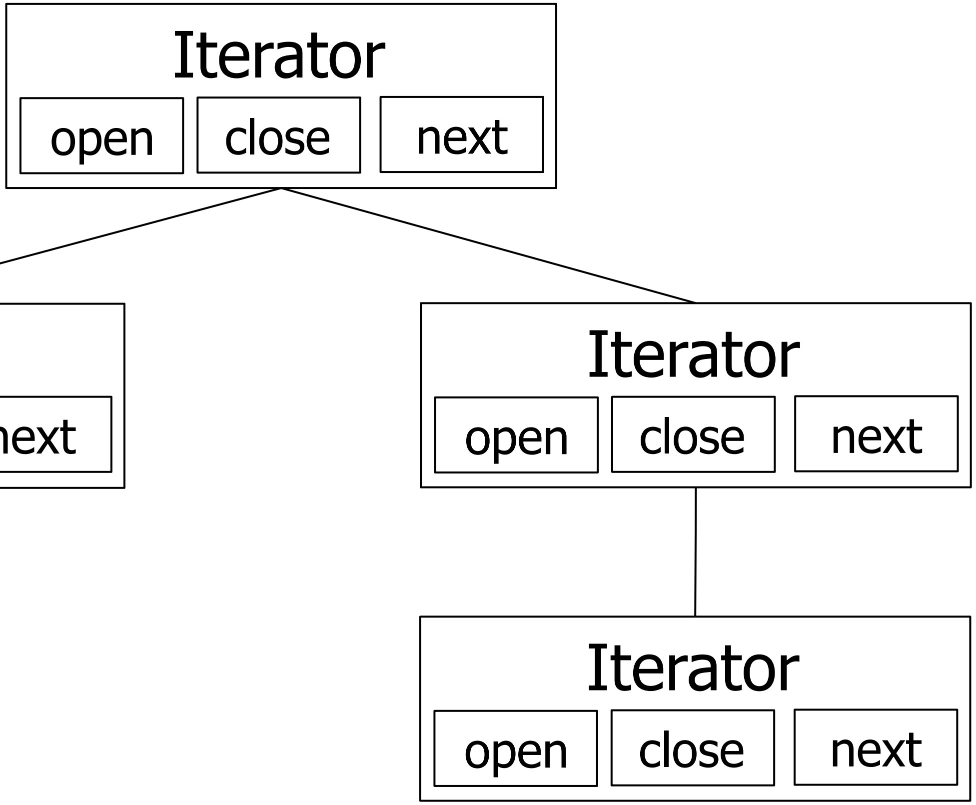
Iterator Model

- Need to be able to combine operators in different ways
  - E.g., join inputs may be scans, or outputs of other joins, …
  - -> define generic interface for operators
  - be able to arbitrarily compose complex plans from a small set of operators

Iterator Model - Interface

- **Open**
  - – Prepare operator to read outputs
- **Close**
  - – Close operator and clean up
- **Next**
  - – Return next result tuple

Query Execution – Iterator Model



Iterator

open close next

Iterator

open close next

Iterator

open close next

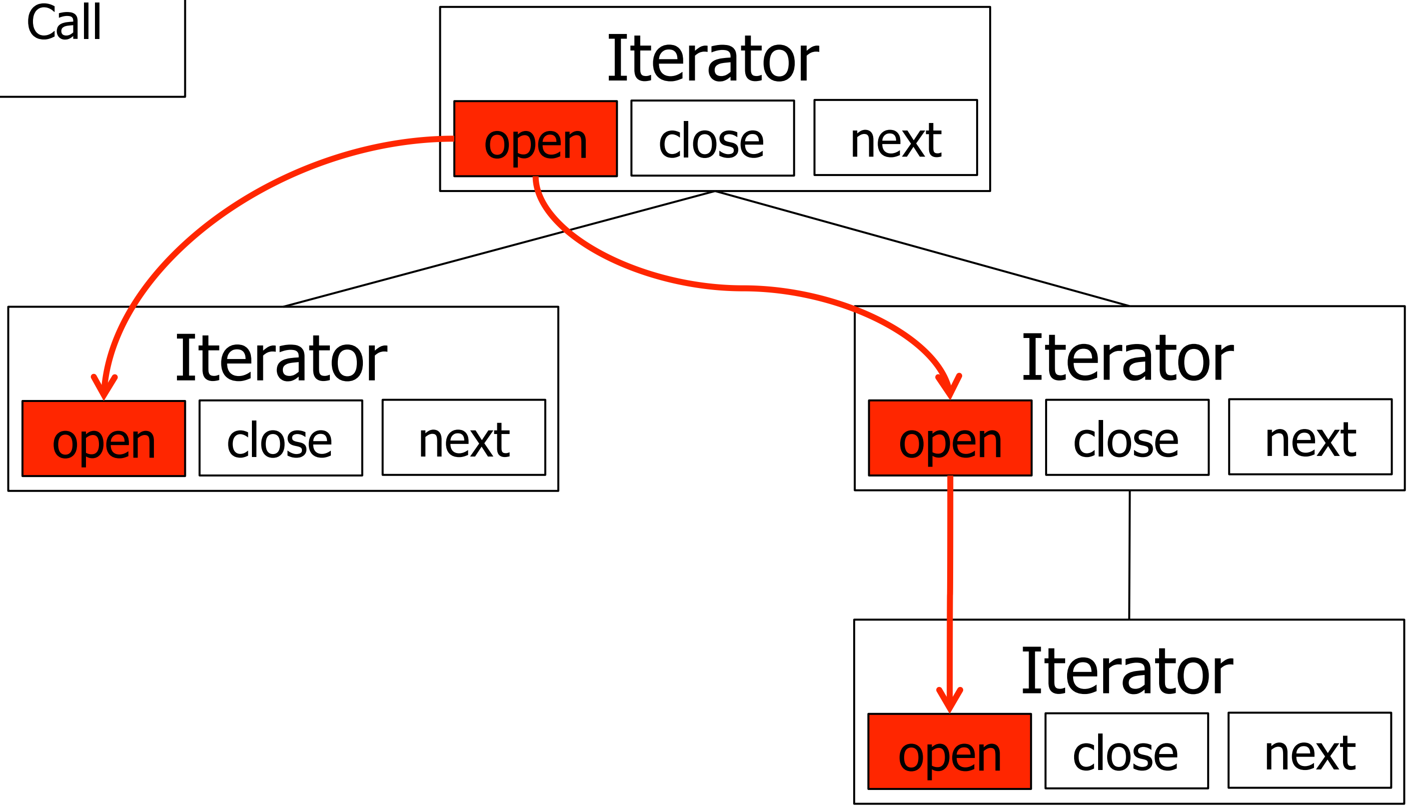
Iterator

open close next

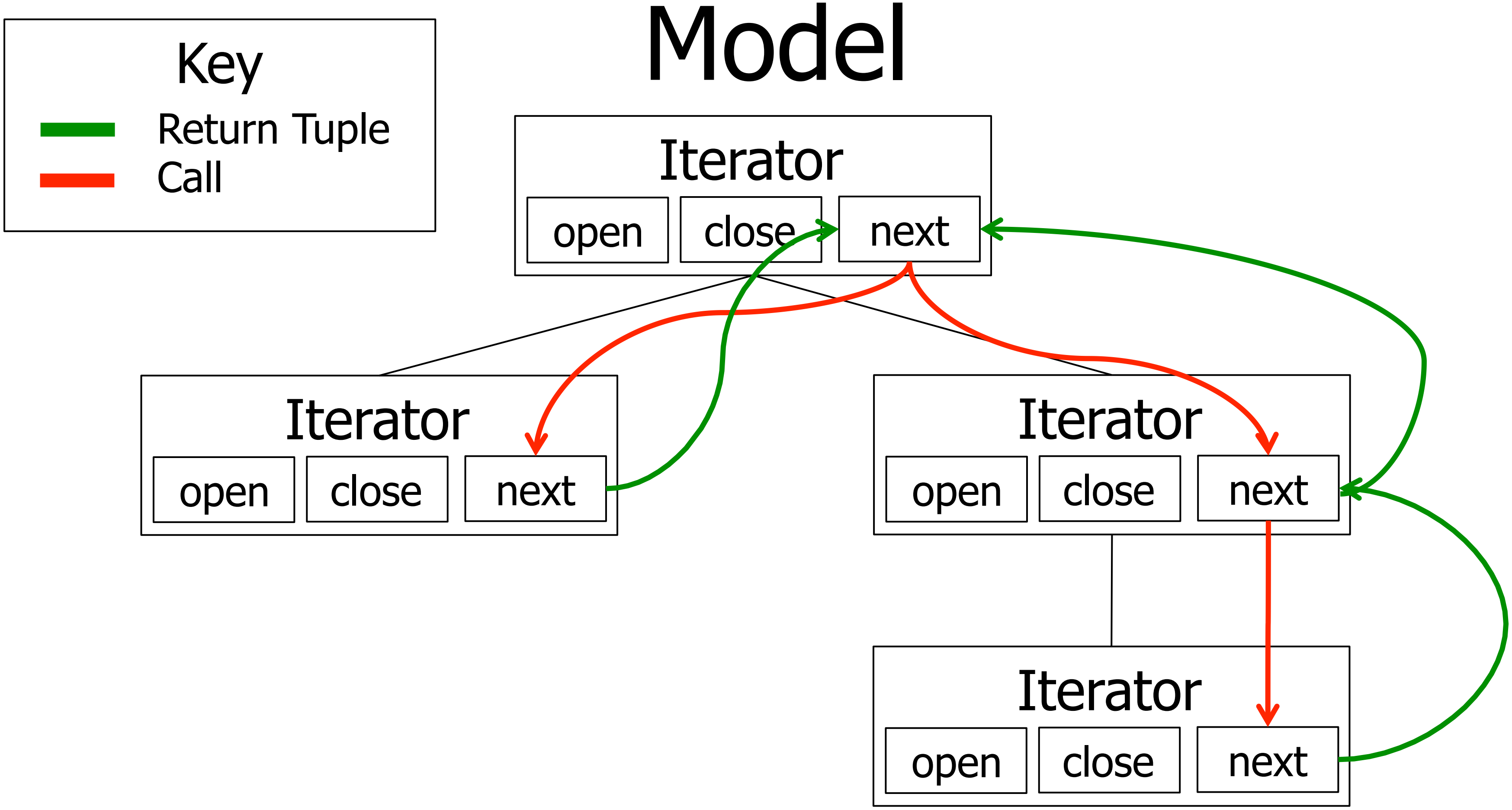
Query Execution – Iterator Model

Key

🟥 Call



Query Execution – Iterator Model



Parallelism

- Iterator Model
  - **Pull-based** query execution
- Potential types of parallelism
  - Inter-query (every multiuser system)
  - Intra-operator
  - Inter-operator

Intra-Operator Parallelism

- Execute portions of an operator in parallel
  - Merge-Sort
    - Assign a processor to each merge phase
  - Scan
    - Partition tables
    - Each process scans one partition

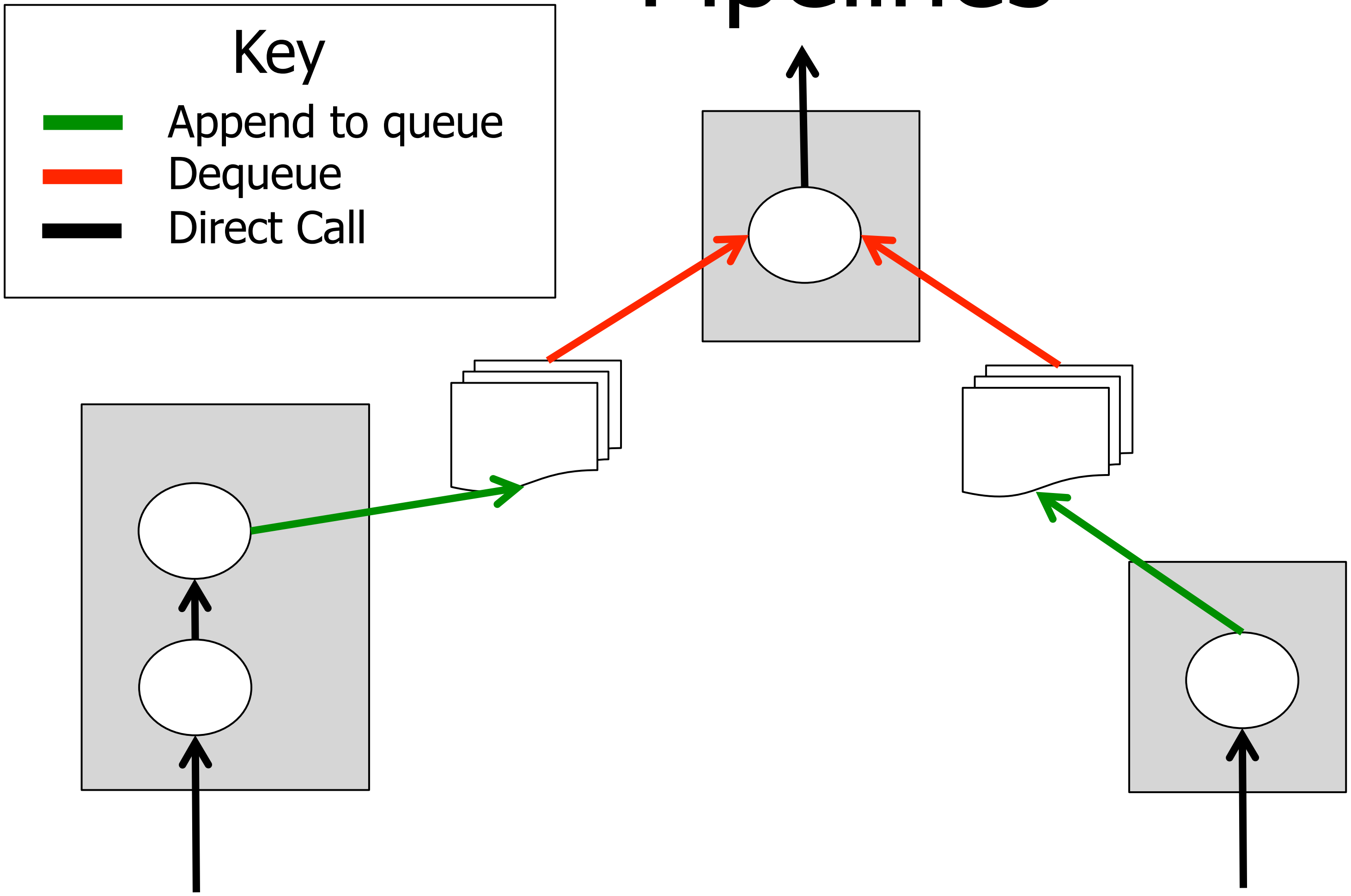
Inter-Operator Parallelism

- Each process executes one or more operators
- **Pipelining**
  - **Push-based** query execution
  - Chain operators directly produce results
  - Pipeline-breakers
    - Operators that need to consume the whole input (or large parts) before producing outputs

Pipelining Communication

- Queues
  - Operators push their results to queues
  - Operators read their inputs from queues
- Direct call
  - Operator calls its parent in the tree with results
  - Within one process

Pipelines



Key

🟩 Append to queue

🟥 Dequeue

⬛ Direct Call

Pipeline-breakers

- Sorting
  - All operators that apply sorting
- Aggregation
- Set Difference
- Some implementations of
  - Join
  - Union

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Sorting

- Why do we want/need to sort
  - – Query requires sorting (ORDER BY)
  - – Operators require sorted input
    - Merge-sort
    - Aggregation by sorting
    - Duplicate removal using sorting

In-memory sorting

- Algorithms from data structures 101
  - Quick sort
  - Merge sort
  - Heap sort
  - Intro sort
  - …

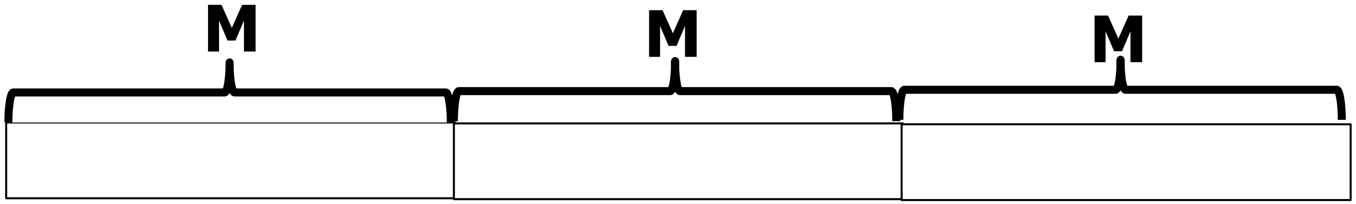
External sorting

- Problem:
  - Sort **N** pages of data with **M** pages of memory

- Solutions?

First Idea

- Split data into runs of size **M**
- Sort each run in memory and write back to disk
  - $\lceil \mathbf{N/M} \rceil$ sorted runs of size **M**
- Now what?



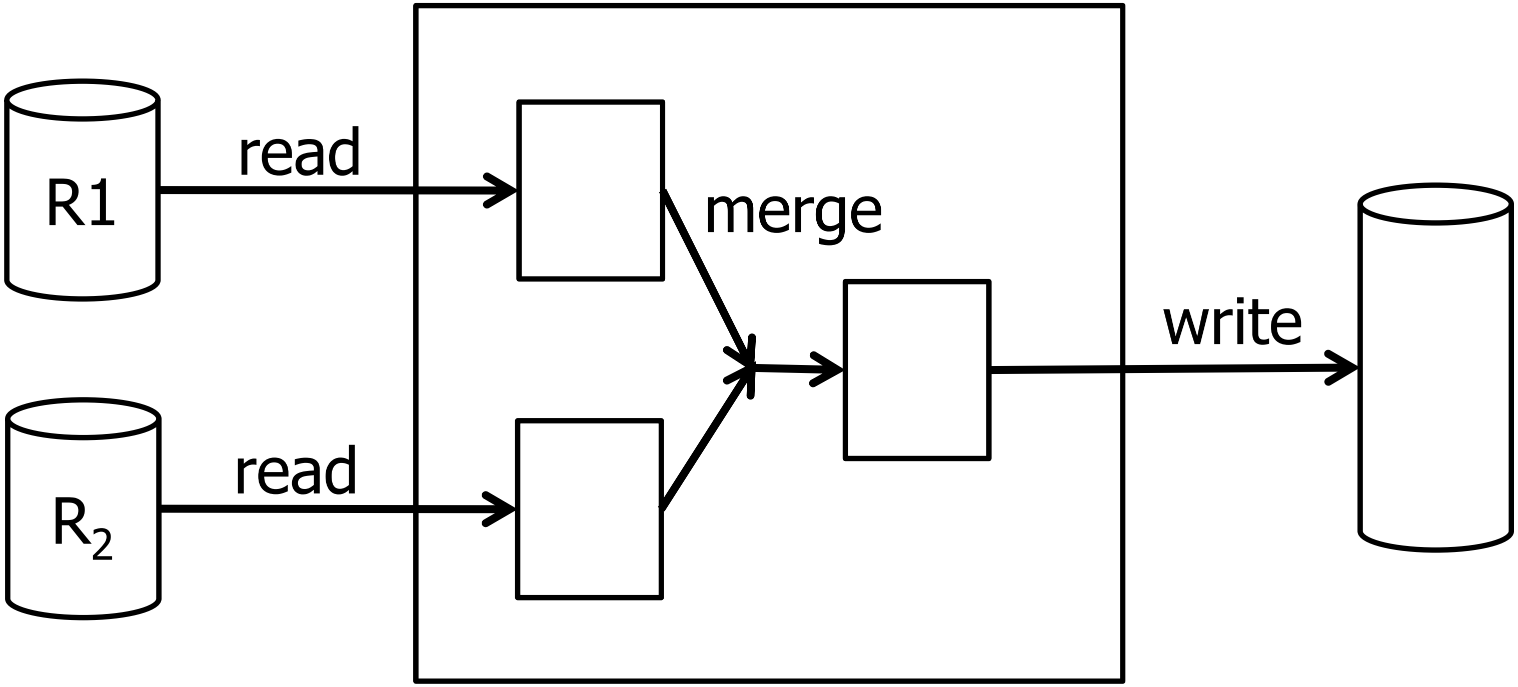
Merging Runs

- Need to create bigger sorted runs out of sorted smaller runs
  - – Divide and Conquer
  - – Merge Sort?
- How to merge two runs that are bigger than **M?**

