

Carnegie  
Mellon  
University

Intro to Database  
Systems (15-445/645)

Lecture #10

# Sorting & Aggregations

SPRING 2024 >> Prof. Jignesh Patel



# ADMINISTRIVIA

---

Done

**Project #1** is due Feb 18<sup>th</sup> @ 11:59pm

→ Special Office Hours: Feb 17<sup>th</sup> @ 3:00-5:00pm GHC 4405

**Homework #2** is due Wed Feb 16<sup>th</sup> @ 11:59pm

**Project #2** is due Wed March 13<sup>th</sup> @ 11:59pm

**Mid-Term Exam** is on Wednesday Feb 28<sup>th</sup>

→ During regular class time from 12:30-1:50pm

→ Please contact us if you need accommodations.

→ Review session on Feb 26 during regular class time. Not recorded!

Bring your questions to class – it will be an interactive review session.

# COURSE STATUS

---

We are now going to talk about how to execute queries using the DBMS components we have discussed so far.

Next four lectures:

- Operator Algorithms
- Query Processing Models
- Runtime Architectures

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

# QUERY PLAN

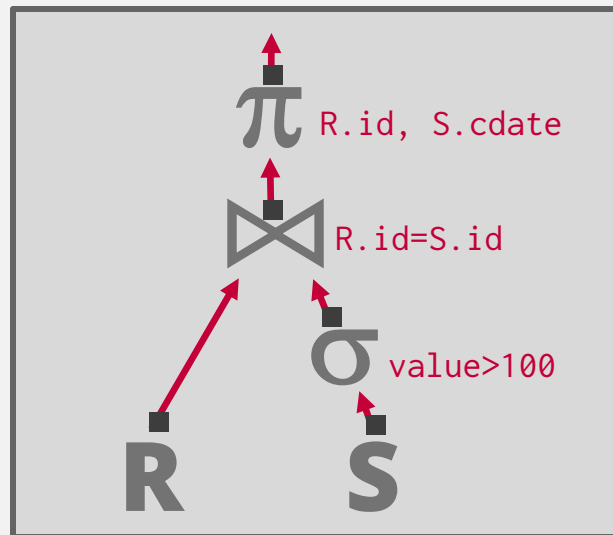
The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

→ We will discuss the granularity of the data movement next week.

The output of the root node is the result of the query.

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



# DISK-ORIENTED DBMS

---

Just like it cannot assume that a table fits entirely in memory, a disk-oriented DBMS cannot assume that query results fit in memory.

We will use the buffer pool to implement algorithms that need to spill to disk.

We are also going to prefer algorithms that maximize the amount of sequential I/O.

# WHY DO WE NEED TO SORT?

---

Relational model/SQL is unsorted.

Queries may request that tuples are sorted in a specific way (**ORDER BY**).

But even if a query does not specify an order, we may still want to sort to do other things:

- Trivial to support duplicate elimination (**DISTINCT**).
- Bulk loading sorted tuples into a B+Tree index is faster.
- Aggregations (**GROUP BY**).

# IN-MEMORY SORTING

---

Most **database systems** use Quicksort for in-memory sorting.

In other **data platforms**, notably Python – the default sort algorithm is TimSort. It is a combination of insertion and binary merge sort. Often works well on real data.

If data fits in memory, then we can use a standard sorting algorithm like Quicksort.

If data does not fit in memory, then we need to use a technique that is aware of the cost of reading and writing disk pages ...

<https://www.toptal.com/developers/sorting-algorithms>

<https://visualgo.net/en/sorting>



# TODAY'S AGENDA

---

Top-N Heap Sort

External Merge Sort

Aggregations

# TOP-N HEAP SORT

---

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---

*Sorted Heap*

--	--	--	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

3			
---	--	--	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

4	3		
---	---	--	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

6	4	3	
---	---	---	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

6	4	3	2
---	---	---	---



# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



Skip!

*Sorted Heap*

6	4	3	2
---	---	---	---

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

4	3	2	1
---	---	---	---

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

4	4	3	2	1			
---	---	---	---	---	--	--	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



*Sorted Heap*

4	4	4	3	2	1		
---	---	---	---	---	---	--	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---



Skip and done!

*Sorted Heap*

4	4	4	3	2	1		
---	---	---	---	---	---	--	--

# TOP-N HEAP SORT

If a query contains an **ORDER BY** with a **LIMIT**, then the DBMS only needs to scan the data once to find the top-N elements.

Ideal scenario for heapsort: if the top-N elements fit in memory.