Merging Runs using 3 pages

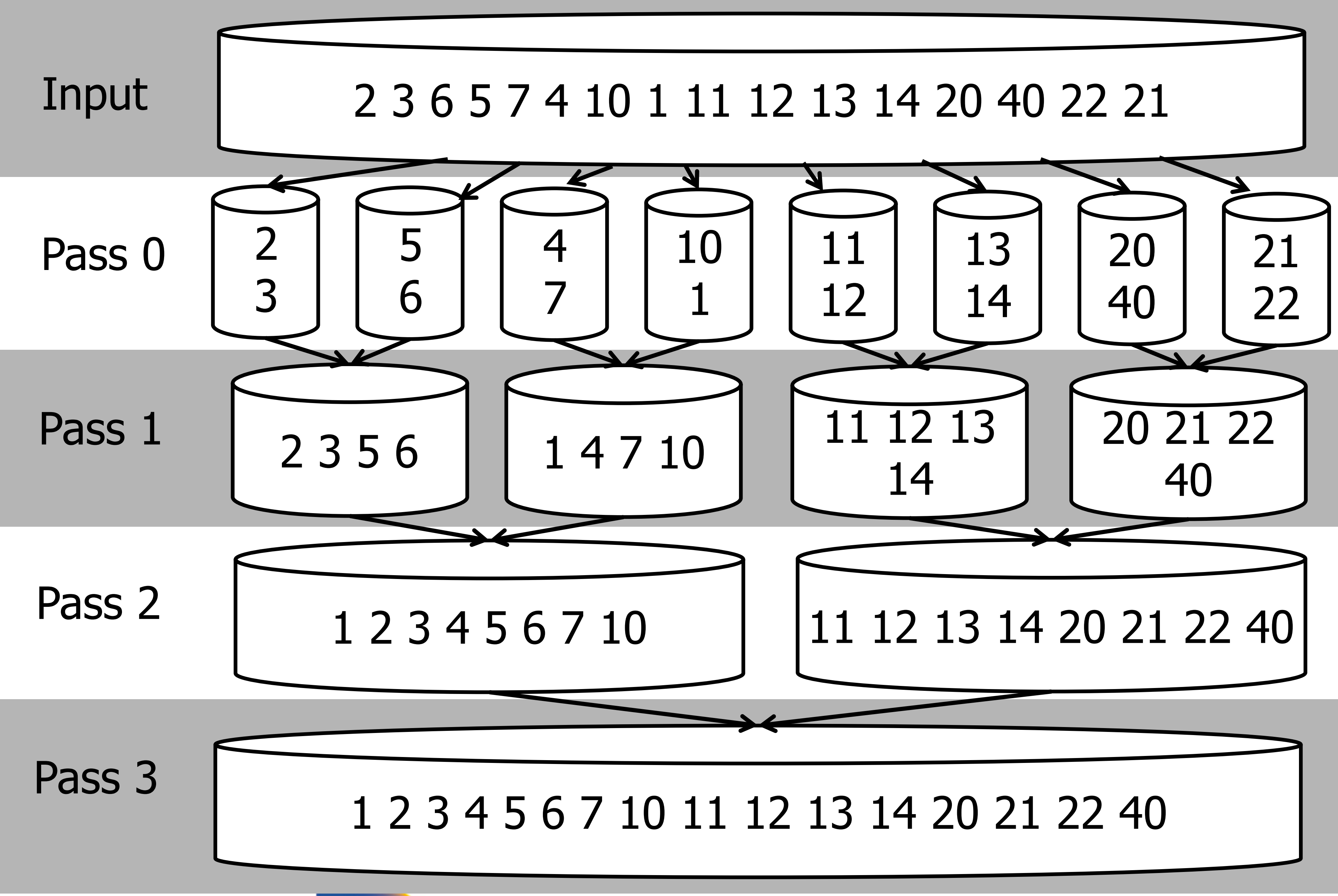
- Merging runs R_1 and R_2
- Need 3 pages
  - One page to buffer pages from R_1
  - One page to buffer pages from R_2
  - One page to buffer the result
    - Whenever this buffer is full, write it to disk

Merging Runs



2-Way External Mergesort

- Repeat process until we have one sorted run
- Each iteration (pass) reads and writes the whole table once: **2 B(R)** I/Os
- Each pass doubles the run size
  - $\mathbf{1 + \lceil \log_2 (B(R) / M) \rceil}$ runs
  - $\mathbf{2\ B(R) * (1 + \lceil \log_2 (B(R) / M) \rceil)}$ I/Os



Input

2 3 6 5 7 4 10 1 11 12 13 14 20 40 22 21

Pass 0

2 3

5 6

4 7

10 1

11 12

13 14

20 40

21 22

Pass 1

2 3 5 6

1 4 7 10

11 12 13 14

20 21 22 40

Pass 2

1 2 3 4 5 6 7 10

11 12 13 14 20 21 22 40

Pass 3

1 2 3 4 5 6 7 10 11 12 13 14 20 21 22 40

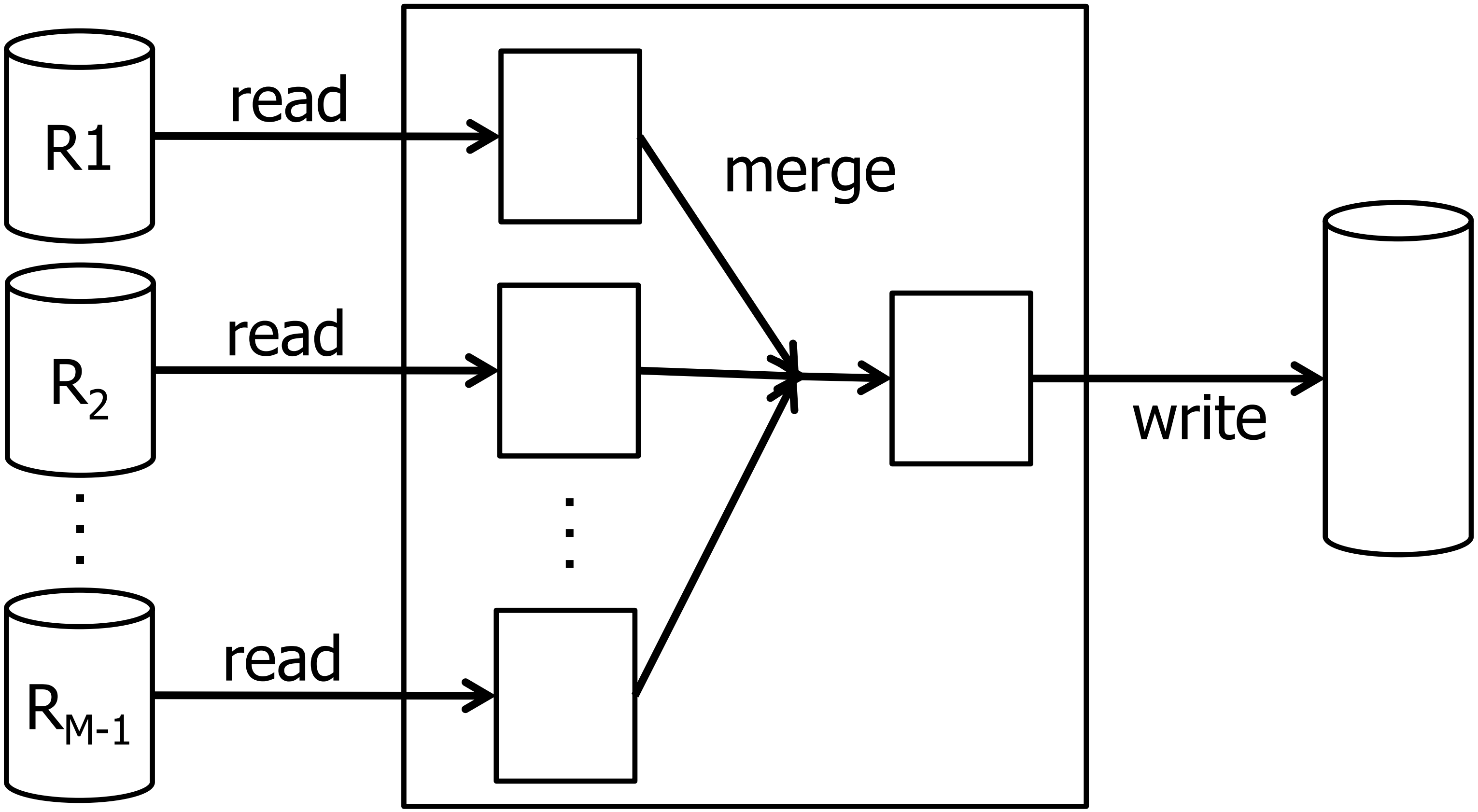
N-Way External Mergesort

- How to utilize **M** buffer during merging?
- Each pass merges **M-1** runs at once
  - One memory page as buffer for each run
- #I/Os

$\mathbf{1 + \lceil \log_{M-1} (B(R) / M) \rceil}$ runs

$\mathbf{2\, B(R) * (1 + \lceil \log_{M-1} (B(R) / M) \rceil)}$ I/Os

Merging Runs



How many passes do we need?

N	M=17	M=129	M=257	M=513	M=1025
100	2	1	1	1	1
1,000	3	2	2	2	1
10,000	4	2	2	2	2
100,000	5	3	3	2	2
1,000,000	5	3	3	3	2
10,000,000	6	4	3	3	3
100,000,000	7	4	4	3	3
1,000,000,000	8	5	4	4	3

To put into perspective

- Scenario
  - Page size 4KB
  - 1TB of data (250,000,000)
  - 10MB of buffer for sorting (250)
- Passes
  - 4 passes

Merge

- In practice would want larger I/O buffer for each run
- Trade-off between number of runs and efficiency of I/O

Improving in-memory merging

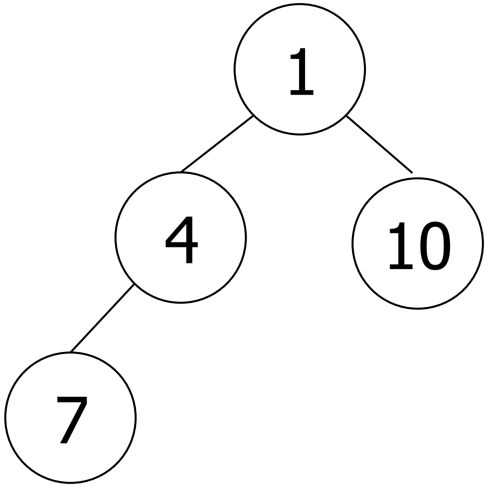
- Merging **M** runs
  - To choose next element to output
  - Have to compare **M** elements
  - -> complexity linear in **M**: **O(M)**
- How to improve that?
  - Use priority queue to store current element from each run
  - -> **$O(\log_2(M))$**

Priority Queue

- Queue for accessing elements in some given order
  - **`pop-smallest`** = return and remove smallest element in set
  - **`Insert(e)`** = insert element into queue

Min-Heap

- Implementation of priority queue
  - Store elements in a binary tree
  - All levels are full (except leaf level)
  - Heap property
    - Parent is smaller than child
- Example: { 1, 4, 7, 10 }



Min-Heap Insertion

- **`insert(e)`**
  1. Add element at next free leaf node
     - This may invalidate heap property
  2. If node smaller than parent then
     - Switch node with parent
  3. Repeat until 2) cannot be applied anymore

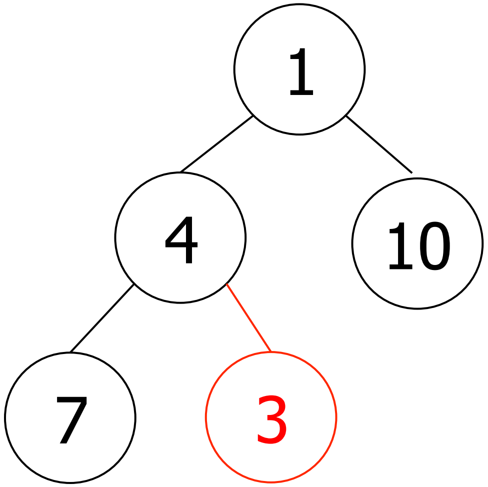
Min-Heap Dequeue

- **`pop–smallest`**
  1. Return Root and use right-most leaf as new root
     - This may invalidate heap property
  2. If node smaller than child then
     - Switch node with smaller child
  3. Repeat until 2) cannot be applied anymore

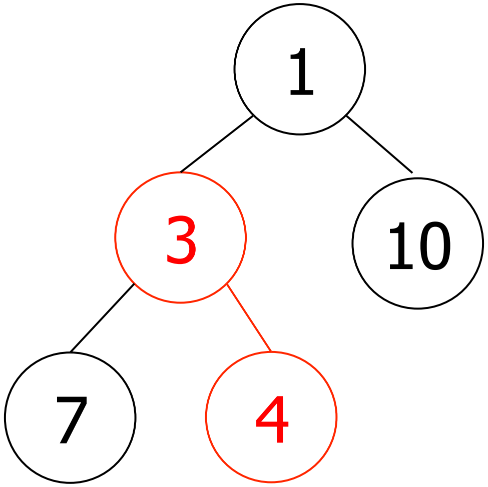
Insertion

- Insert 3

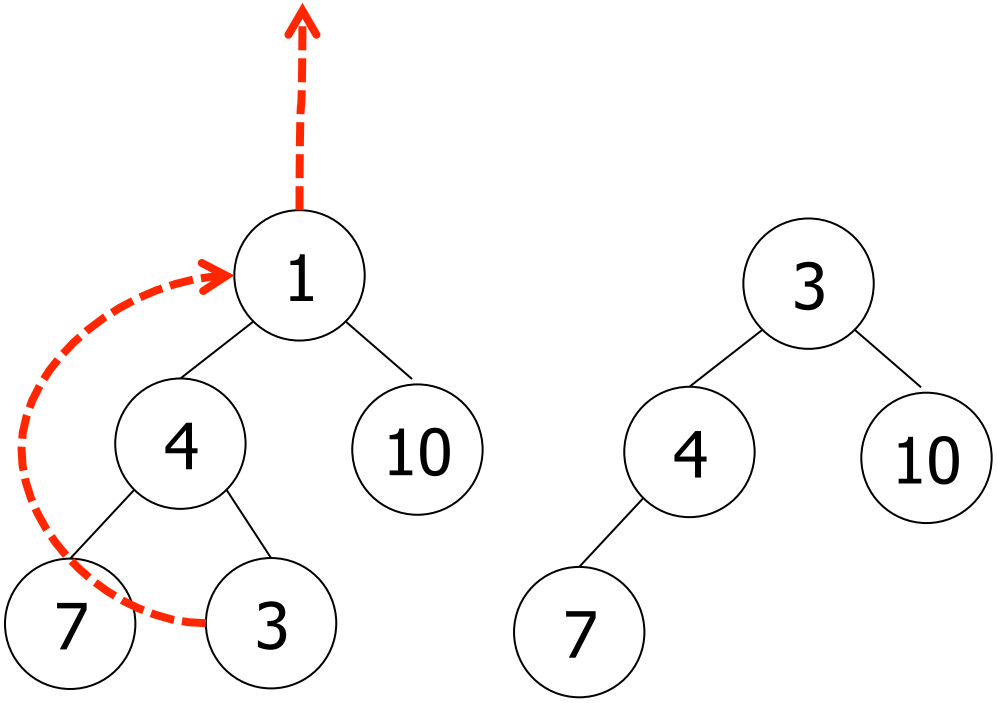
Insert at first free position



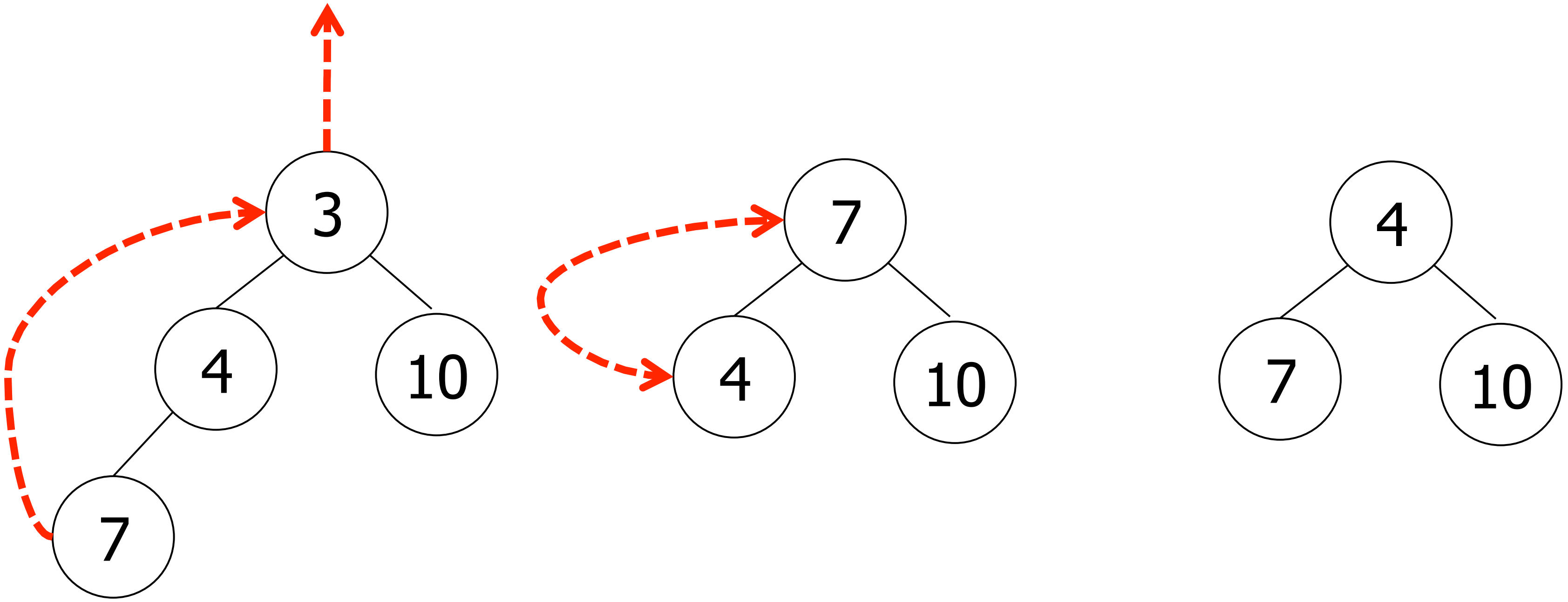
Restore heap property



Dequeue



Dequeue

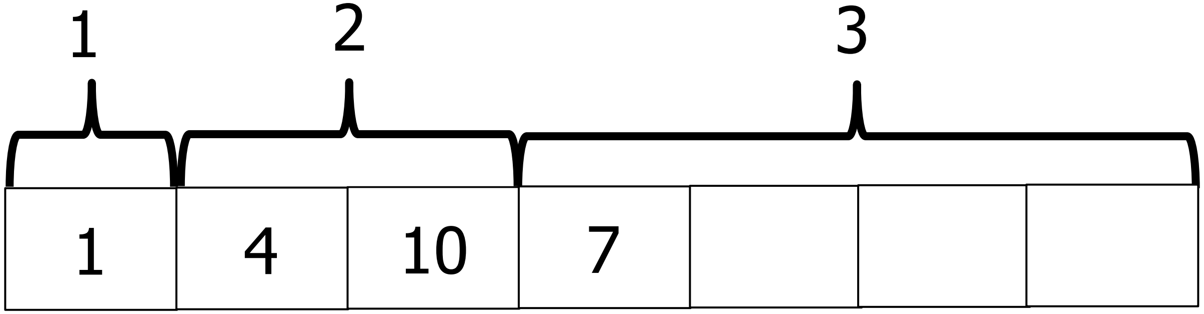


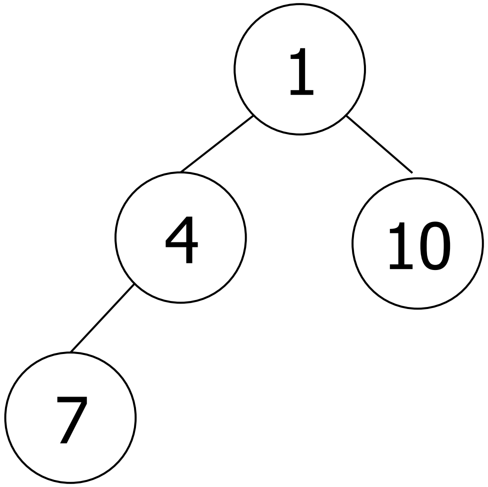
Min/Max-Heap Complexity

- Head is a complete tree
  - Height is $O(\log_2(n))$
- Insertion
  - Maximal height of the tree switches
  - -> $O(\log_2(n))$
- Dequeue
  - Maximal height of the tree switches
  - -> $O(\log_2(n))$