→ Scan data once, maintain an in-memory sorted priority queue.

```
SELECT * FROM enrolled
ORDER BY sid
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

3	4	6	2	9	1	4	4	8
---	---	---	---	---	---	---	---	---

*Sorted Heap*

4	4	4	3	2	1		
---	---	---	---	---	---	--	--

Output

# EXTERNAL MERGE SORT

---

Divide-and-conquer algorithm that splits data into separate **runs**, sorts them individually, and then combines them into longer sorted runs.

## Phase #1 – Sorting

→ Sort chunks of data that fit in memory and then write back the sorted chunks to a file on disk.

## Phase #2 – Merging

→ Combine sorted runs into larger chunks.

# SORTED RUN

---

A run is a list of key/value pairs.

**Key:** The attribute(s) to compare to compute the sort order.

**Value:** Two choices

→ Tuple (*early materialization*).

→ Record ID (*late materialization*).

## *Early Materialization*

K1	<i>&lt;Tuple Data&gt;</i>
K2	<i>&lt;Tuple Data&gt;</i>

⋮



# SORTED RUN

A run is a list of key/value pairs.

**Key:** The attribute(s) to compare to compute the sort order.

**Value:** Two choices

→ Tuple (*early materialization*).

→ Record ID (*late materialization*).

## Early Materialization

K1	<Tuple Data>
K2	<Tuple Data>

⋮

## Late Materialization



Record ID

## 2-WAY EXTERNAL MERGE SORT

---

We will start with a simple example of a 2-way external merge sort.

→ “2” is the number of runs that we are going to merge into a new run for each pass.

Data is broken up into  $N$  pages.

The DBMS has a finite number of  $B$  buffer pool pages to hold input and output data.

# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

---

## Pass #0

- Read one page of the table into memory
- Sort page into a “run” and write it back to disk
- Repeat until the whole table has been sorted into runs

## Pass #1,2,3,...

- Recursively merge pairs of runs into runs twice as long
- Need at least 3 buffer pages (2 for input, 1 for output)

# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

In each pass, we read and write every page in the file.

3,4	6,2	9,4	8,7	5,6	3,1	2	∅
-----	-----	-----	-----	-----	-----	---	---

EOF

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$

# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

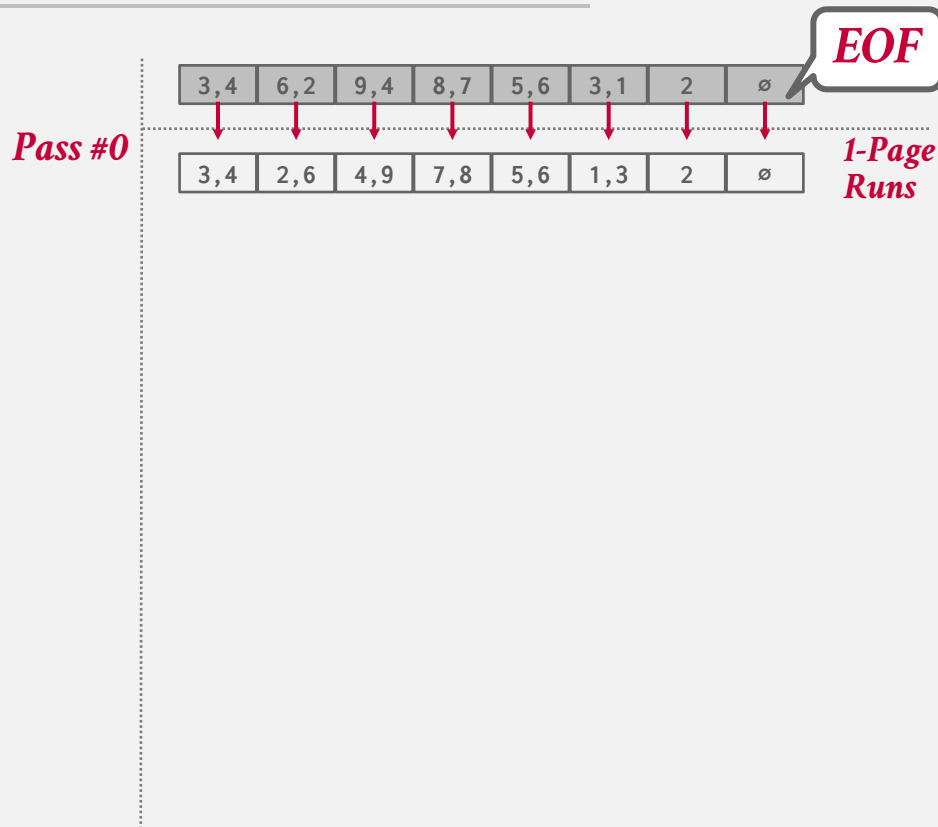
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

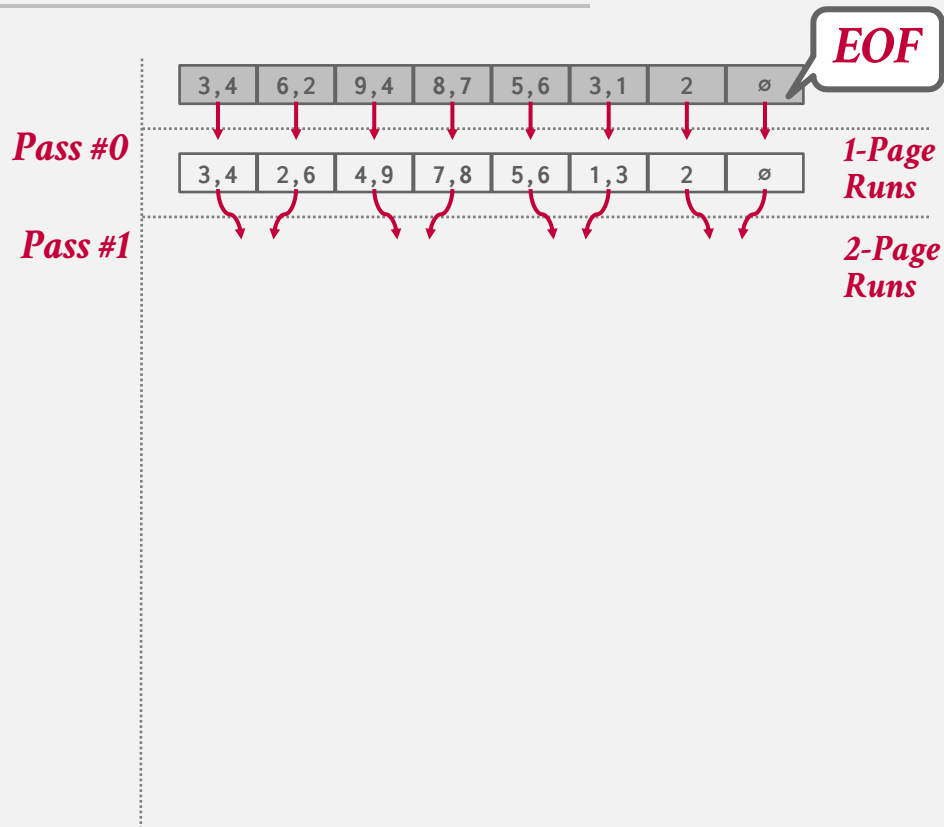
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

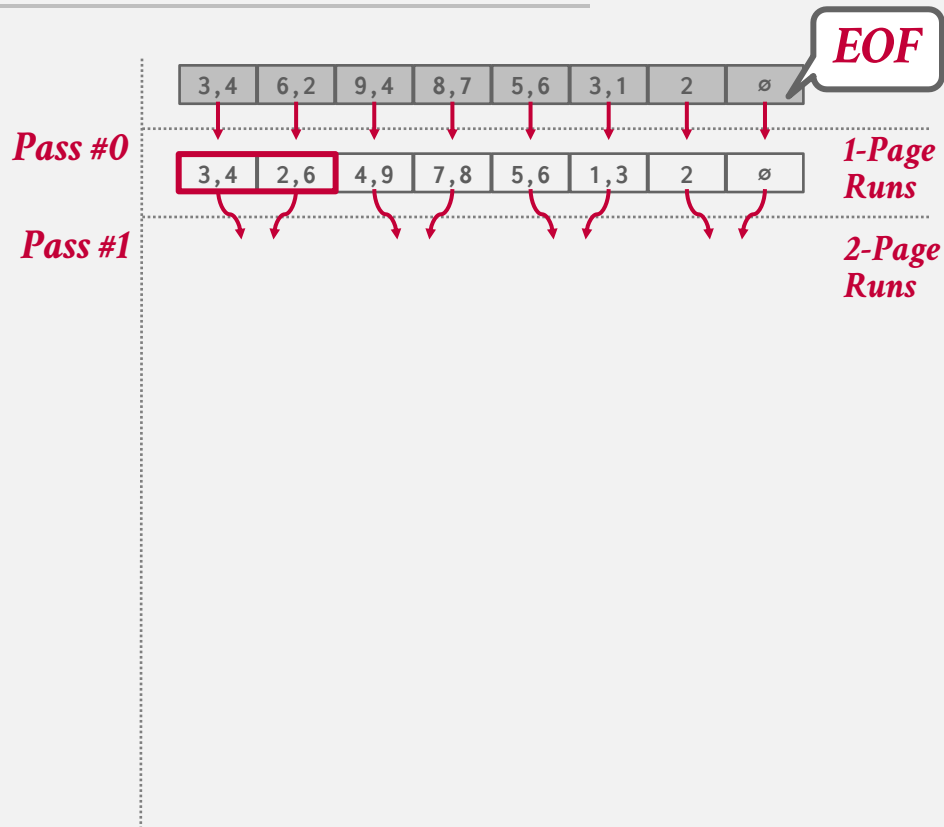
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

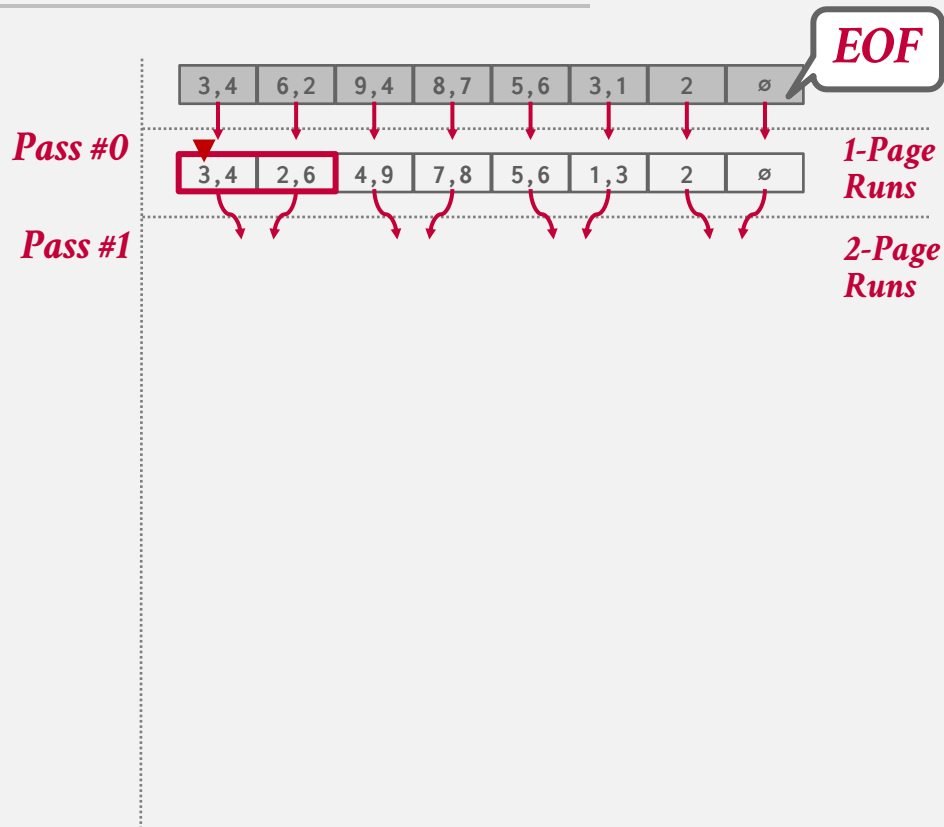