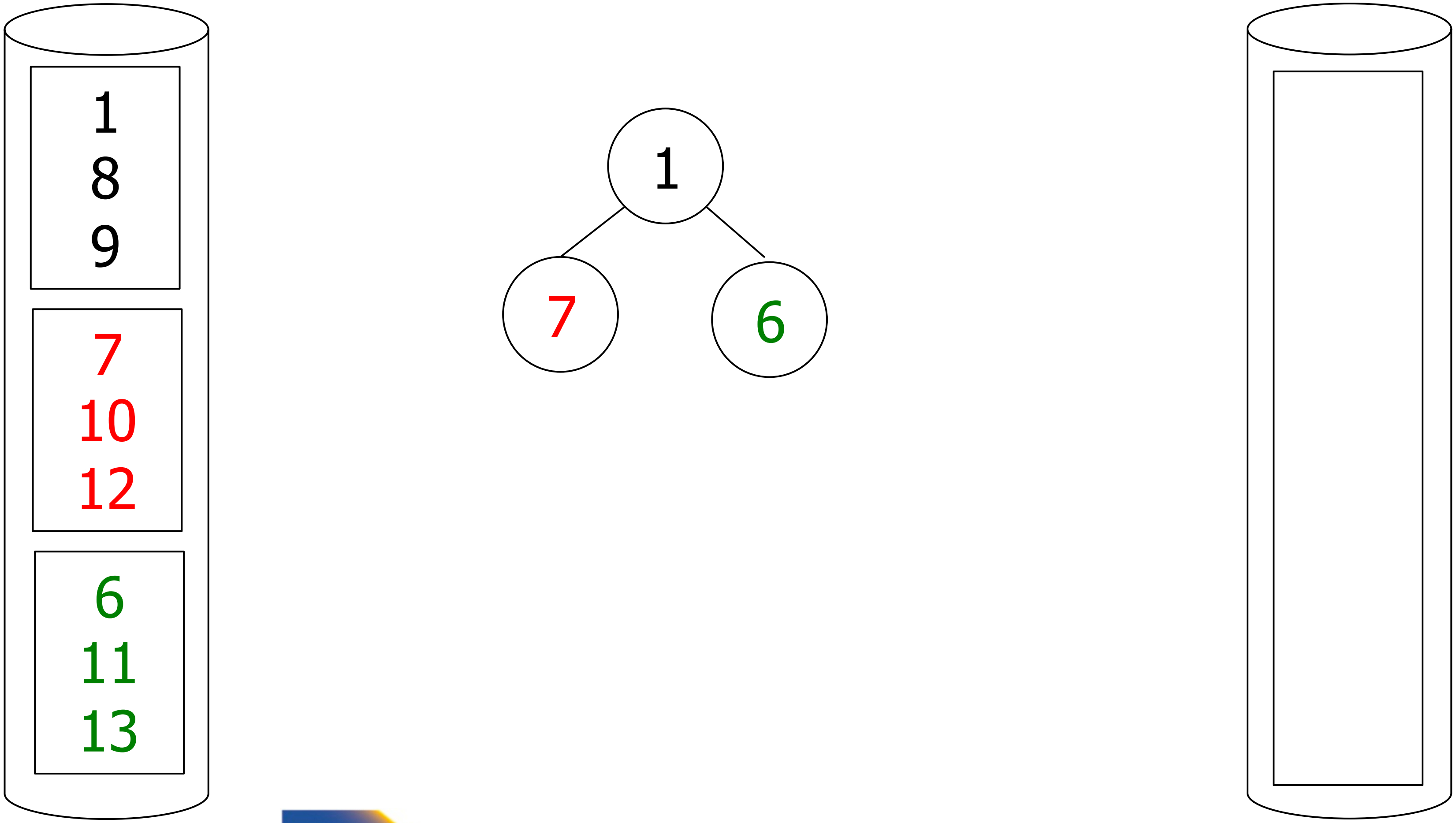
Min-Heap Implementation

- Full tree
  - Use array to implement tree
- Compute positions
  - Parent(n) = $\lfloor (n-1) / 2 \rfloor$
  - Children(n) = 2n + 1, 2n + 2

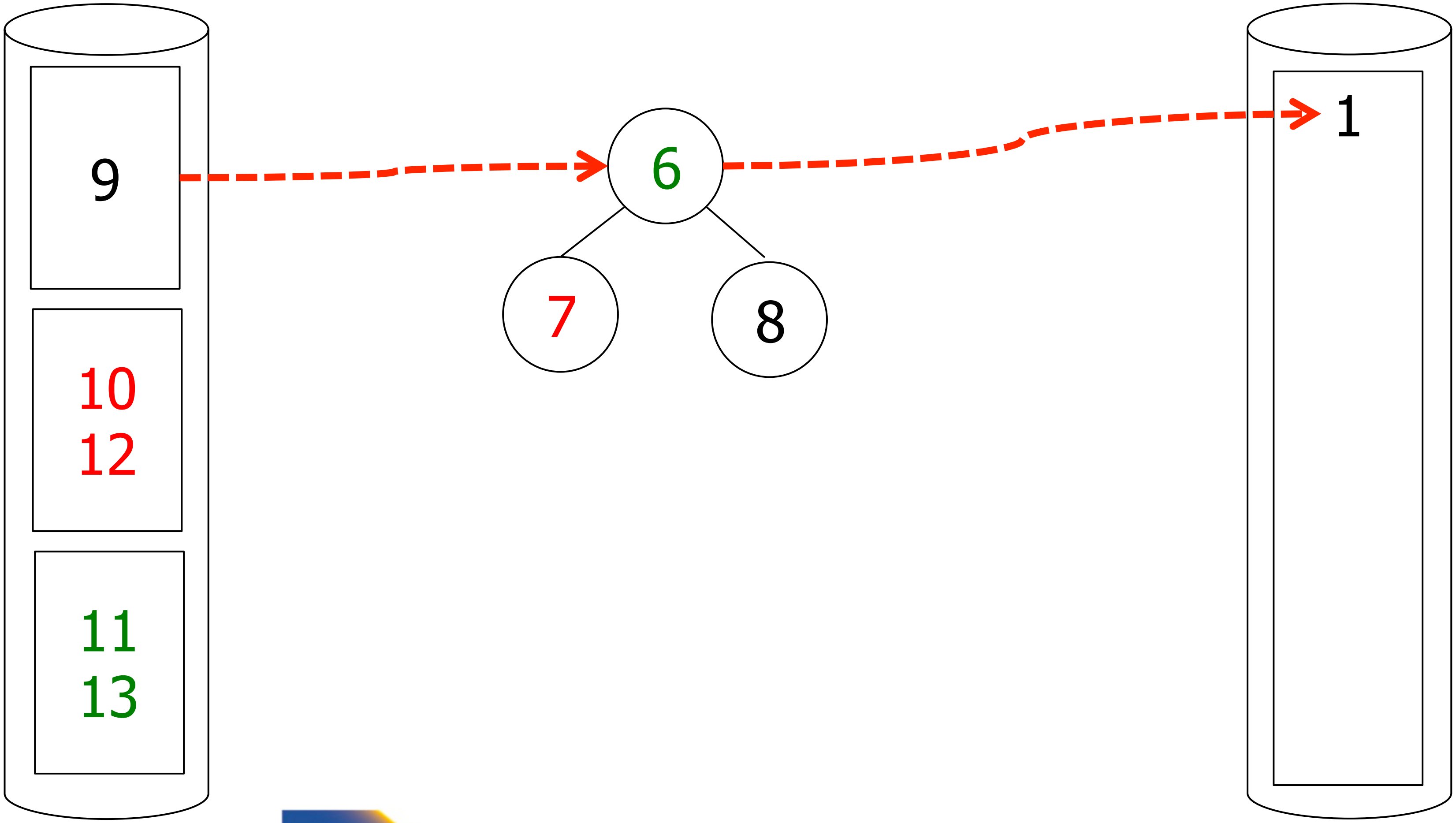




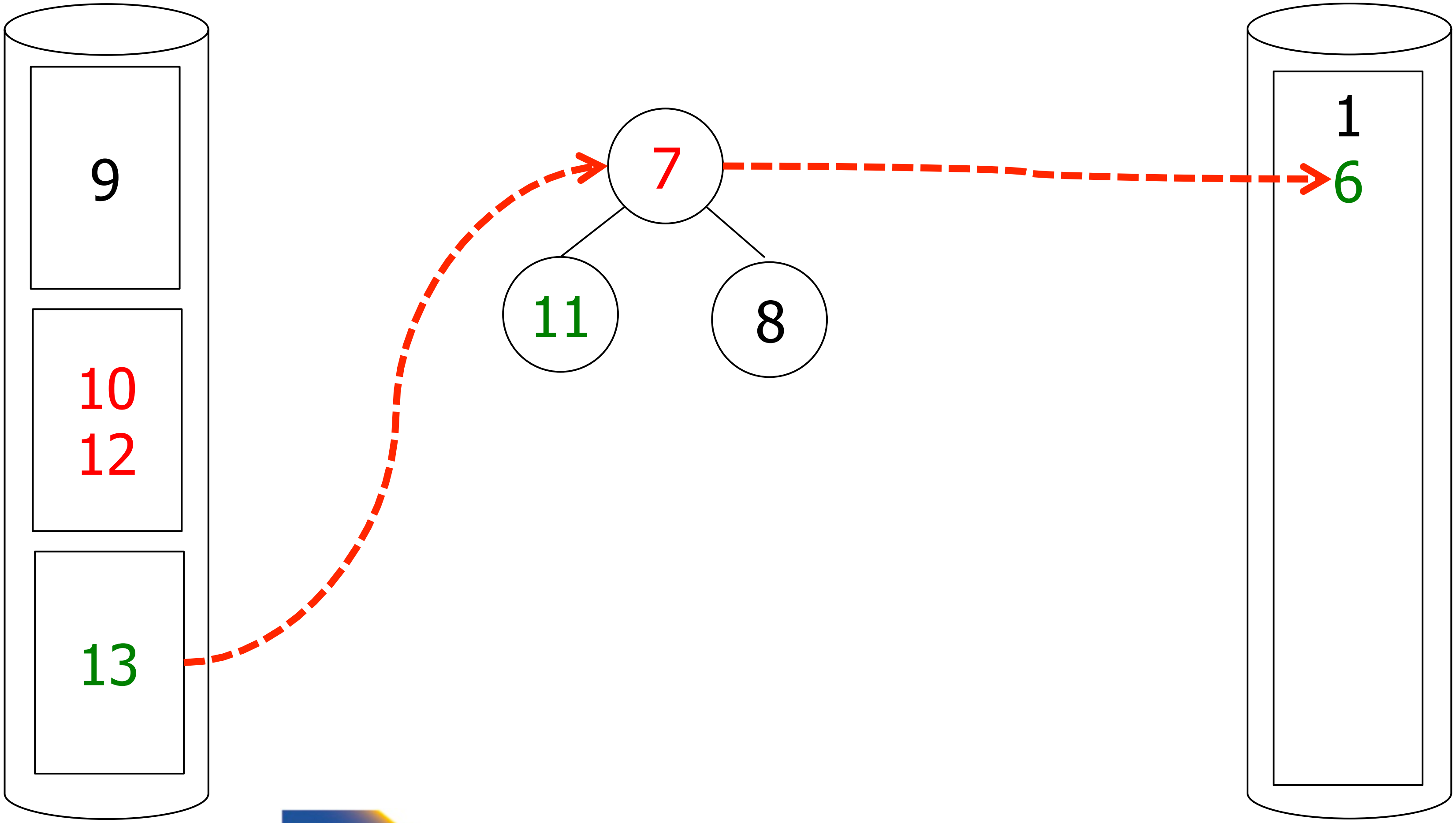
Merging with Priority Queue



Merging with Priority Queue



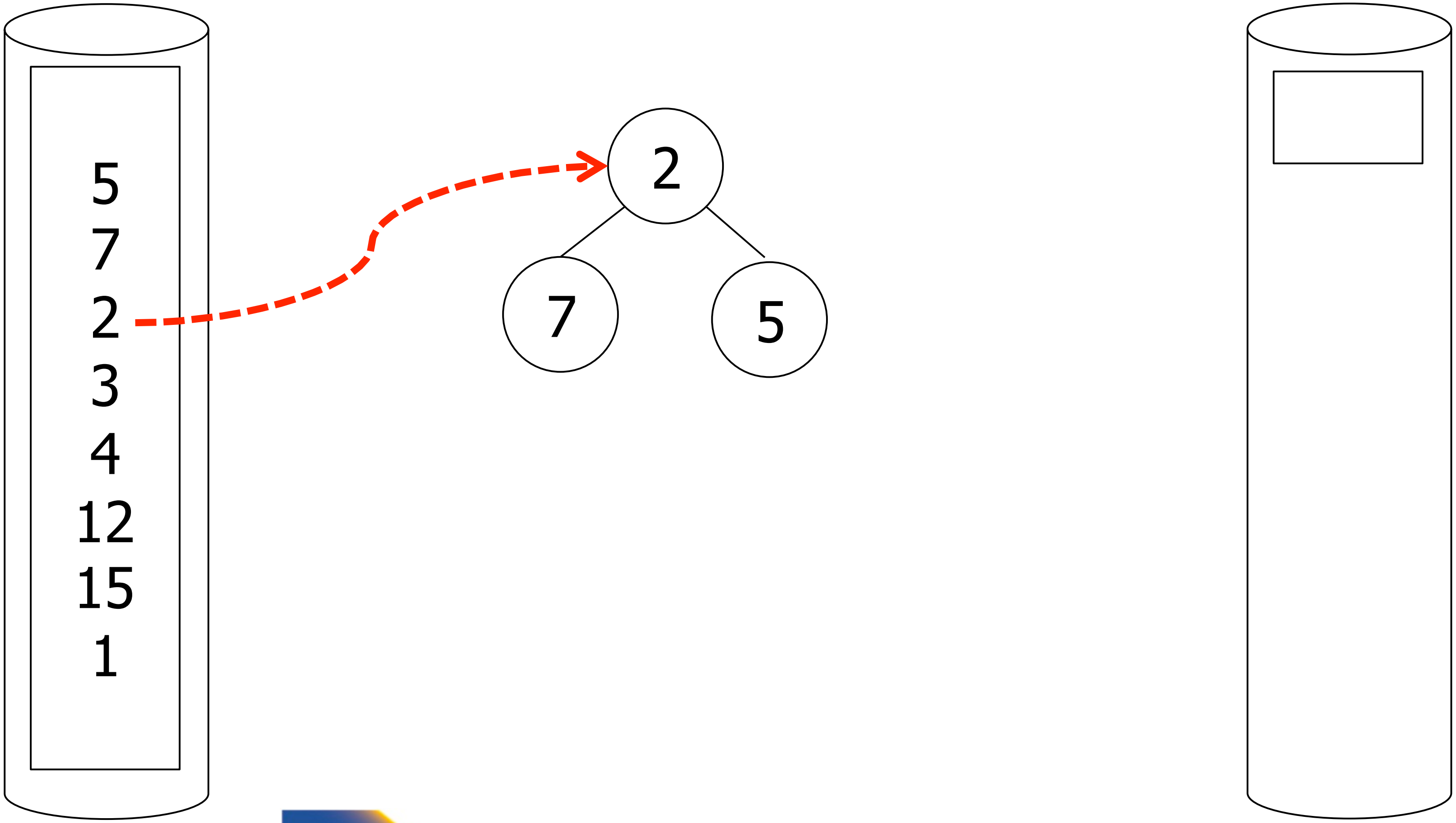
Merging with Priority Queue



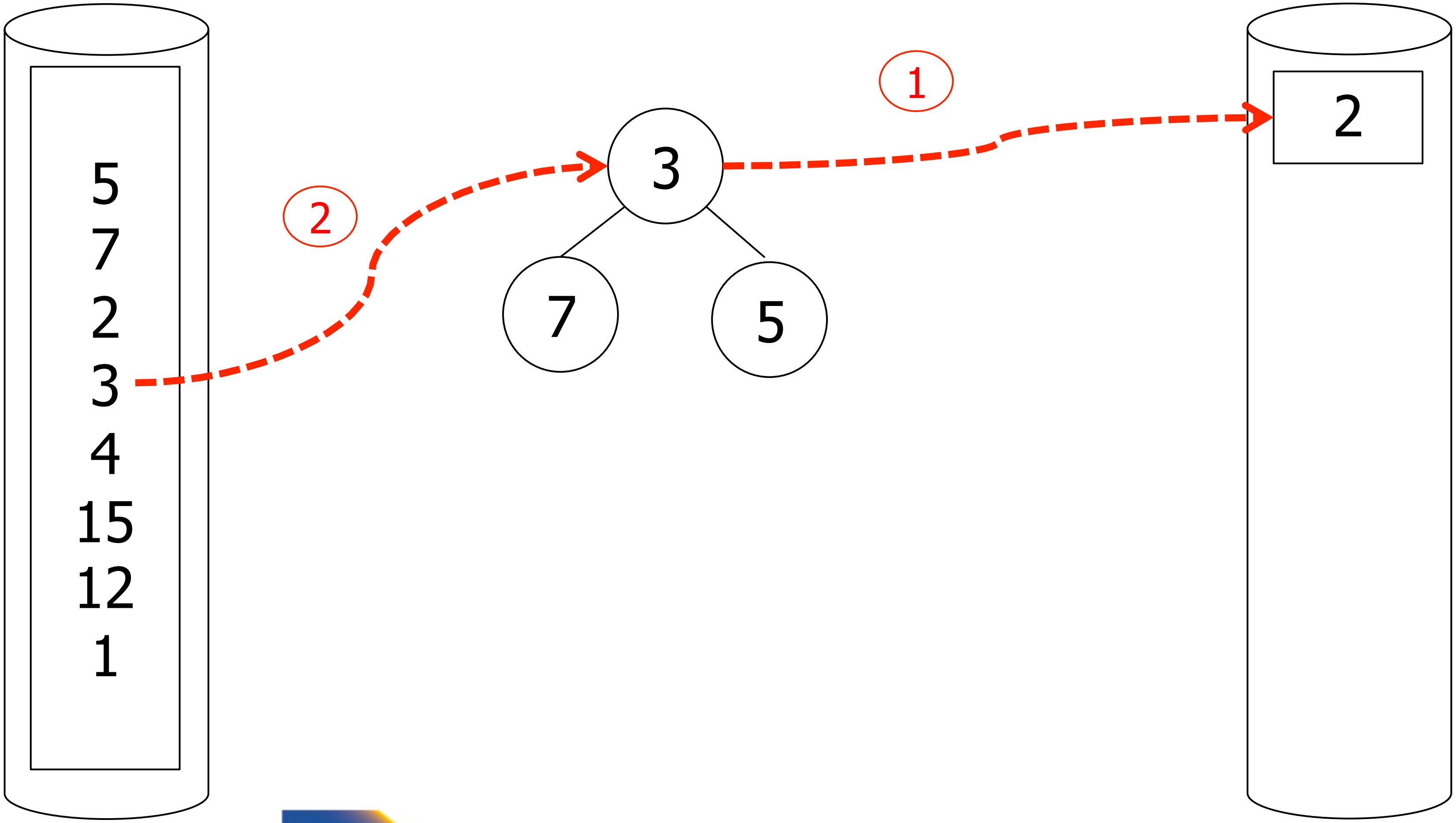
Using a heap to generate runs

- Read inputs into heap
  - Until available memory is full
- Replace elements
  - Remove smallest element from heap
    - If larger then last element written to current run then write to current run
    - Else create a new run
  - Add new element from input to heap