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$





# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

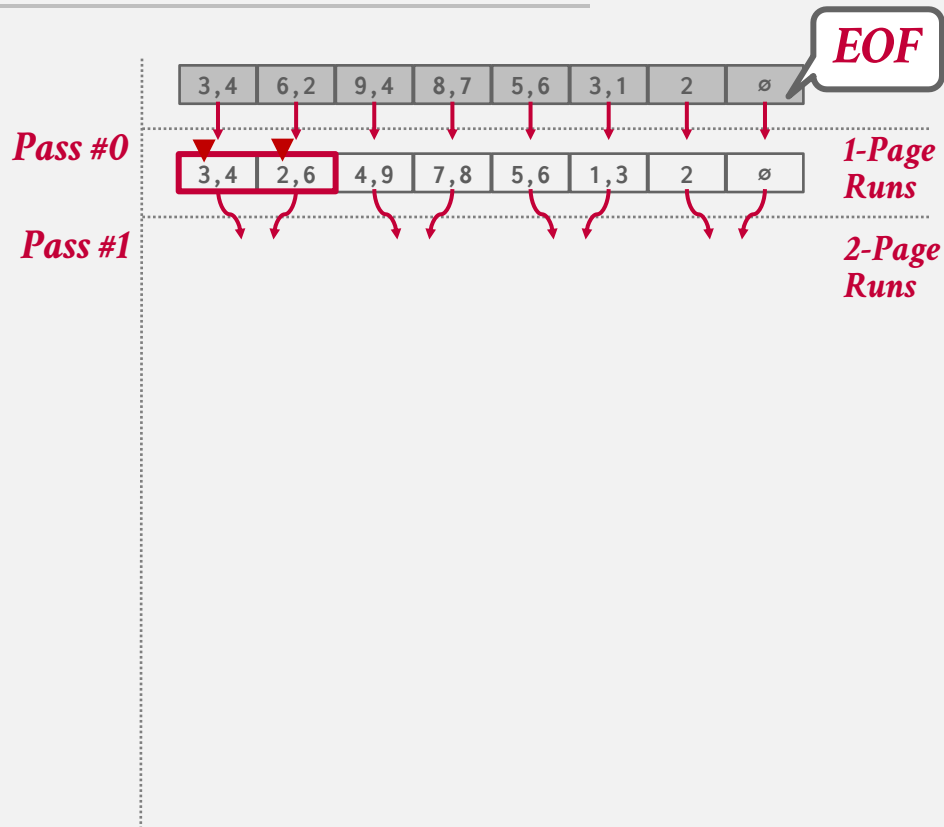
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

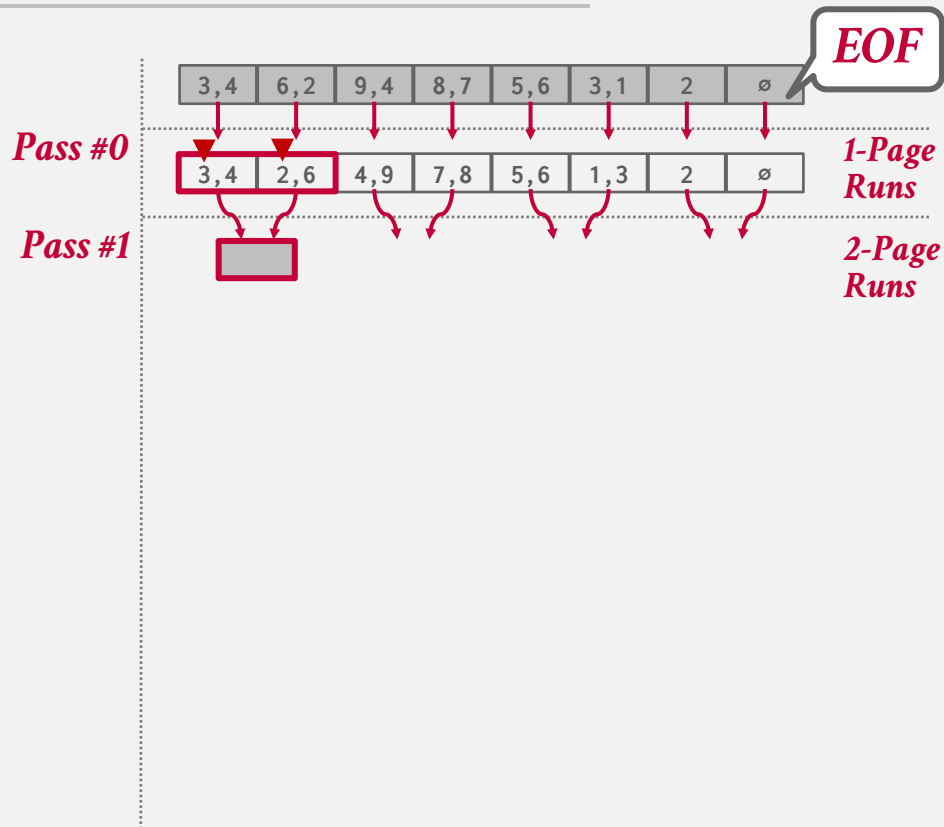
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

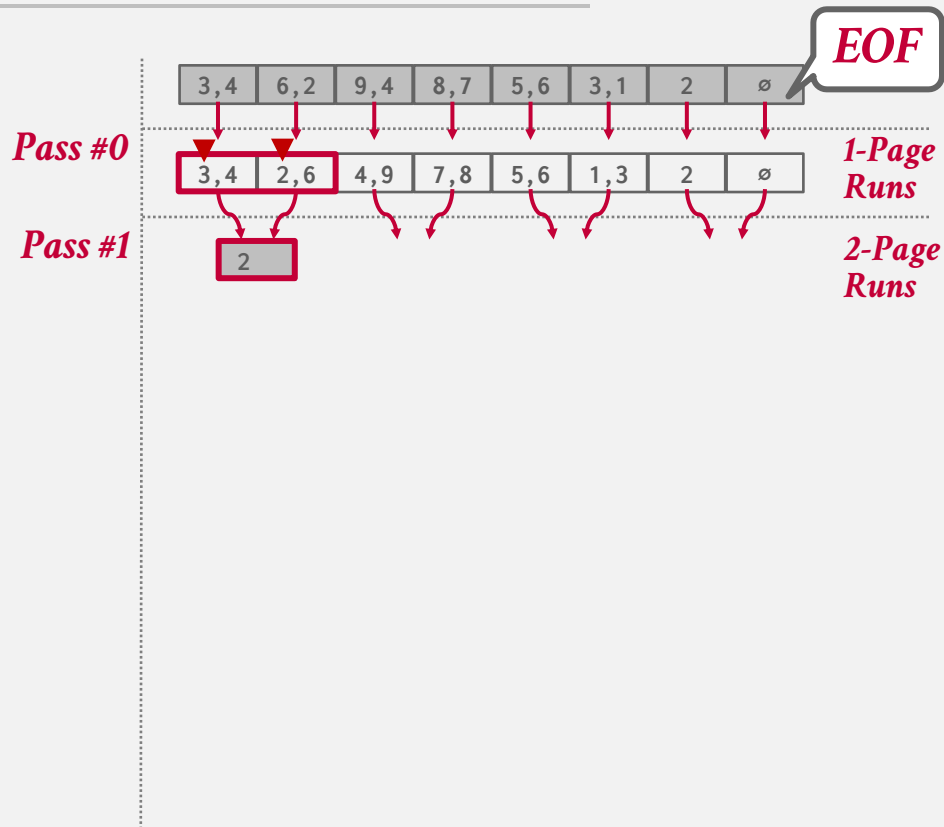
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

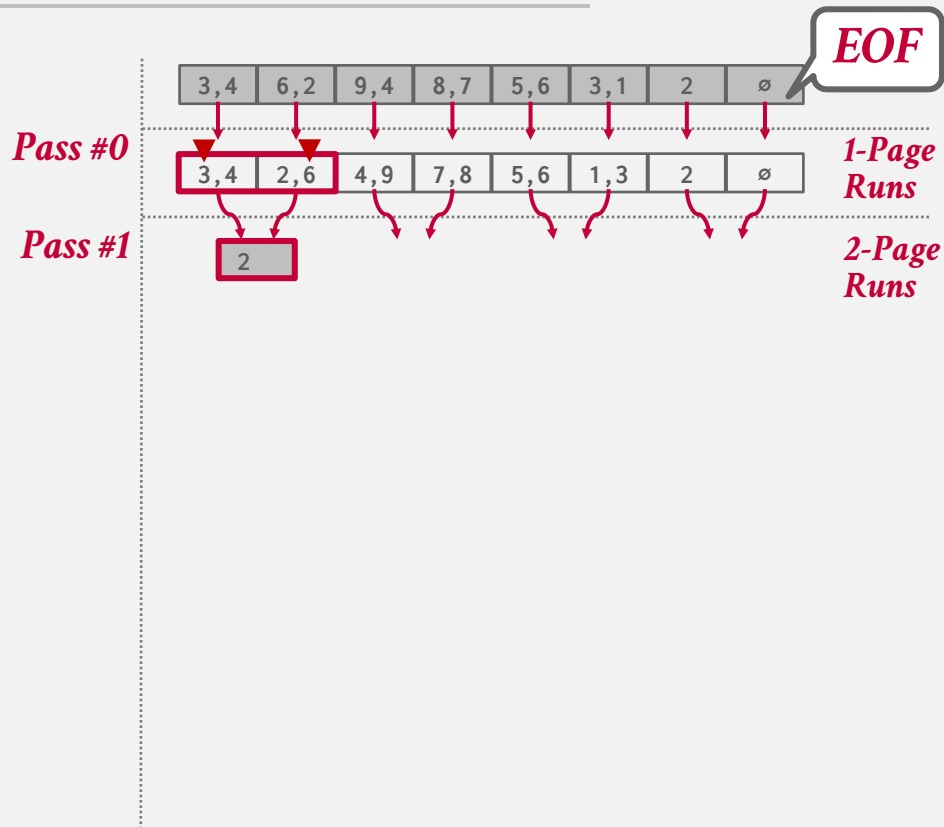
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

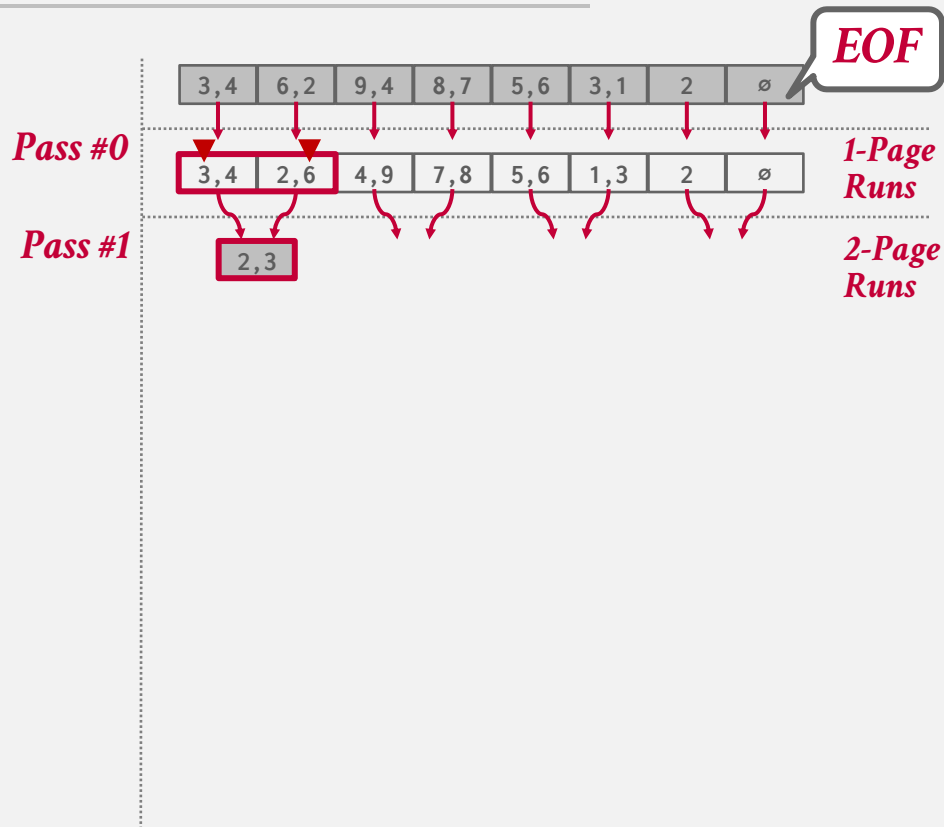