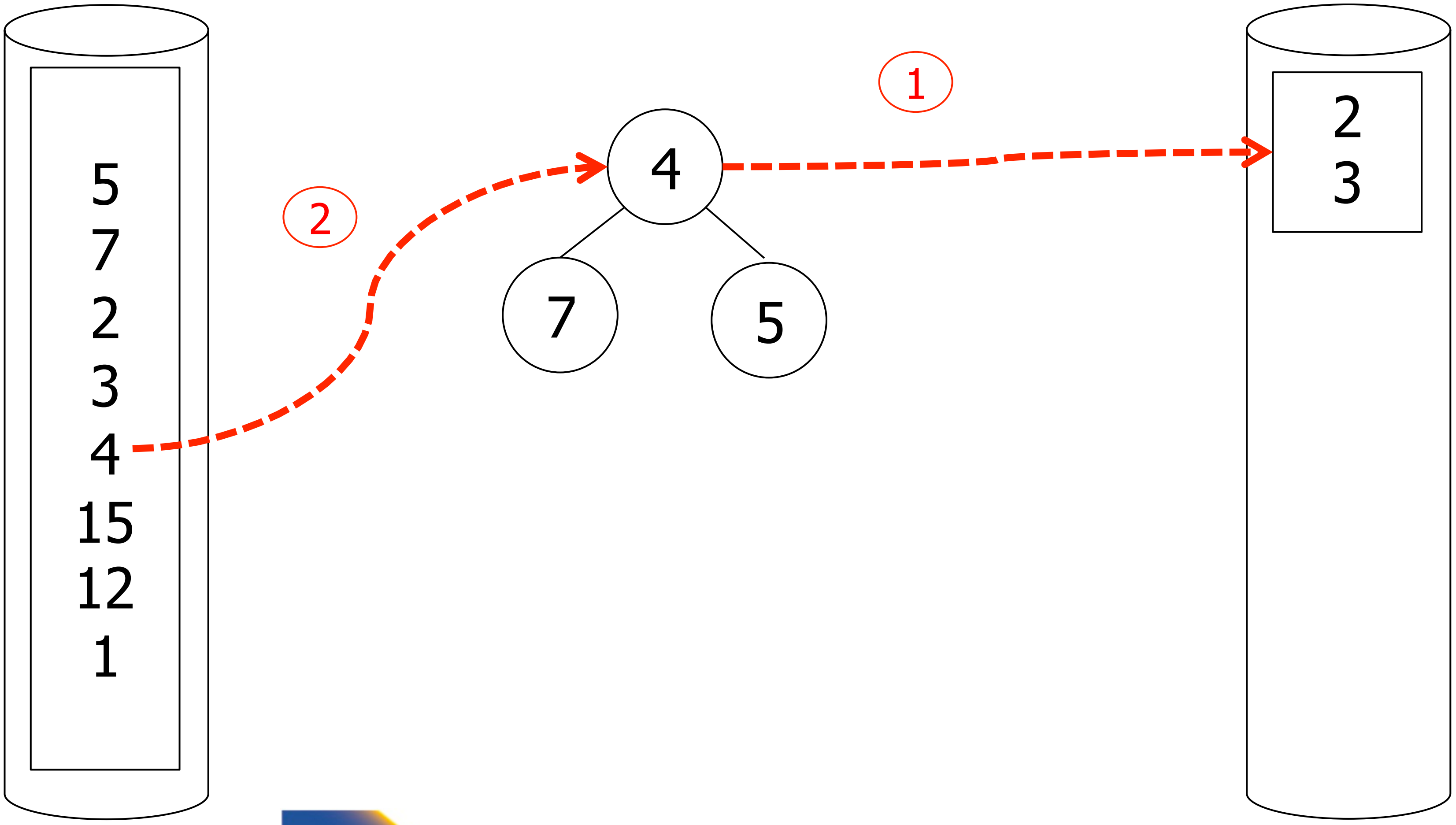
Using a heap to generate runs



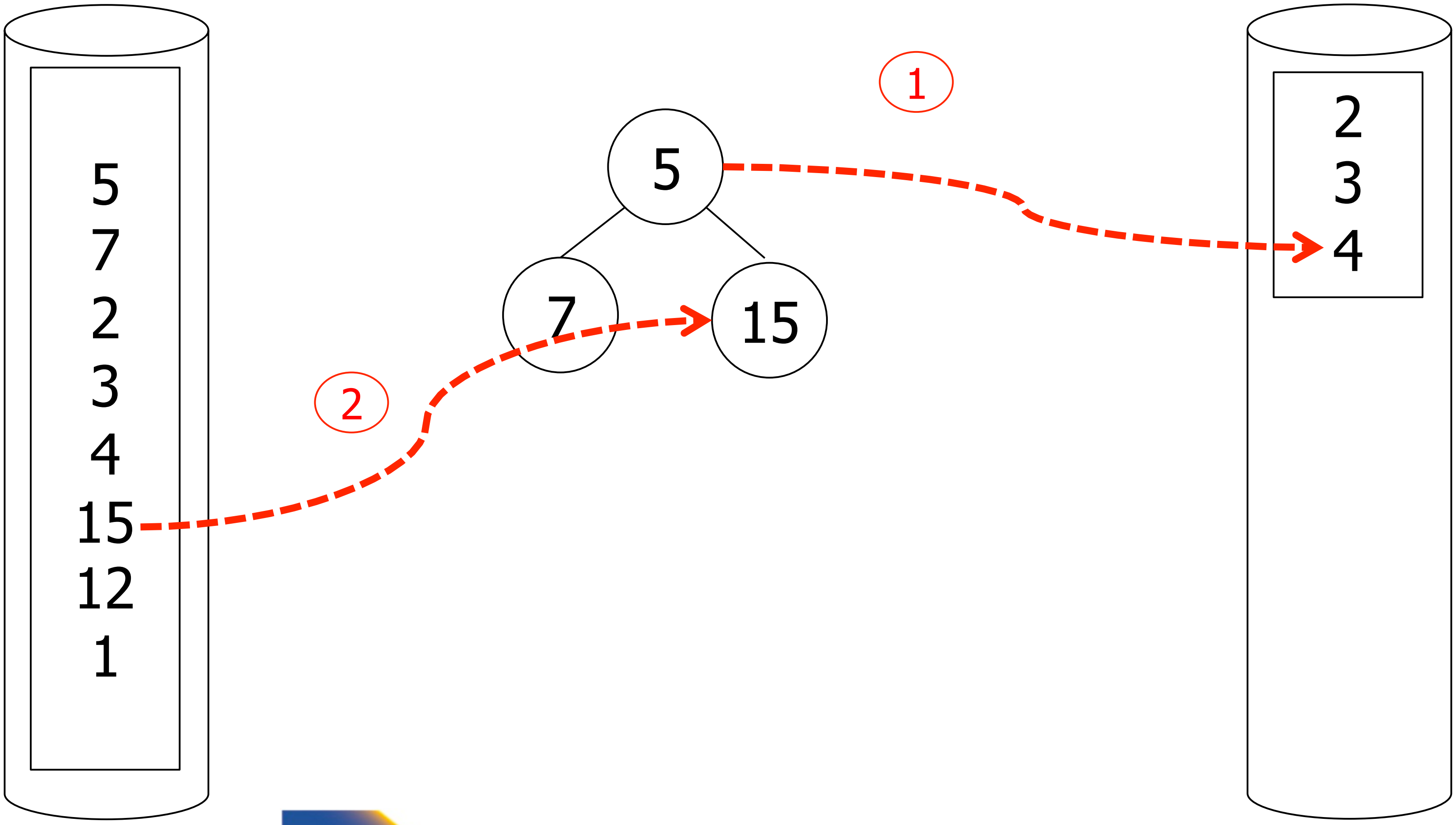
Using a heap to generate runs



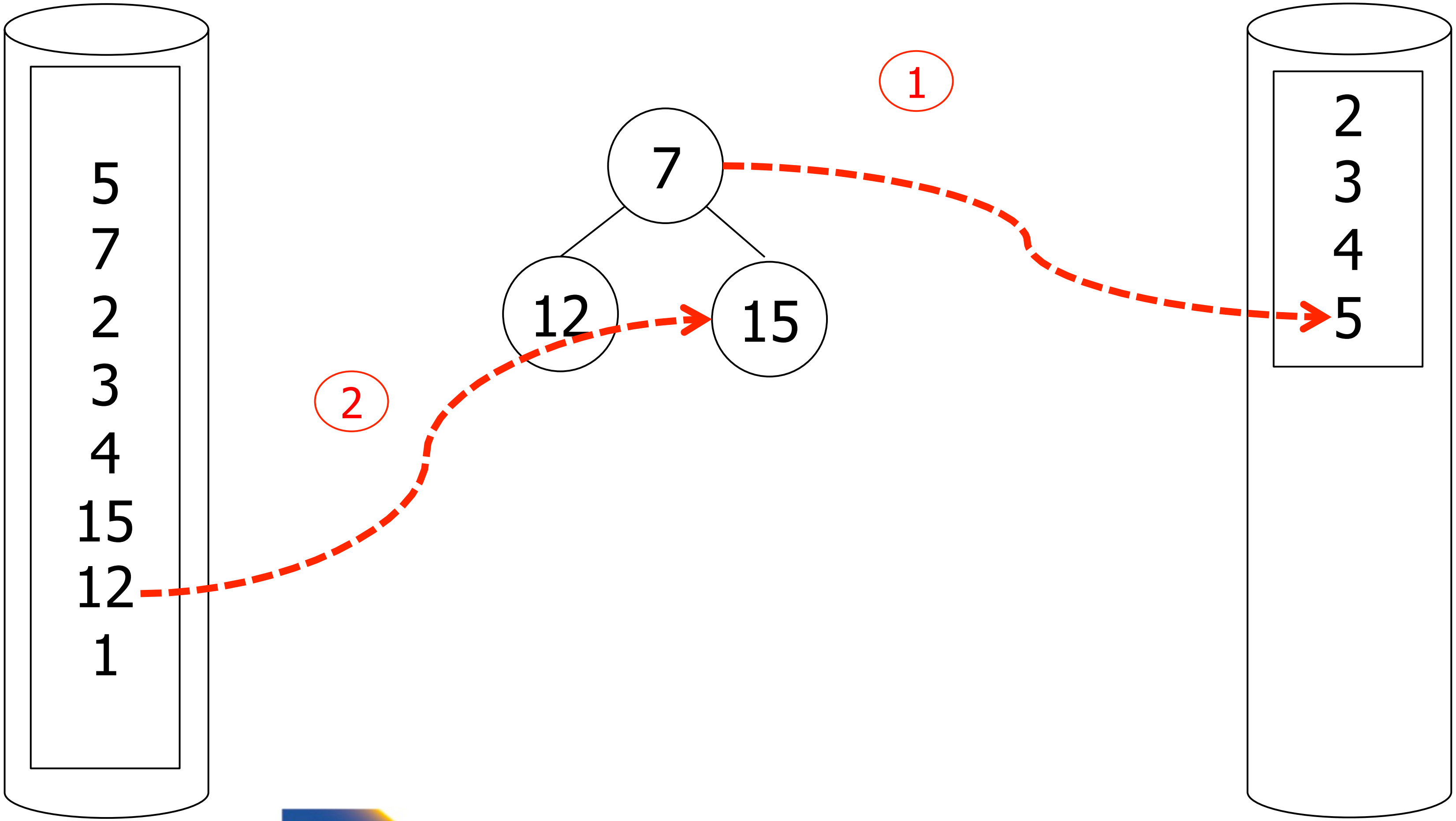
Using a heap to generate runs



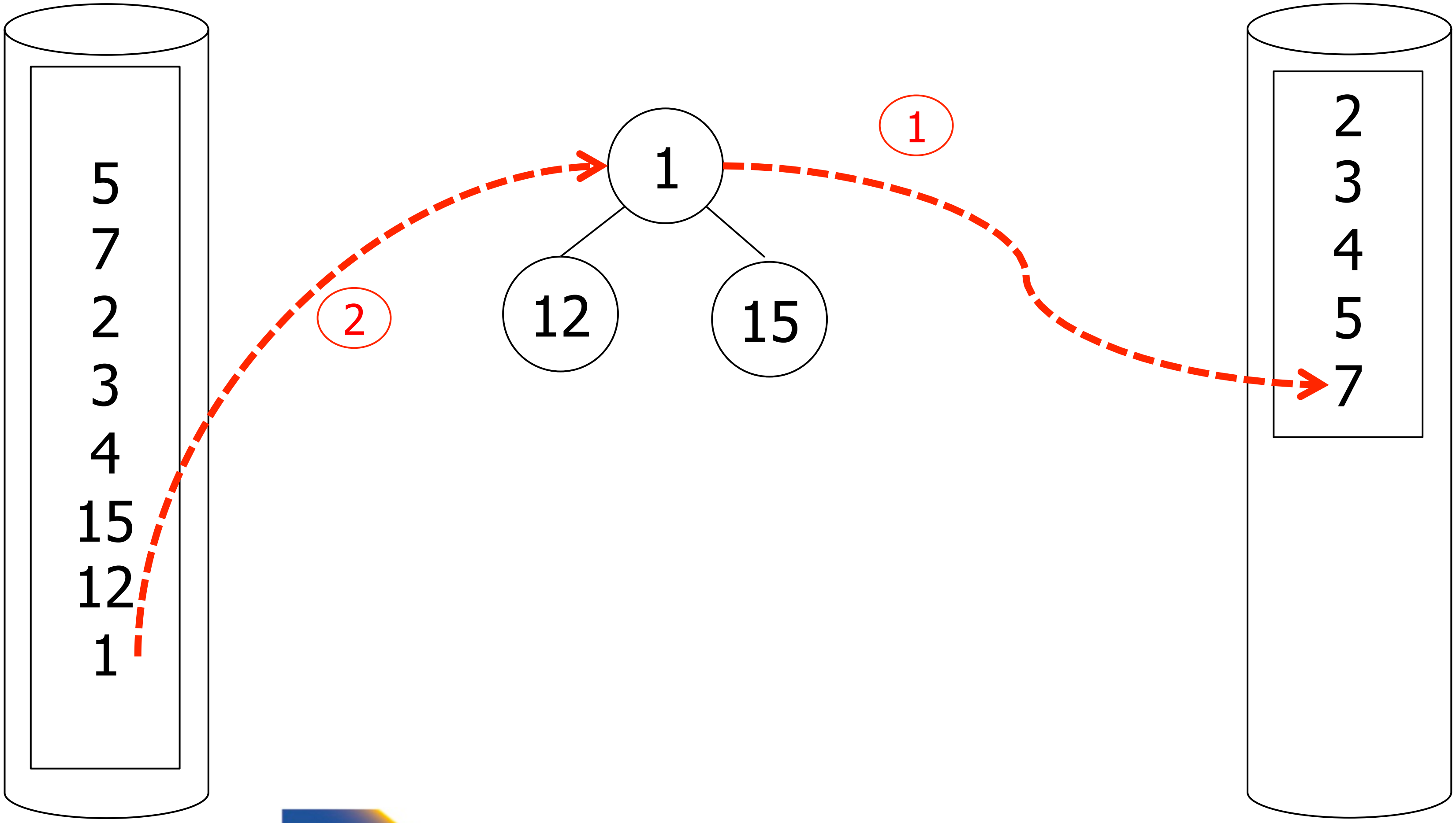
Using a heap to generate runs



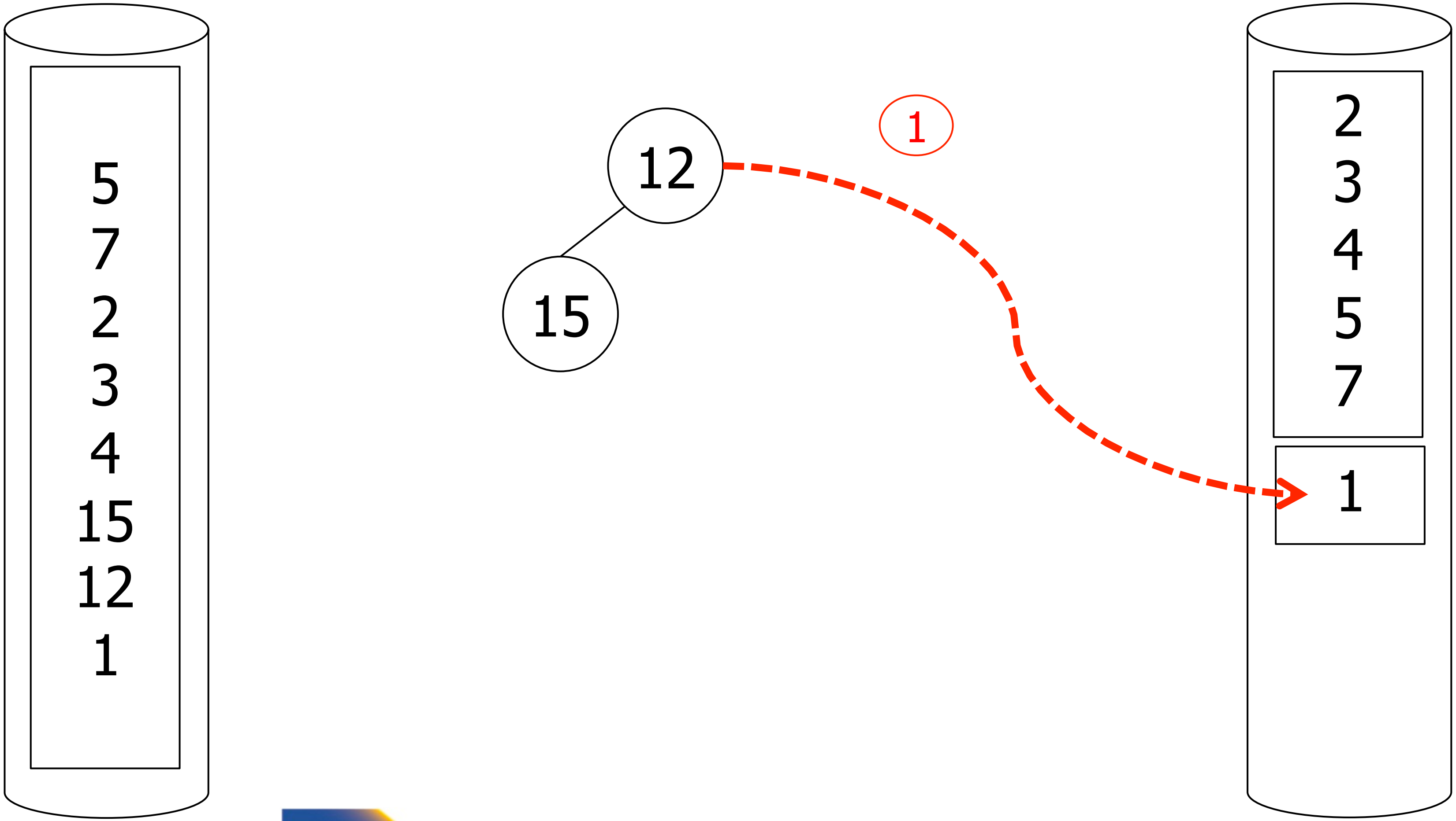
Using a heap to generate runs



Using a heap to generate runs



Using a heap to generate runs



Using a heap to generate runs

- Increases the run-length
    - On average by a factor of 2 (see Knuth)

Use clustered B+-tree

- Keys in the B+-tree **I** are in sort order
  - If B+-tree is clustered traversing the leaf nodes is sequential I/O!
  - **K** = #keys/leaf node
- Approach
  - Traverse from root to first leaf: **HT(I)**
  - Follow sibling pointers: **|R| / K**
  - Read data blocks**: B(R)**

I/O Operations

- **HT(I) + |R| / K + B(R)** I/Os
- Less than **2 B(R)** = 1 pass external mergesort
- ->Better than external merge-sort!

Unclustered B+-tree?

- Each entry in a leaf node may point to different page of relation R
  - For each leaf page we may read up to **K** pages from relation R
  - Random I/O
- In worst-case we have
  - **K ∗ B(R)**
  - **K = 500**
    - **500 ∗ B(R)** = 250 merge passes

Sorting Comparison

B(R) = number of block of R
M = number of available memory blocks
#RB = records per page
HT = height of B+-tree (logarithmic)
K = number of keys per leaf node

Property	**Ext. Mergesort**	**B+ (clustered)**	**B+ (unclustered)**
Runtime	$O(N \log_{M-1}(N))$	O(N)	O(N)
#I/O (random)	$2\,B(R) * (1 + \lceil \log_{M-1}(B(R) / M) \rceil)$	$HT + \lvert R \rvert / K + B(R)$	$HT + \lvert R \rvert / K + K * \#RB$
Memory	M	1 (better HT + X)	1 (better HT + X)
Disk Space	2 B(R)	0	0
Variants	1) Merge with heap 2) Run generation with heap 3) Larger Buffer		

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Scan

- Implements access to a table
  - Combined with selection
  - Probably projection too
- Variants
  - **Sequential**
    - Scan through all tuples of relation
  - **Index**
    - Use index to find tuples that match selection

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Options

- Transformations: $R_1 \bowtie_c R_2$, $R_2 \bowtie_c R_1$
- Joint algorithms:
  - Nested loop
  - Merge join
  - Join with index
  - Hash join
- Outer join algorithms

Nested Loop Join (conceptually)

```
for each $r \in R_1$ do
    for each $s \in R_2$ do
        if $(r,s) \models C$ then output $(r,s)$
```

Applicable to:

- Any join condition C
- Cross-product

- Merge Join (conceptually)

```
(1) if $R_1$ and $R_2$ not sorted, sort them
(2) i ← 1; j ← 1;
    While $(i \leq T(R_1)) \wedge (j \leq T(R_2))$ do
        if $R_1\{ i \}.C = R_2\{ j \}.C$ then outputTuples
        else if $R_1\{ i \}.C > R_2\{ j \}.C$ then j ← j+1
        else if $R_1\{ i \}.C < R_2\{ j \}.C$ then i ← i+1
```

Applicable to:

- C is conjunction of equalities or </>:

$A_1 = B_1$ AND … AND $A_n = B_n$

Procedure Output-Tuples

```
While $(R_1\{ i \}.C = R_2\{ j \}.C) \wedge (i \leq T(R_1))$ do
    [jj ← j;
     while $(R_1\{ i \}.C = R_2\{ jj \}.C) \wedge (jj \leq T(R_2))$ do
            [output pair $R_1\{ i \}, R_2\{ jj \}$;
             jj ← jj+1  ]
     i ← i+1  ]
```

Example

i	$R_1\{i\}.C$	$R_2\{j\}.C$	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

Index nested loop (Conceptually)

Assume R_2.C index

```
For each $r \in R_1$ do
    [ X ← index ($R_2$, C, r.C)
        for each $s \in X$ do
            output (r,s) pair]
```

Note: X ← index(rel, attr, value)

then X = set of rel tuples with attr = value

Hash join (conceptual)

Hash function h, range 0 → k

Buckets for R_1: G_0, G_1, ... G_k

Buckets for R_2: H_0, H_1, ... H_k

Applicable to:

- C is conjunction of equalities

$$A_1 = B_1 \text{ AND } \dots \text{ AND } A_n = B_n$$

Hash join (conceptual)

Hash function h, range 0 → k

Buckets for R_1: $G_0, G_1, \ldots G_k$

Buckets for R_2: $H_0, H_1, \ldots H_k$

Algorithm

```
(1) Hash $R_1$ tuples into G buckets
(2) Hash $R_2$ tuples into H buckets
(3) For $i = 0$ to k do
        match tuples in $G_i$, $H_i$ buckets
```

Simple example hash: even/odd

R_1	R_2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R_1	R_2
Even:	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

Factors that affect performance

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Example 1(a) Iteration Join $R_1 \bowtie R_2$

- Relations not contiguous
- Recall $\left\{ \begin{array}{l} T(R_1) = 10{,}000 \quad T(R_2) = 5{,}000 \\ S(R_1) = S(R_2) = 1/10 \text{ block} \\ MEM=101 \text{ blocks} \end{array} \right.$

Example 1(a)

Nested Loop Join $R_1 \bowtie R_2$

- Relations not contiguous
- Recall $\begin{cases} T(R_1) = 10{,}000 \qquad T(R_2) = 5{,}000 \\ S(R_1) = S(R_2) = 1/10 \text{ block} \\ MEM = 101 \text{ blocks} \end{cases}$

Cost: for each R_1 tuple:

[Read tuple + Read R_2]

Total $= 10{,}000\ [1 + 500] = 5{,}010{,}000$ IOs

- Can we do better?

- Can we do better?

Use our memory

(1) Read 100 blocks of R_1

(2) Read all of R_2 (using 1 block) + join

(3) Repeat until done

Cost: for each R_1 chunk:

Read chunk: 100 IOs

Read R_2: 500 IOs

600

Cost: for each R_1 chunk:

Read chunk: 100 IOs

Read R_2: $\underline{500\ }$ IOs

600

$$\text{Total} = \frac{1{,}000}{100} \text{ x } 600 = 6{,}000 \text{ IOs}$$

- Can we do better?

- Can we do better?

☛ Reverse join order: $R_2 \bowtie R_1$

$$\text{Total} = \frac{500}{100} \times (100 + 1{,}000) =$$

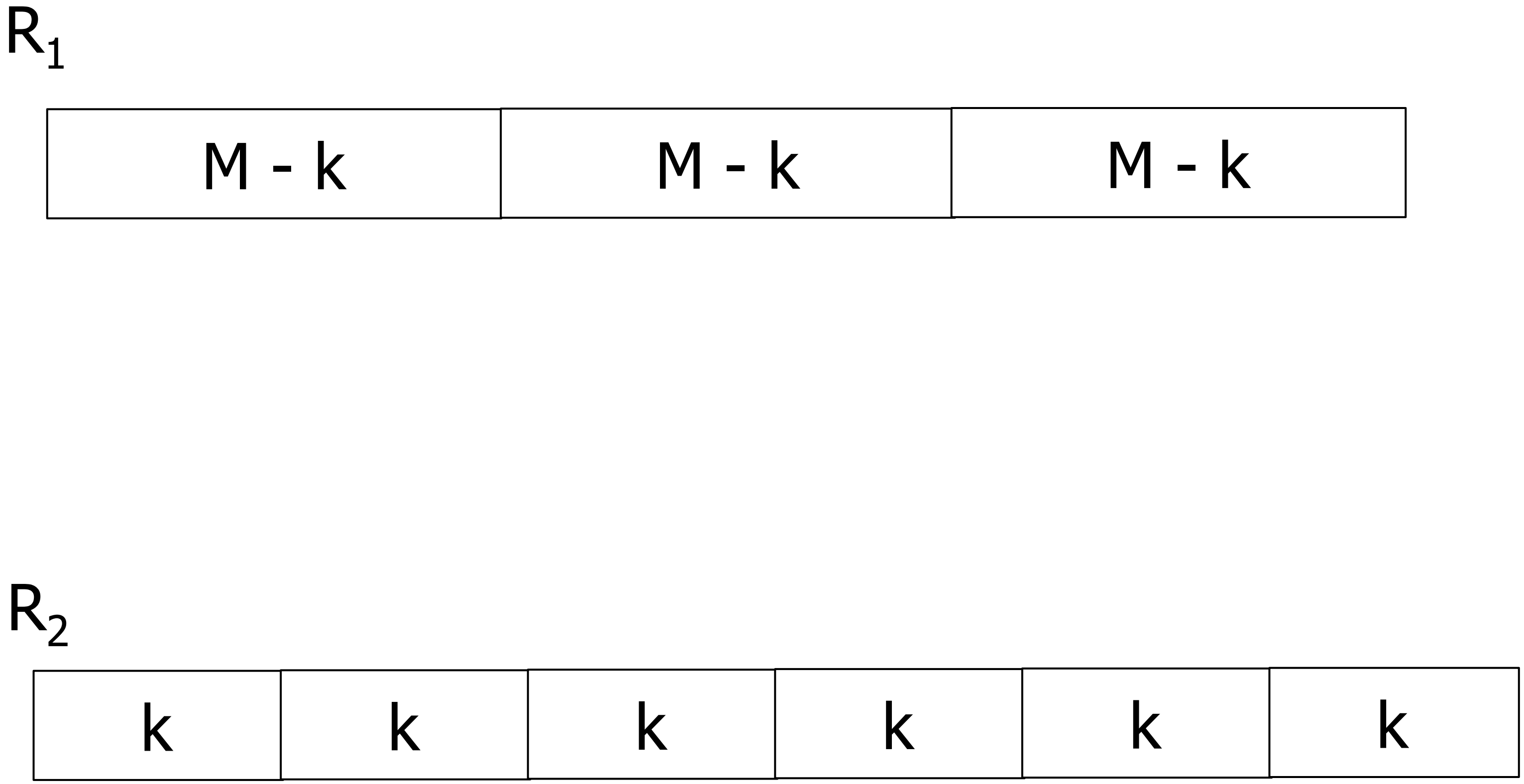
$$5 \times 1{,}100 = 5{,}500 \text{ IOs}$$

Block-Nested Loop Join (conceptual)

```
for each M-1 blocks of $R_1$ do
    read M-1 blocks of $R_1$ into buffer
    for each block of $R_2$ do
        read next block of $R_2$
        for each tuple r in $R_1$ block
            for each tuple s in $R_2$ block
            if $(r,s) \models C$ then output $(r,s)$
```

Note

- How much memory for buffering inner and for outer chunks?
  - – 1 for inner would minimize I/O
  - – But, larger buffer better for I/O



R_1

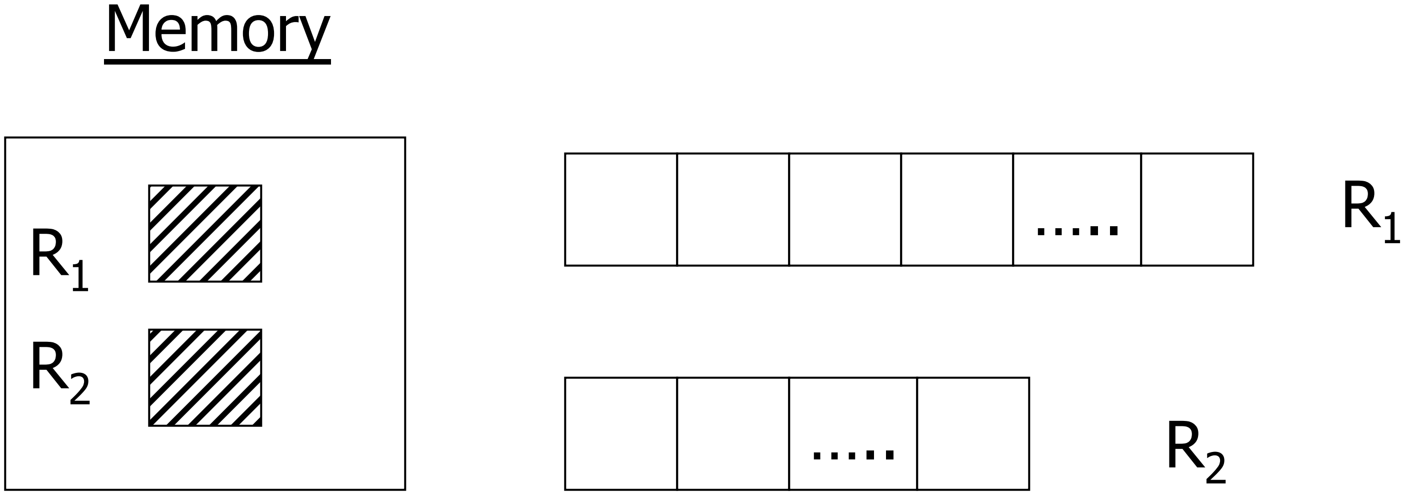
M - k | M - k | M - k

R_2

k | k | k | k | k | k

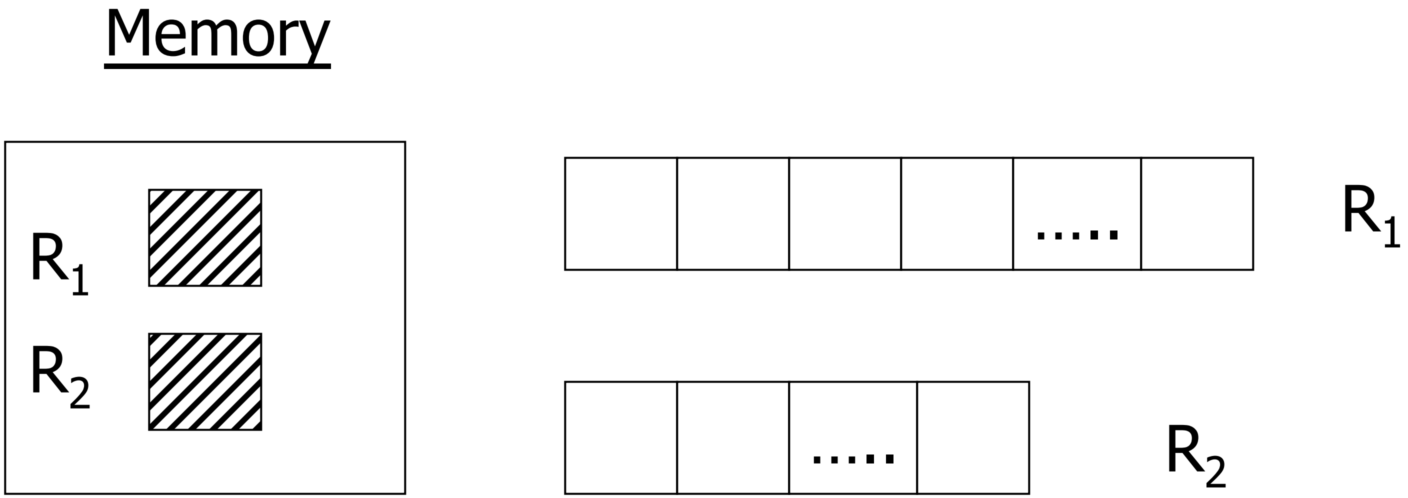
Example 1(b) Merge Join

- Both R_1, R_2 ordered by C; relations contiguous



Example 1(b) Merge Join

- Both R_1, R_2 ordered by C; relations contiguous



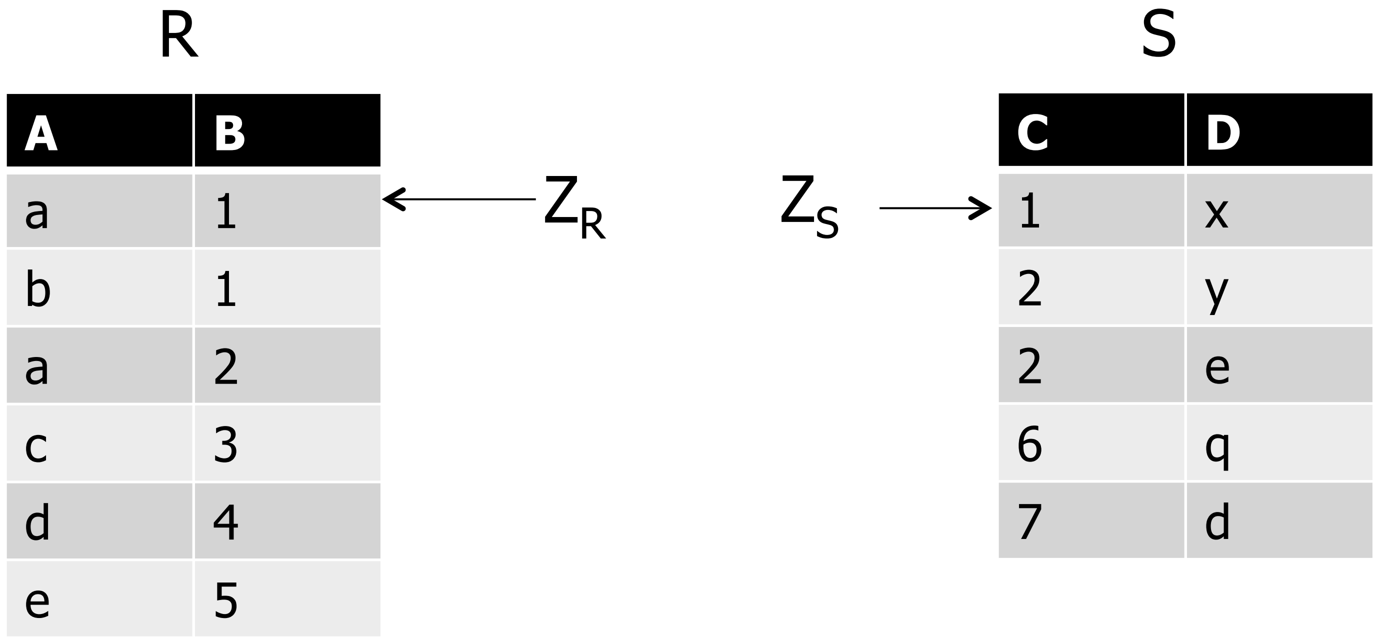
Total cost: Read R_1 cost + read R_2 cost

= 1000 + 500 = 1,500 IOs

Merge Join Example

$R \bowtie_{B=C} S$

Output: (a,1,1,X)



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

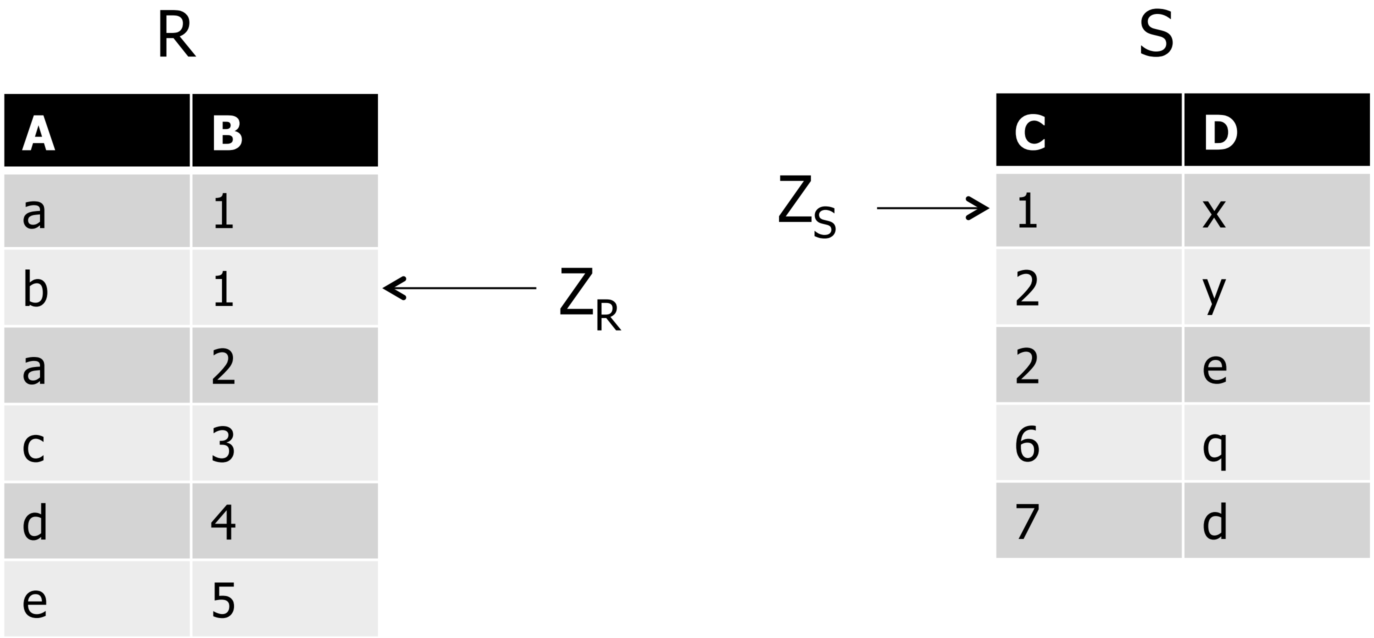
S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Join Example

$R \bowtie_{B=C} S$

Output: (b,1,1,X)



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

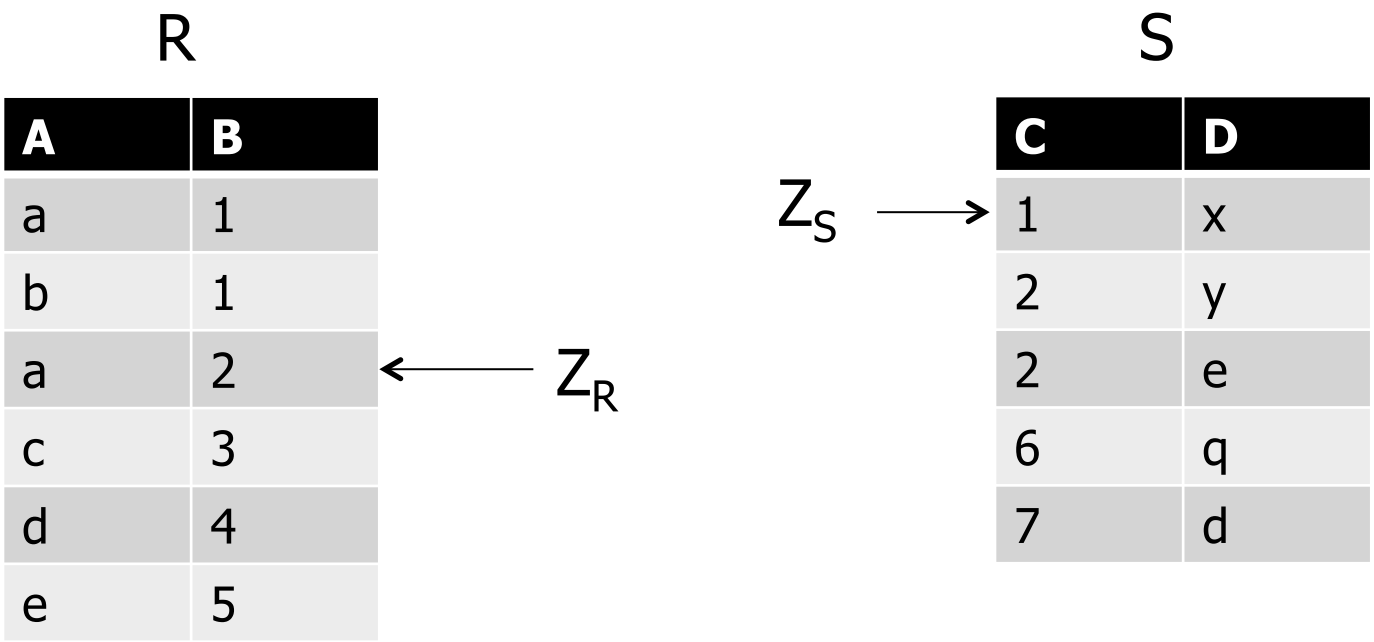
S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Join Example

$R \bowtie_{B=C} S$

R.B > S.C: advance Z_S



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

Z_R

S

C	D
1	x
2	y
2	e
6	q
7	d

Z_S

Merge Join Example

$R \bowtie_{B=C} S$

Output: (a,2,2,y)

R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

$\longleftarrow Z_R$

S

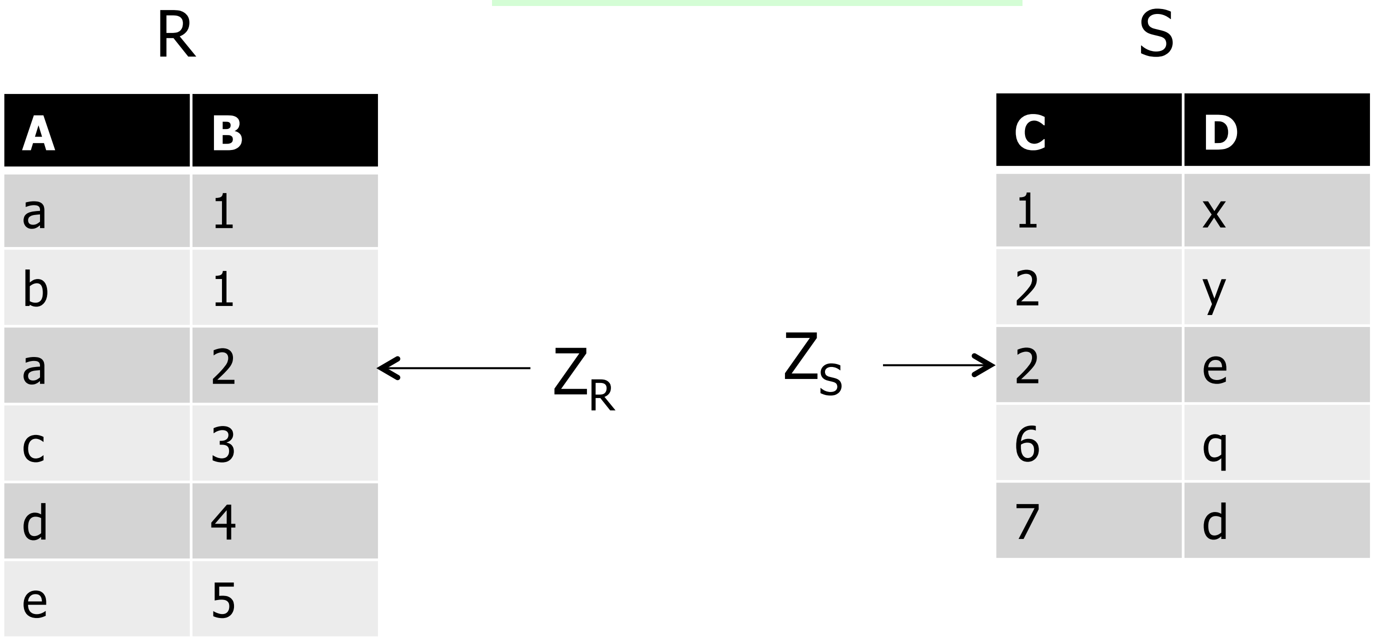
C	D
1	x
2	y
2	e
6	q
7	d

$Z_S \longrightarrow$

Merge Join Example

$R \bowtie_{B=C} S$

Output: (a,2,2,e)



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

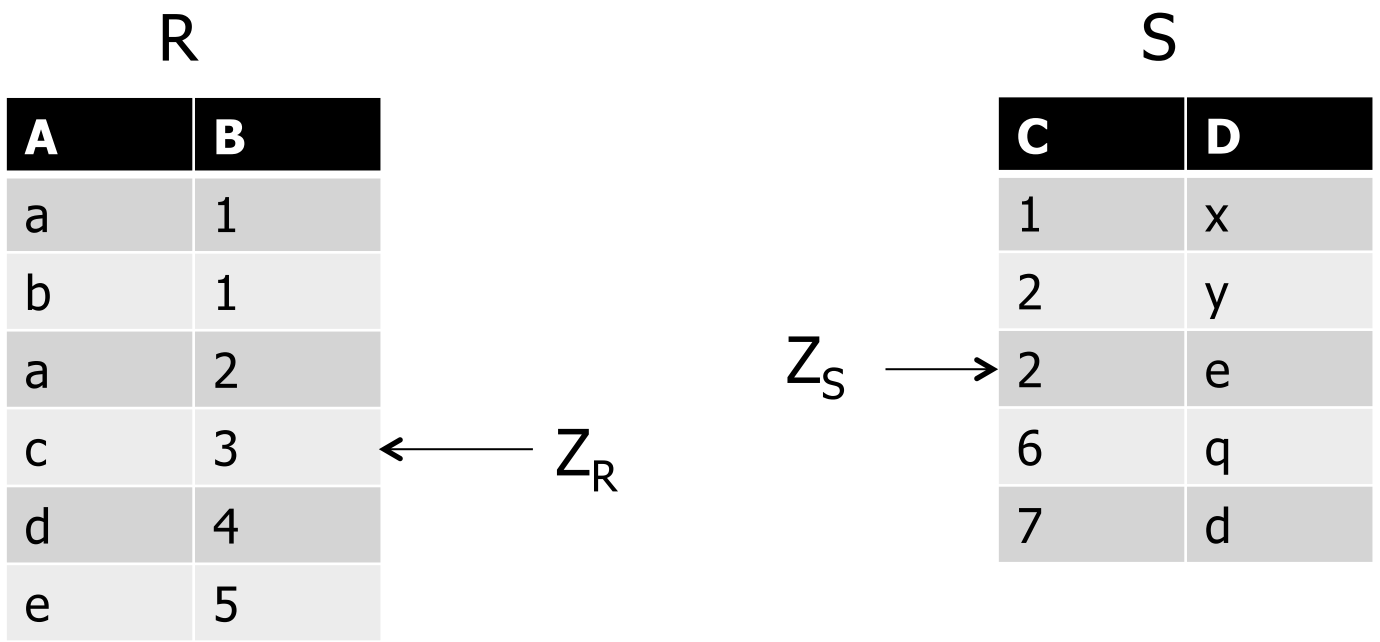
S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Join Example

$R \bowtie_{B=C} S$

R.B > S.C: advance Z_S



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

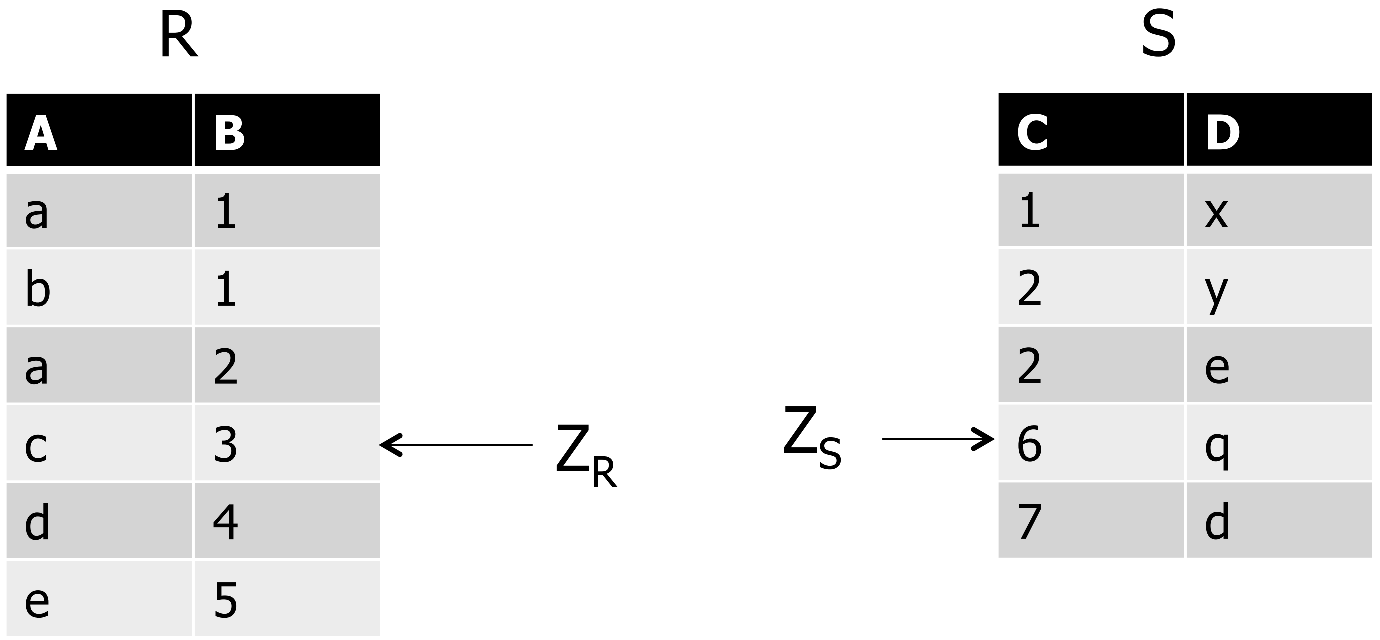
S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Join Example

$R \bowtie_{B=C} S$

R.B < S.C: advance Z_R



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

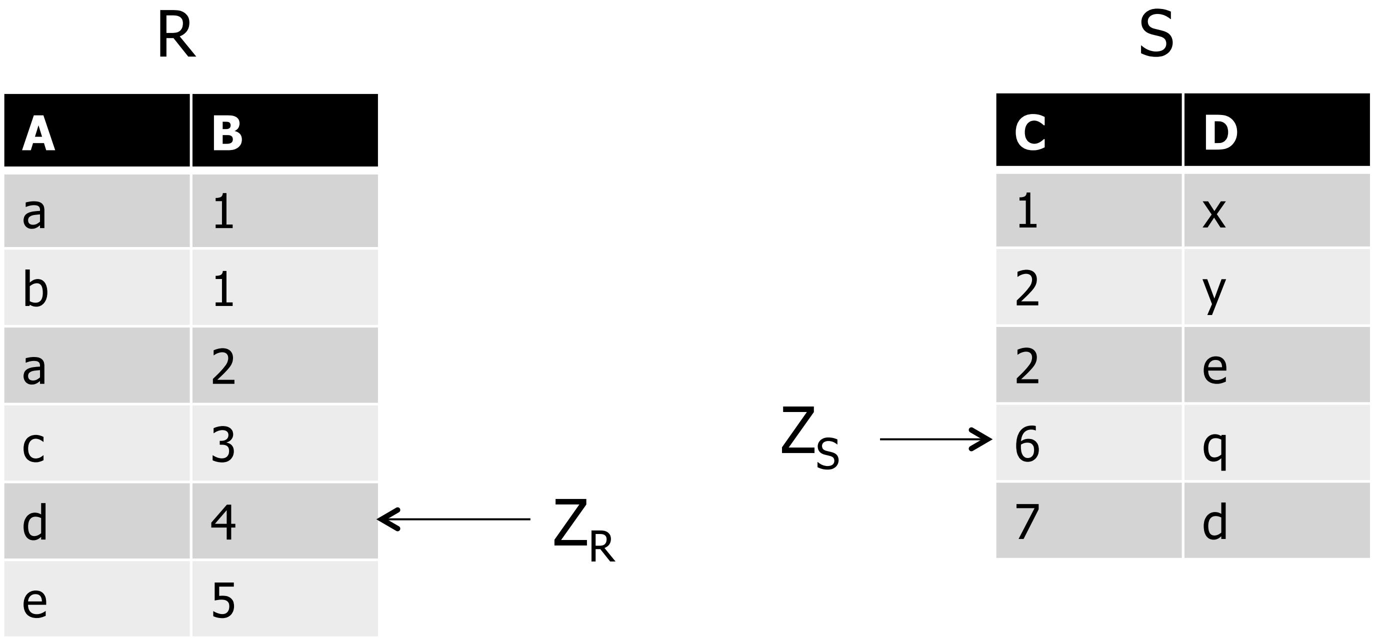
S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Join Example

$R \bowtie_{B=C} S$

R.B < S.C: advance Z_R



R

A	B
a	1
b	1
a	2
c	3
d	4
e	5

S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Join Example

$R \bowtie_{B=C} S$

R.B < S.C: **DONE**

R

A	**B**
a	1
b	1
a	2
c	3
d	4
e	5

$\leftarrow Z_R$ (at row e 5)

S

C	**D**
1	x
2	y
2	e
6	q
7	d

$Z_S \rightarrow$ (at row 6 q)

Example 1(c) Merge Join

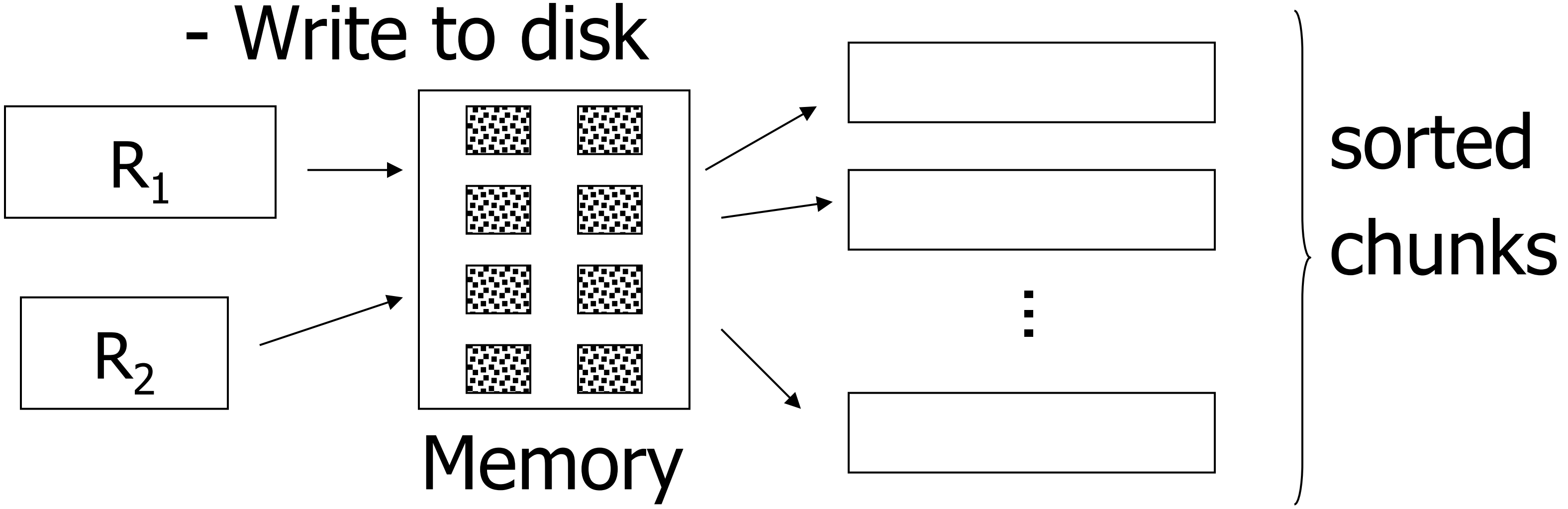
- R_1, R_2 not ordered, but contiguous

--> Need to sort R_1, R_2 first

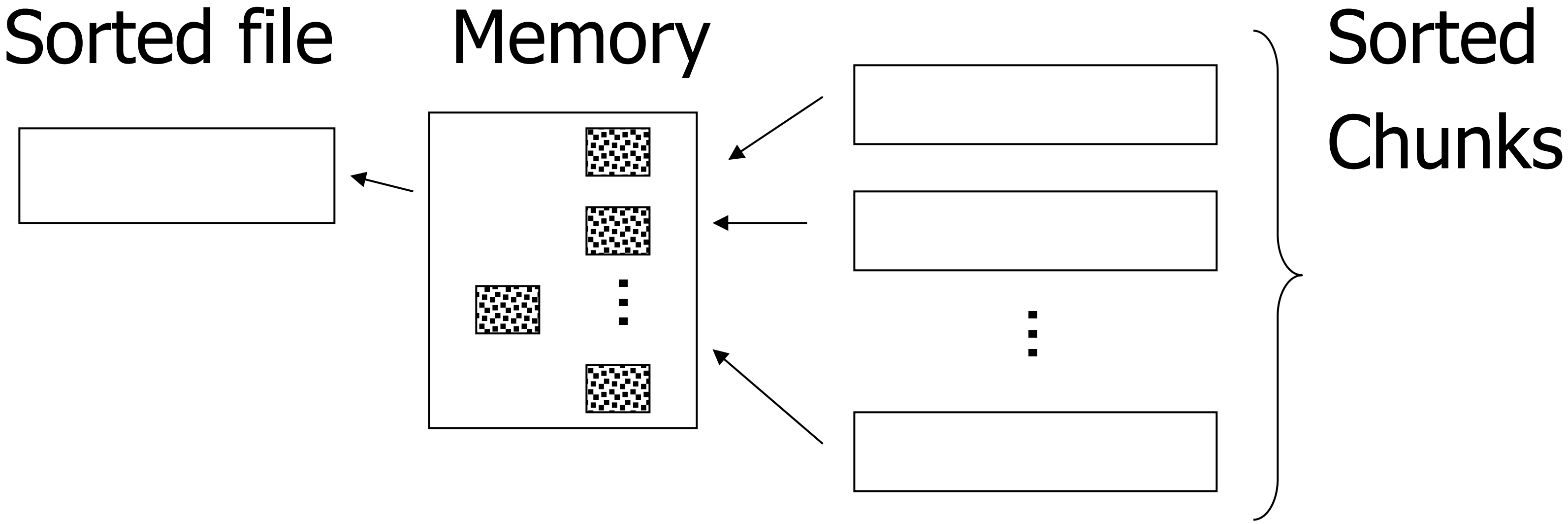
One way to sort: Merge Sort

(i) For each 100 blk chunk of R:

- Read chunk
- Sort in memory
- Write to disk



(ii) Read all chunks + merge + write out



Cost: Sort

Each tuple is read,written,
read, written

so...

Sort cost R_1: 4 x 1,000 = 4,000

Sort cost R_2: 4 x 500 = 2,000

Example 1(d) Merge Join (continued)

R_1,R_2 contiguous, but unordered

Total cost = sort cost + join cost

= 6,000 + 1,500 = 7,500 IOs

Example 1(c) Merge Join (continued)

R_1,R_2 contiguous, but unordered

Total cost = sort cost + join cost

= 6,000 + 1,500 = 7,500 IOs

But: Iteration cost = 5,500

so merge joint does not pay off!

But say R_1 = 10,000 blocks contiguous
R_2 = 5,000 blocks not ordered

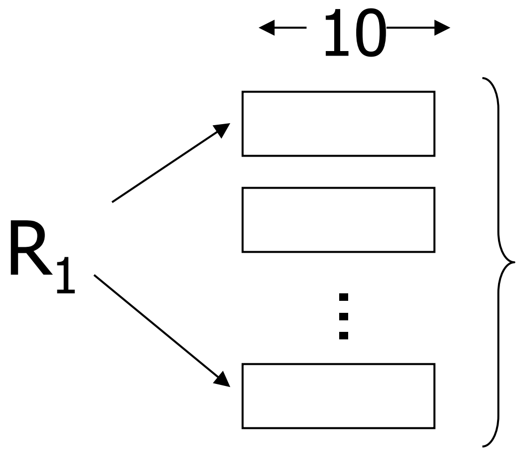
Iterate: $\frac{5000}{100}$ x (100+10,000) = 50 x 10,100
= 505,000 IOs

Merge join: 5(10,000+5,000) = 75,000 IOs

Merge Join (with sort) WINS!

How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks



100 chunks ⇒ to merge, need 100 blocks!

In general:

Say k blocks in memory

x blocks for relation sort

chunks = (x/k)   size of chunk = k

In general:

Say k blocks in memory

x blocks for relation sort

chunks = (x/k)   size of chunk = k

chunks < buffers available for merge

In general:

Say k blocks in memory

x blocks for relation sort

chunks = (x/k) size of chunk = k

chunks < buffers available for merge

so... $(x/k) \leq k$

or $k^2 \geq x$ or $k \geq \sqrt{x}$

In our example

R_1 is 1000 blocks, $k \geq 31.62$

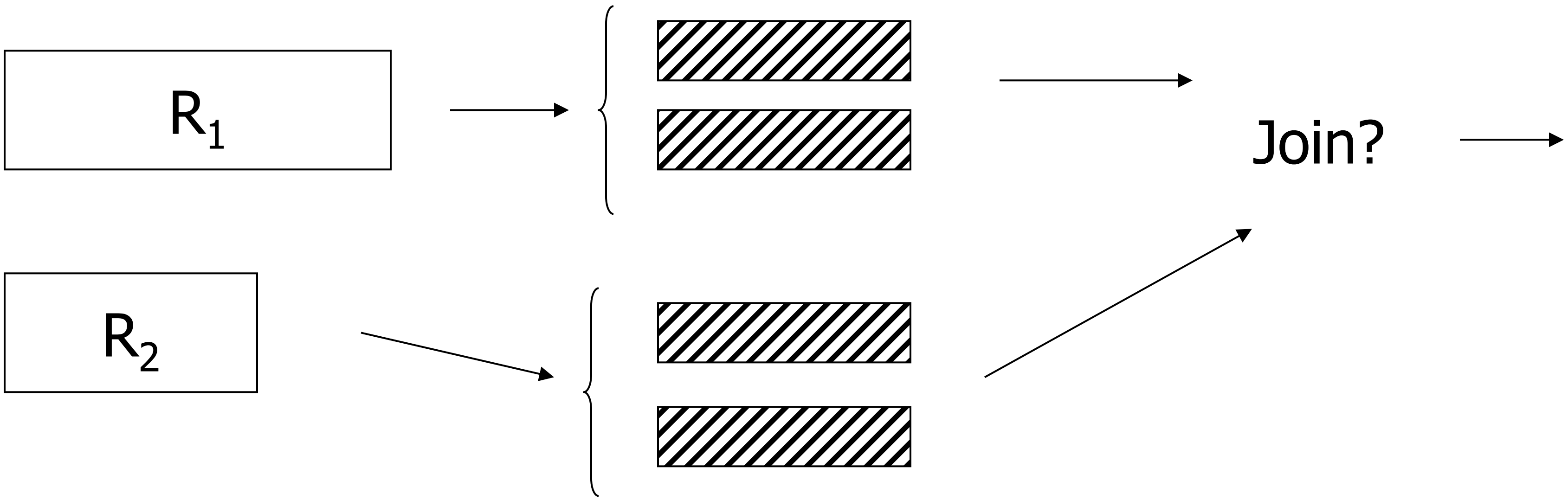
R_2 is 500 blocks, $k \geq 22.36$

Need at least 32 buffers

Again: in practice we would not want to use only one buffer per run!

Can we improve on merge join?

Hint: do we really need the fully sorted files?



sorted runs

Cost of improved merge join:

C = Read R_1 + write R_1 into runs

+ read R_2 + write R_2 into runs

+ join

= 2,000 + 1,000 + 1,500 = 4,500

--> Memory requirement?

Example 1(d) Index Join

- Assume R_1.C index exists; 2 levels
- Assume R_2 contiguous, unordered

- Assume R_1.C index fits in memory

Cost: Reads: 500 IOs

for each R_2 tuple:

- probe index - free
- if match, read R_1 tuple: 1 IO

What is expected # of matching tuples?

(a) say $R_1.C$ is key, $R_2.C$ is foreign key

then expect = 1

(b) say $V(R_1,C) = 5000$, $T(R_1) = 10{,}000$

with uniform assumption

expect = 10,000/5,000 = 2

What is expected # of matching tuples?

(c) Say $DOM(R_1, C)=1{,}000{,}000$

$T(R_1) = 10{,}000$

with alternate assumption

$$\text{Expect} = \frac{10{,}000}{1{,}000{,}000} = \frac{1}{100}$$

Total cost with index join

(a) Total cost = 500+5000(1)1 = 5,500

(b) Total cost = 500+5000(2)1 = 10,500

(c) Total cost = 500+5000(1/100)1=550

What if index does not fit in memory?

Example: say R_1.C index is 201 blocks

- Keep root + 99 leaf nodes in memory
- Expected cost of each probe is

$$E = (0)\frac{99}{200} + (1)\frac{101}{200} \approx 0.5$$

Total cost (including probes)

= 500+5000 [Probe + get records]

= 500+5000 [0.5+2]   uniform assumption

= 500+12,500 = 13,000   (case b)

Total cost (including probes)

= 500+5000 [Probe + get records]

= 500+5000 [0.5+2] uniform assumption

= 500+12,500 = 13,000 (case b)

For case (c):

= $500+5000[0.5 \times 1 + (1/100) \times 1]$

= 500+2500+50 = 3050 IOs

So far

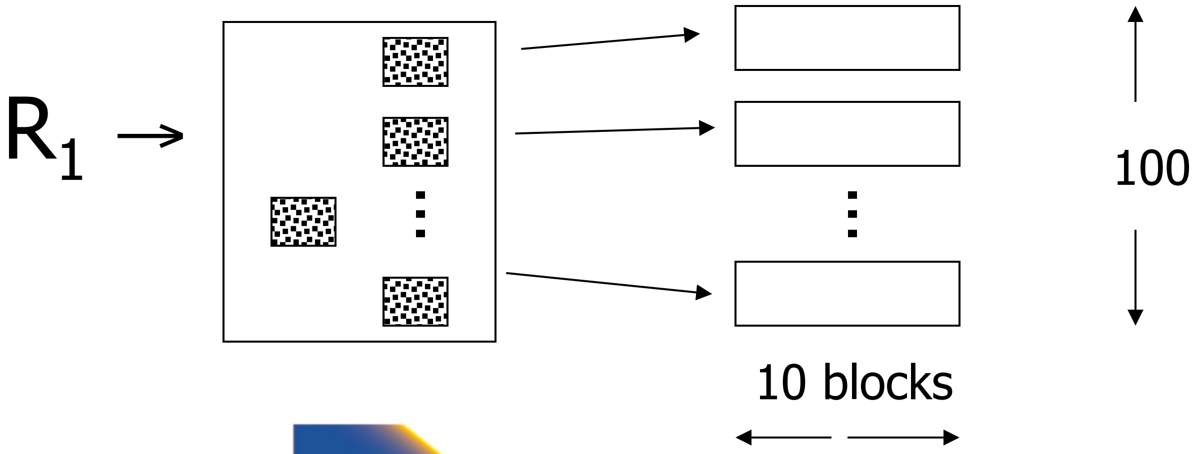
Nested Loop	5500
Merge join	1500
Sort+Merge Join	7500 → 4500
R_1.C Index	5500 → 3050 → 550
R_2.C Index	________

Example 1(e) Partition Hash Join

- R_1, R_2 contiguous (un-ordered)

→ Use 100 buckets

→ Read R_1, hash, + write buckets

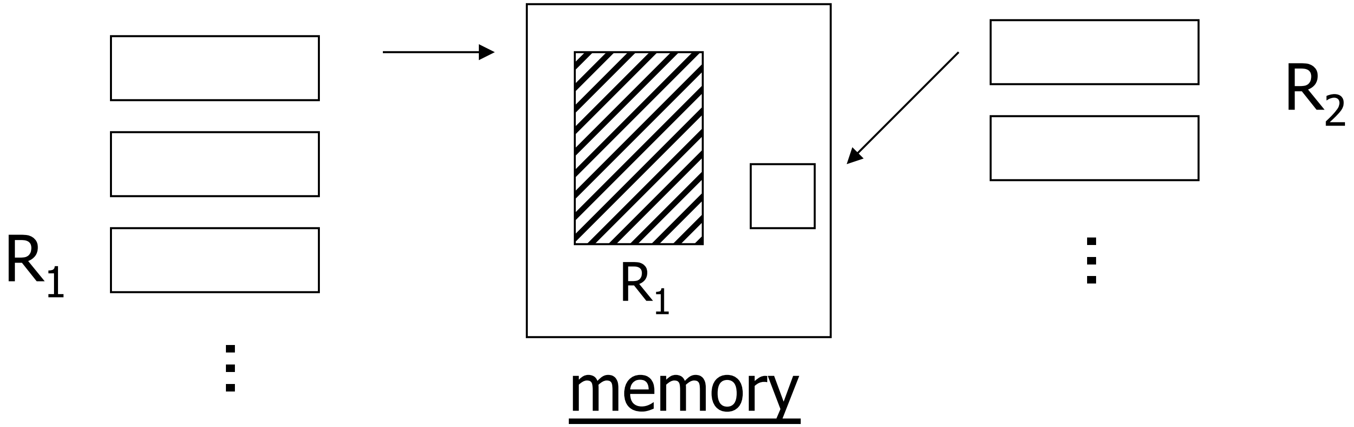


-> Same for R_2

-> Read one R_1 bucket; build memory hash table

-using different hash function h'

-> Read corresponding R_2 bucket + hash probe



✏ Then repeat for all buckets

Cost:

"Bucketize:" Read R_1 + write

Read R_2 + write

Join: Read R_1, R_2

Total cost = 3 x [1000+500] = 4500

Cost:

"Bucketize:" Read R_1 + write

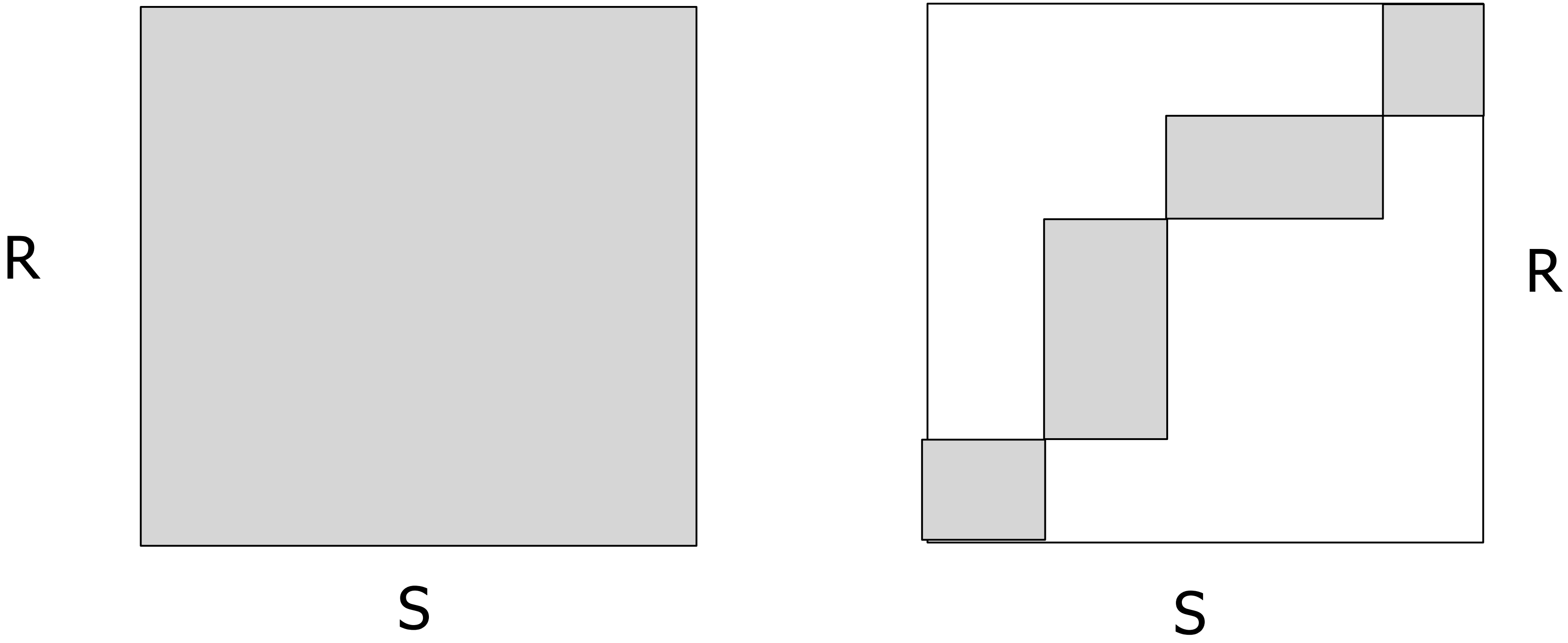
Read R_2 + write

Join: Read R_1, R_2

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Why is Hash Join good?



Minimum memory requirements:

Size of R_1 bucket = (x/k)

k = number of memory buffers

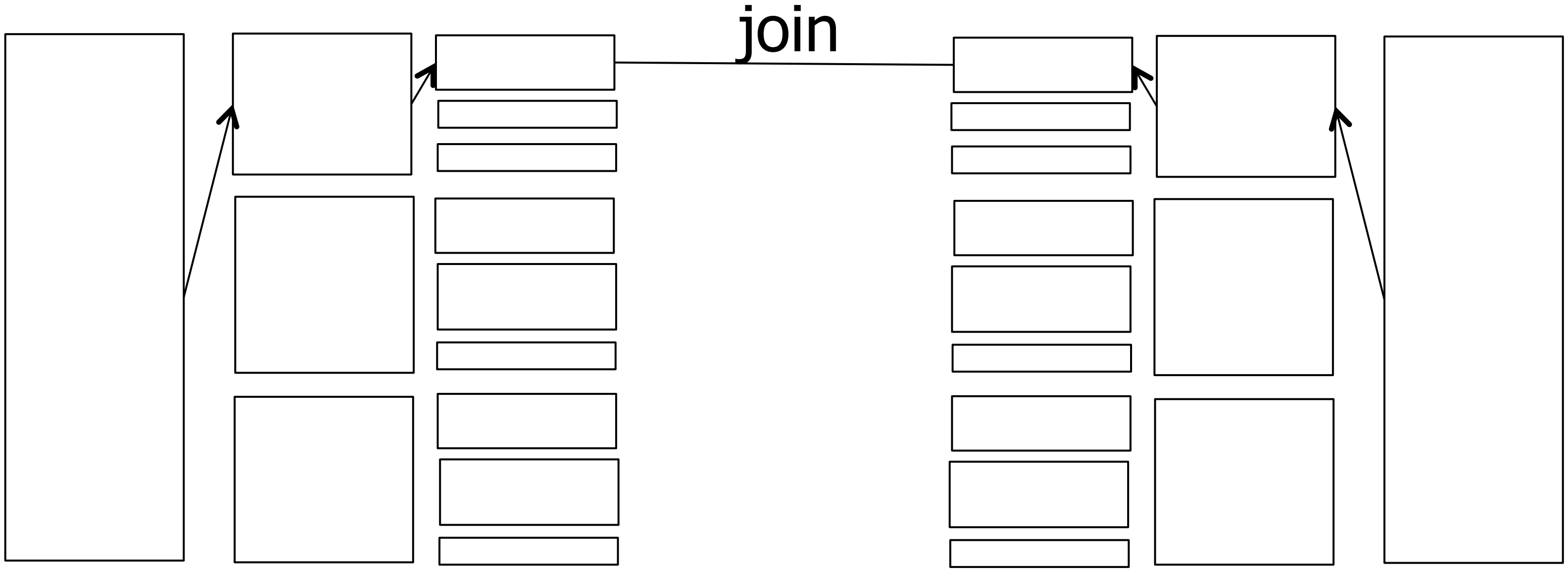
x = number of R_1 blocks

So... $(x/k) < k$

$k > \sqrt{x}$ need: k+1 total memory buffers

Can we use Hash-join when buckets do not fit into memory?:

- Treat buckets as relations and apply Hash-join recursively

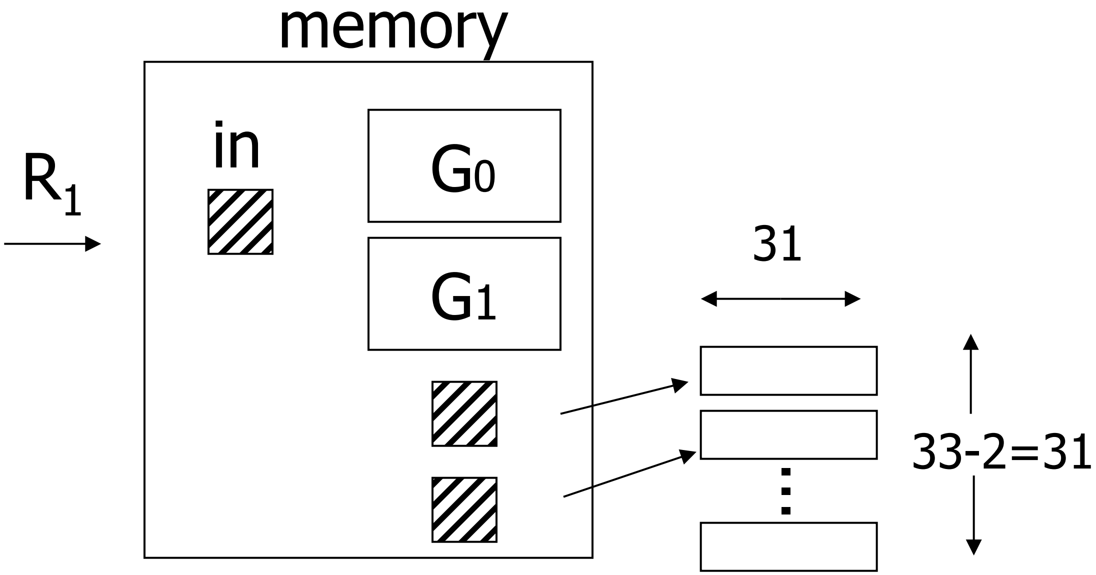


Duality Hashing-Sorting

- Both partition inputs
- Until input fits into memory
- Logarithmic number of phases in memory size

Trick: keep some buckets in memory

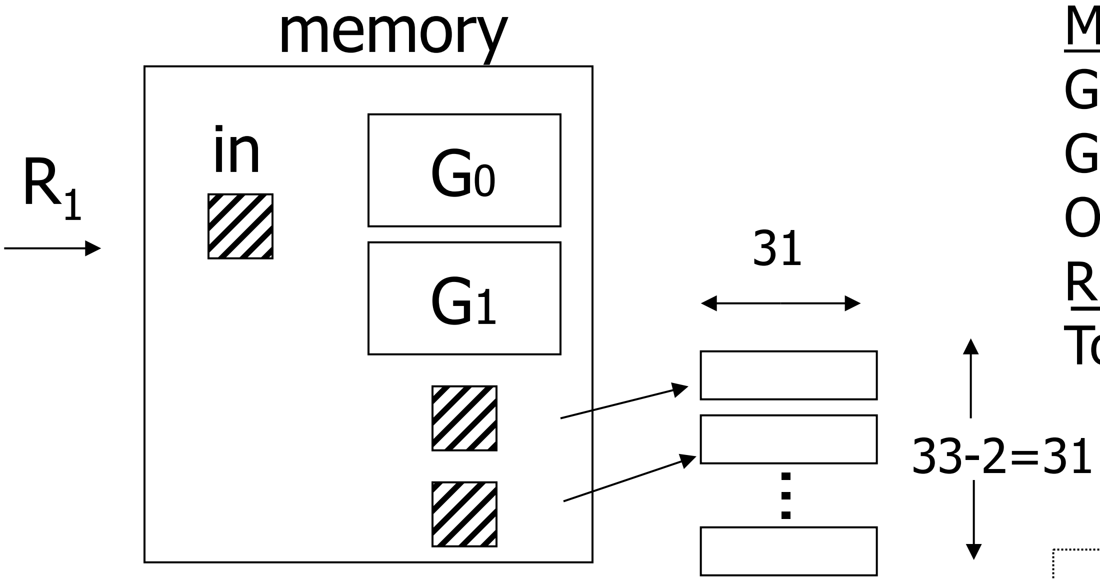
E.g., k' =33 R_1 buckets = 31 blocks
keep 2 in memory



called hybrid hash-join

Trick: keep some buckets in memory

E.g., k' =33 R_1 buckets = 31 blocks
keep 2 in memory



Memory use:

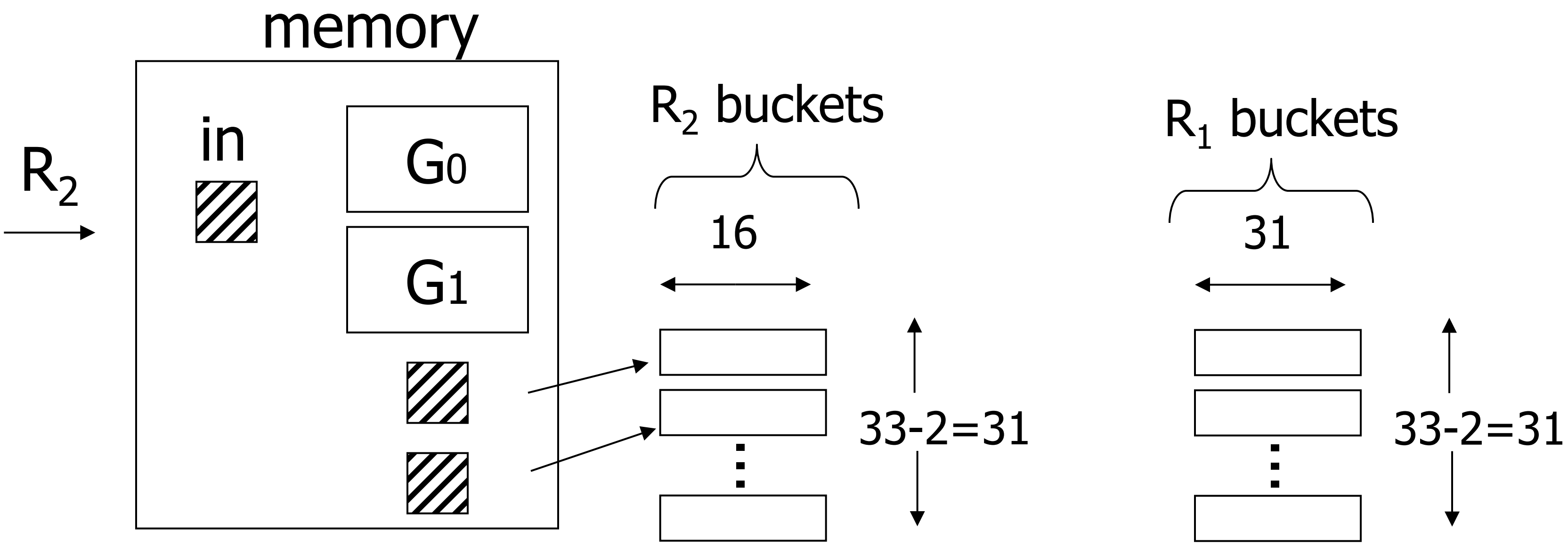
G_0	31 buffers
G_1	31 buffers
Output	33-2 buffers
R_1 input	1
Total	94 buffers

6 buffers to spare!!

called hybrid hash-join

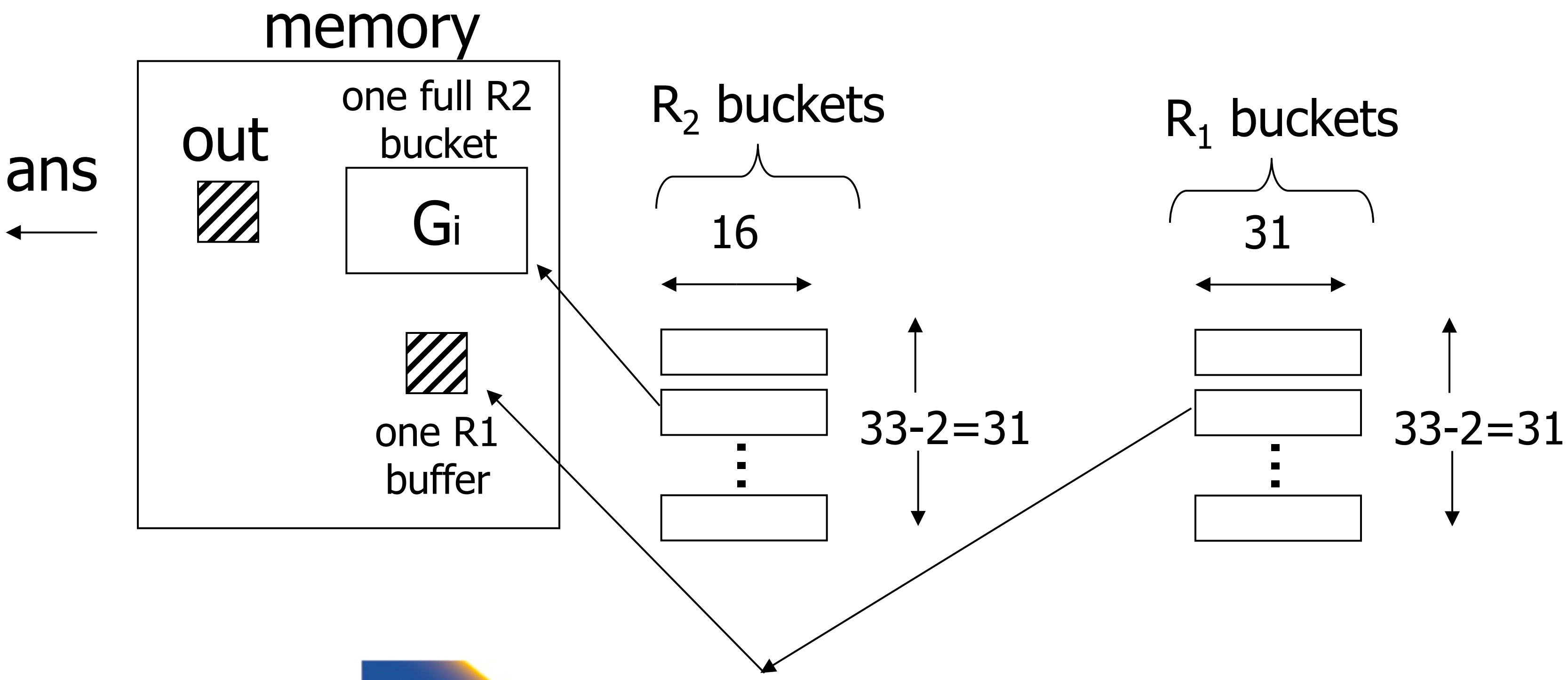
Next: Bucketize R_2

- R_2 buckets =500/33= 16 blocks
- Two of the R_2 buckets joined immediately with G_0,G_1



Finally: Join remaining buckets

- for each bucket pair:
  - read one of the buckets into memory
  - join with second bucket

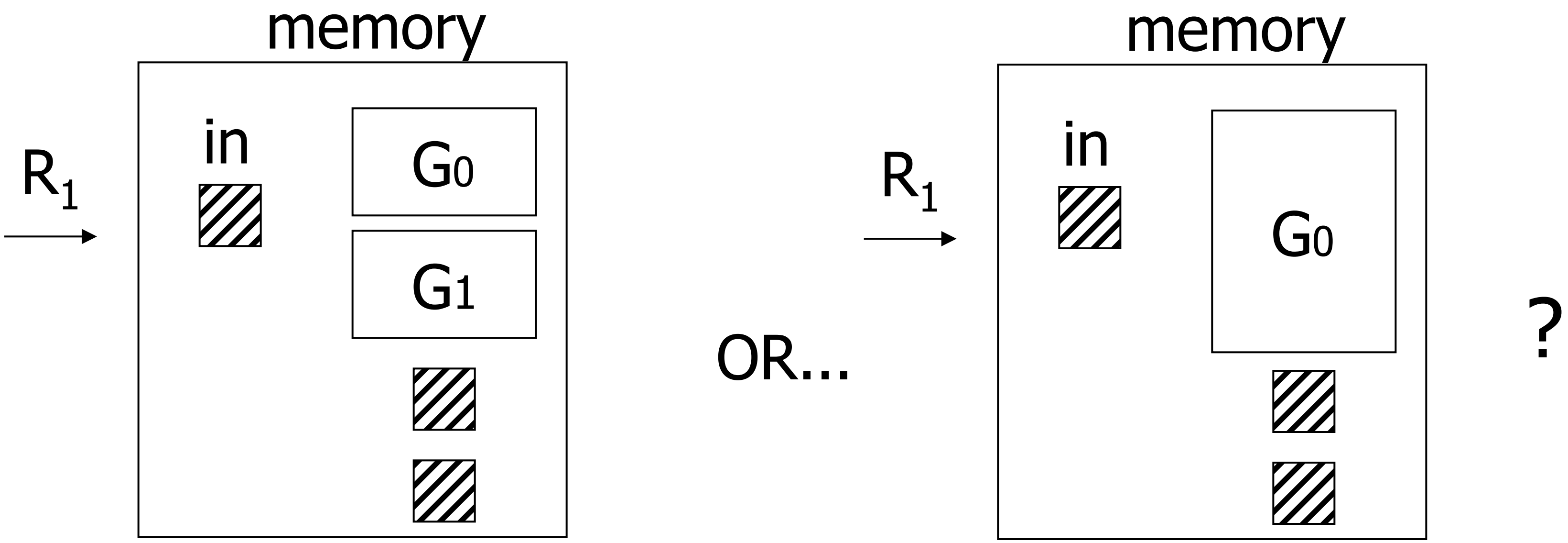


Cost

- Bucketize R_1 = 1000+31×31=1961
- To bucketize R_2, only write 31 buckets:
  so, cost = 500+31×16=996
- To compare join (2 buckets already done)
  read 31×31+31×16=1457

Total cost = 1961+996+1457 = 4414

- How many buckets in memory?



☛ See textbook for answer...

Another hash join trick:

- Only write into buckets
  <val,ptr> pairs
- When we get a match in join phase,
  must fetch tuples

- To illustrate cost computation, assume:
  - – 100 <val,ptr> pairs/block
  - – expected number of result tuples is 100

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100
- Build hash table for R_2 in memory
  5000 tuples → 5000/100 = 50 blocks
- Read R_1 and match
- Read ~ 100 R_2 tuples

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100
- Build hash table for R_2 in memory
  5000 tuples → 5000/100 = 50 blocks
- Read R_1 and match
- Read ~ 100 R_2 tuples

Total cost =

Read R_2: 500
Read R_1: 1000
Get tuples: 100

1600

So far:

Iterate	5500
Merge join	1500
Sort+merge joint	7500
R_1.C index	5500 → 550
R_2.C index	______
Build R_1.C index	______
Build R_2.C index	______
Hash join	4500+
with trick,R_1 first	4414
with trick,R_2 first	______
Hash join, pointers	1600

Yet another hash join trick:

- Combine the ideas of
  - block nested-loop with hash join
- Use memory to build hash-table for one chunk of relation
- Find join partners in O(1) instead of O(M)
- Trade-off
  - Space-overhead of hash-table
  - Time savings from look-up

Summary

- Nested Loop ok for "small" relations (relative to memory size)
  - Need for complex join condition
- For equi-join, where relations not sorted and no indexes exist, hash join usually best

- Sort + merge join good for non-equi-join (e.g., $R_1.C > R_2.C$)
- If relations already sorted, use merge join
- If index exists, it could be useful

(depends on expected result size)

Join Comparison

N_i= number of tuples in R_i
B(R_i) = number of blocks of R_i
#P = number of partition steps for hash join
P_{ij} = average number of join partners

Algorithm	#I/O	Memory	Disk Space
Nested Loop (block)	$B(R_1) * B(R_2) / M$	3	0
Index Nested Loop	$B(R_1) + N_1 * P_{12}$	B(Index) + 2	0
Merge (sorted)	$B(R_1) + B(R_2)$	Max tuples =	0
Merge (unsorted)	$B(R_1) + B(R_2) +$ (sort – 1 pass)	sort	$B(R_1) + B(R_2)$
Hash	$(2\#P + 1)(B(R_1) + B(R_2))$	root(max($B(R_1)$, $B(R_2)$), #P + 1)	~$B(R_1) + B(R_2)$

Why do we need nested loop?

- Remember not all join implementations work for all types of join conditions

Algorithm	Type of Condition	Example
Nested Loop	any	a LIKE '%hello%'
Index Nested Loop	Supported by index: Equi-join (hash) Equi or range (B-tree)	a = b a < b
Merge	Equalities and ranges	a < b, a = b AND c = d
Hash	Equi-join	a = b

Outer Joins

- How to implement (left) outer joins?
- Nested Loop and Merge
  - Use a flag that is set to true if we find a match for an outer tuple
  - If flag is false fill with NULL
- Hash
  - If no matching tuple fill with NULL

Merge Left Outer Join

$R ⟕_{B=C} S$

Output: (a,1,1,X)

R

A	B
a	1
d	4
e	5

← Z_R

Z_S →

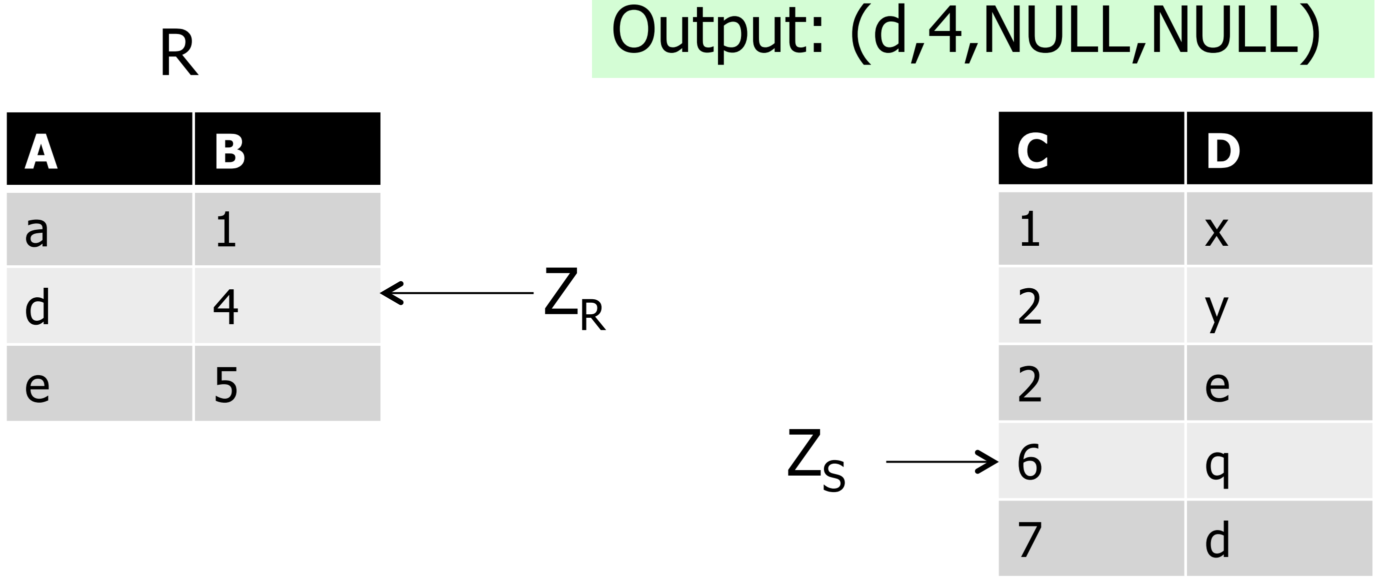
S

C	D
1	x
2	y
2	e
6	q
7	d

Merge Left Outer Join

$R \mathbin{⟕}_{B=C} S$

No match for (d,4)
Output: (d,4,NULL,NULL)



R

A	B
a	1
d	4
e	5

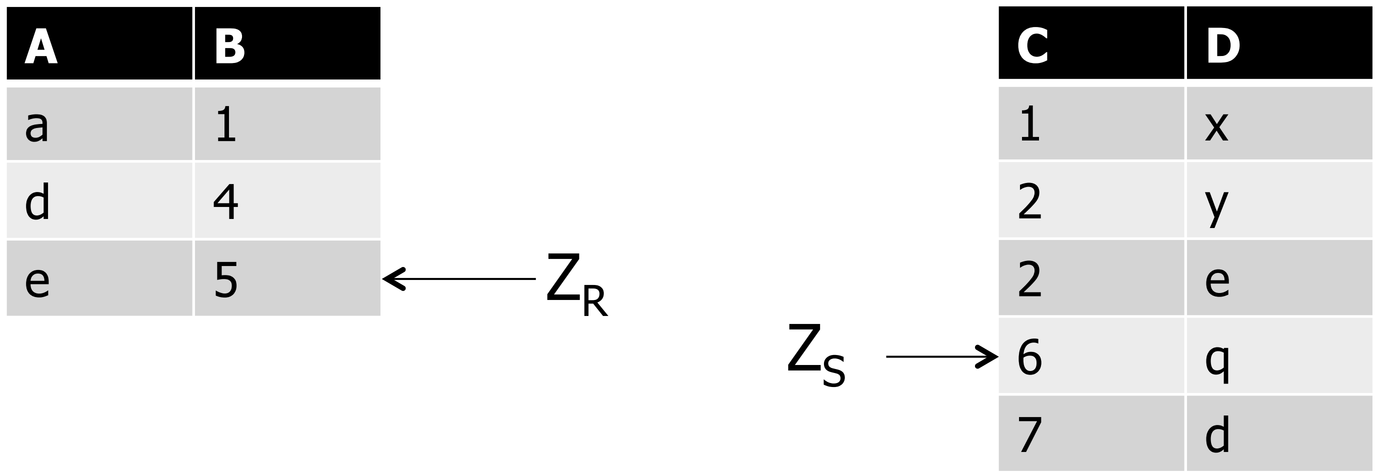
C	D
1	x
2	y
2	e
6	q
7	d

Merge Left Outer Join

$R ⟕_{B=C} S$

No match for (e,5)
Output: (e,5,NULL,NULL)

R



A	B
a	1
d	4
e	5

C	D
1	x
2	y
2	e
6	q
7	d

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Aggregation

- Have to compute aggregation functions
  - for each group of tuples from input
- Groups
  - Determined by equality of group-by attributes

Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

a	b
3	1
4	2
3	1
1	2
1	2

sum(a)	b
6	1
6	2

Aggregation Function Interface

- `init()`
  - Initialize state
- `update(tuple)`
  - Update state with information from tuple
- `close()`
  - Return result and clean-up

Implementation SUM(A)

- `init()`
  - `sum := 0`
- `update(tuple)`
  - `sum += tuple.A`
- `close()`
  - **`return`** `sum`

Aggregation Implementations

- Sorting
  - Sort input on group-by attributes
  - On group boundaries output tuple
- Hashing
  - Store current aggregated values for each group in hash table
  - Update with newly arriving tuples
  - Output result after processing all inputs

Grouping by sorting

- Similar to Merge join
- Sort R on group-by attribute
- Scan through sorted input
  - If group-by values change
    - Output using close() and call init()
  - Otherwise
    - Call update()

Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

sort

a	b
3	1
4	2
3	1
1	2
1	2

a	b
3	1
3	1
4	2
1	2
1	2

init()

0

Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

a	b
3	1
3	1
4	2
1	2
1	2

update(3,1)

3

Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

a	b
3	1
3	1
4	2
1	2
1	2

update(3,1)

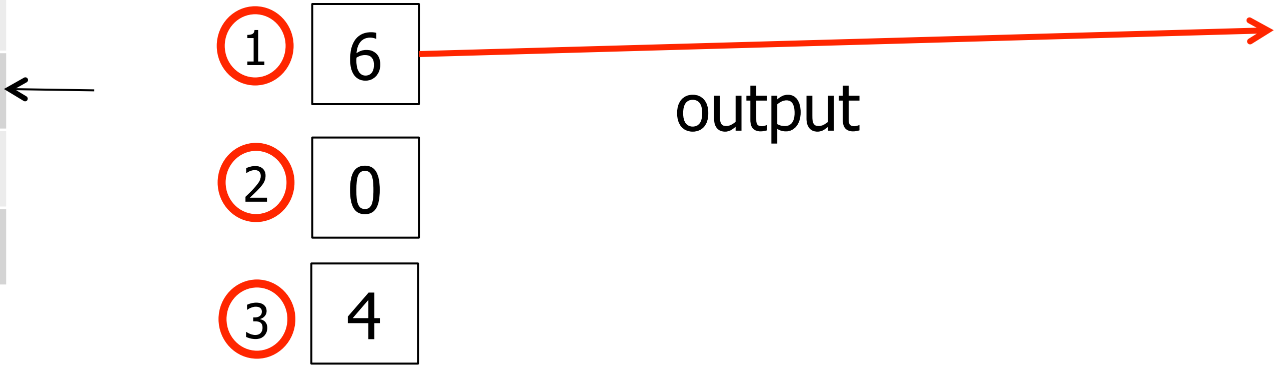
6

Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

a	b
3	1
3	1
4	2
1	2
1	2

Group by changed!
close(), init(), update(4,2)



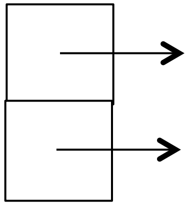
Grouping by Hashing

- Create in-memory hash-table
- For each input tuple probe hash table with group by values
  - If no entry exists then call init(), update(), and add entry
  - Otherwise call update() for entry
- Loop through all entries in hash-table and ouput calling close()

Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

a	b
3	1
4	2
3	1
1	2
1	2

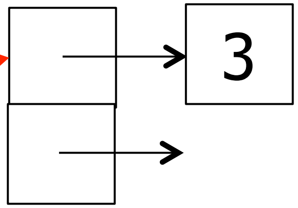


Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

Init() and update(3,1)

a	b
3	1
4	2
3	1
1	2
1	2

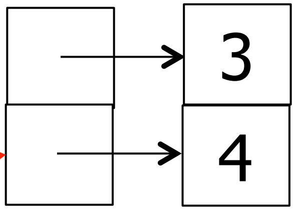


Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

Init() and update(4,2)

a	b
3	1
4	2
3	1
1	2
1	2

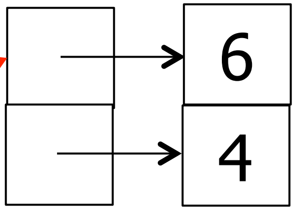


Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

update(3,1)

a	b
3	1
4	2
3	1
1	2
1	2

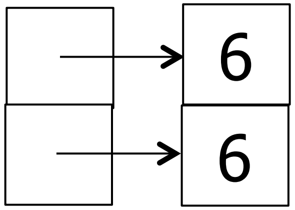


Aggregation Example

```
SELECT sum(a),b
FROM R
GROUP BY b
```

- Loop through hash table entries
- Output tuples

a	b
3	1
4	2
3	1
1	2
1	2



Aggregation Summary

- Hashing
  - No sorting -> no extra I/O
  - Hash table has to fit into memory
  - No outputs before all inputs have been processed
- Sorting
  - No memory required
  - Output one group at a time

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Duplicate Elimination

- Equivalent to group-by on all attributes
- -> Can use aggregation implementations
- Optimization
  - Hash
    - Directly output tuple and use hash table only to avoid outputting duplicates

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Set Operations

- Can be modeled as join
  - with different output requirements
- As aggregation/group by on all columns
  - with different output requirements

Union

- Bag union
  - – Append the two inputs
  - – E.g., using three buffers
- Set union
  - – Apply duplicate removal to result

Intersection

- Set version
  - Equivalent to join + project + duplicate removal
  - 3-state aggregate function (found left, found right, found both)
- Bag version
  - Join + project + min(i,j)
  - Aggegate min(count(i),count(j))

Set Difference

- Using join methods
  - Find matching tuples
  - If no match found, then output
- Using aggregation
  - count(i) – count(j) **(bag)**
  - true(i) AND false(j) **(set)**

Summary

- Operator implementations
  - Joins!
  - Other operators
- Cost estimations
  - I/O
  - memory
- Query processing architectures

Next

- Query Optimization Physical
- -> How to **efficiently** choose an **efficient** plan