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

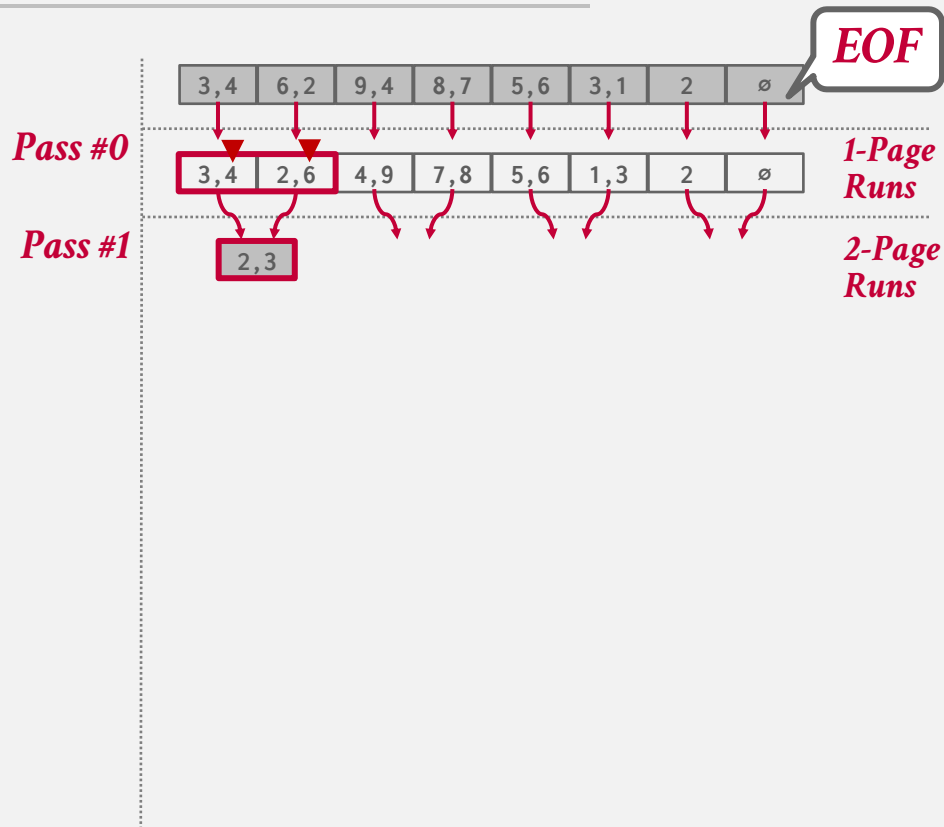
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

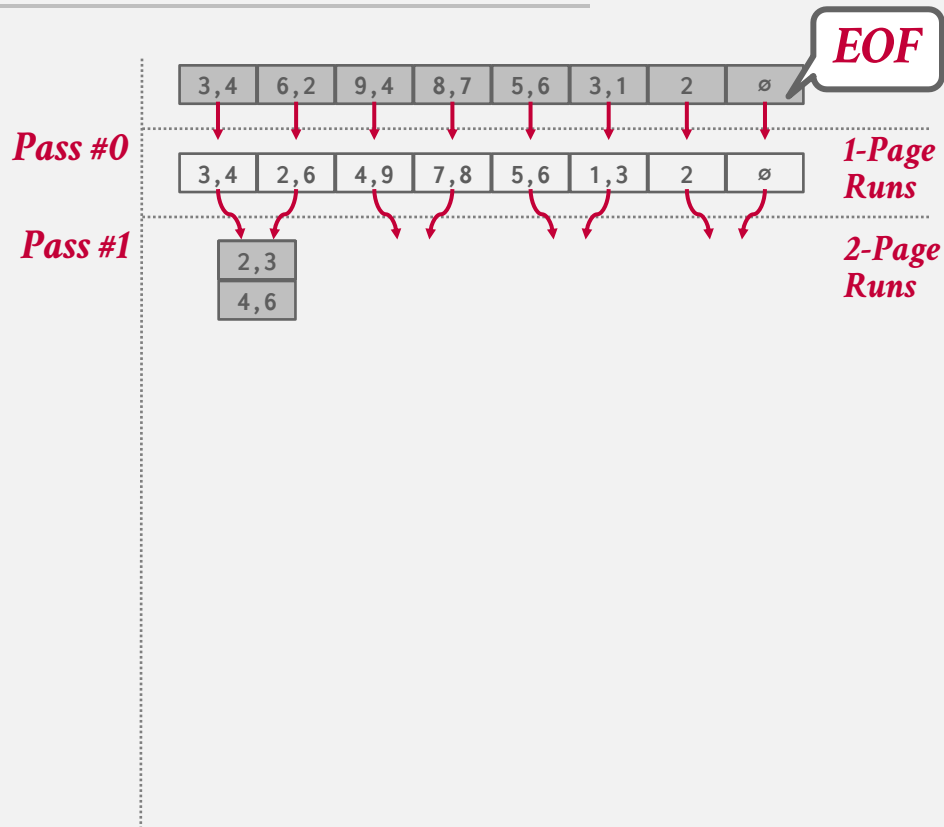
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

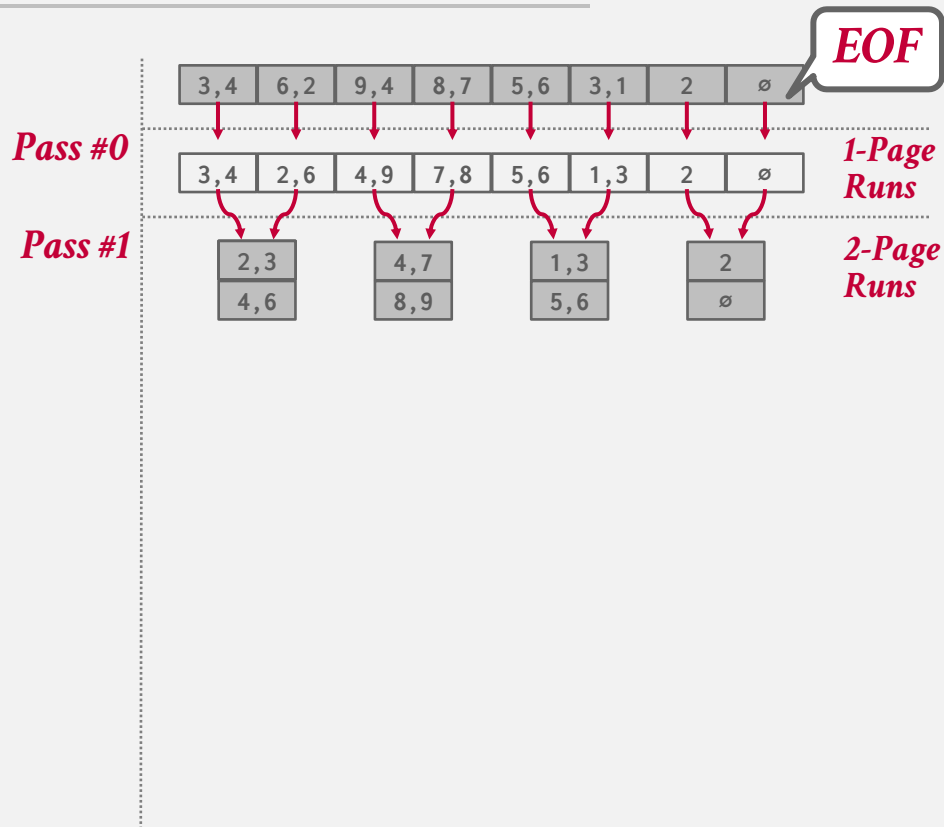
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$





# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

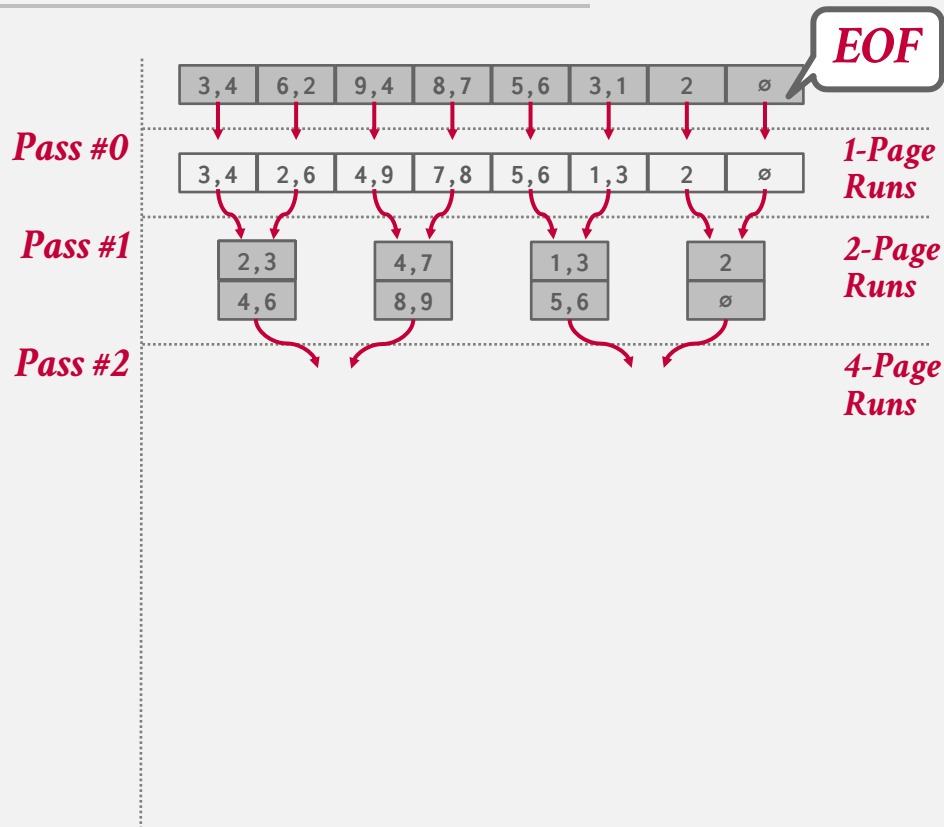
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

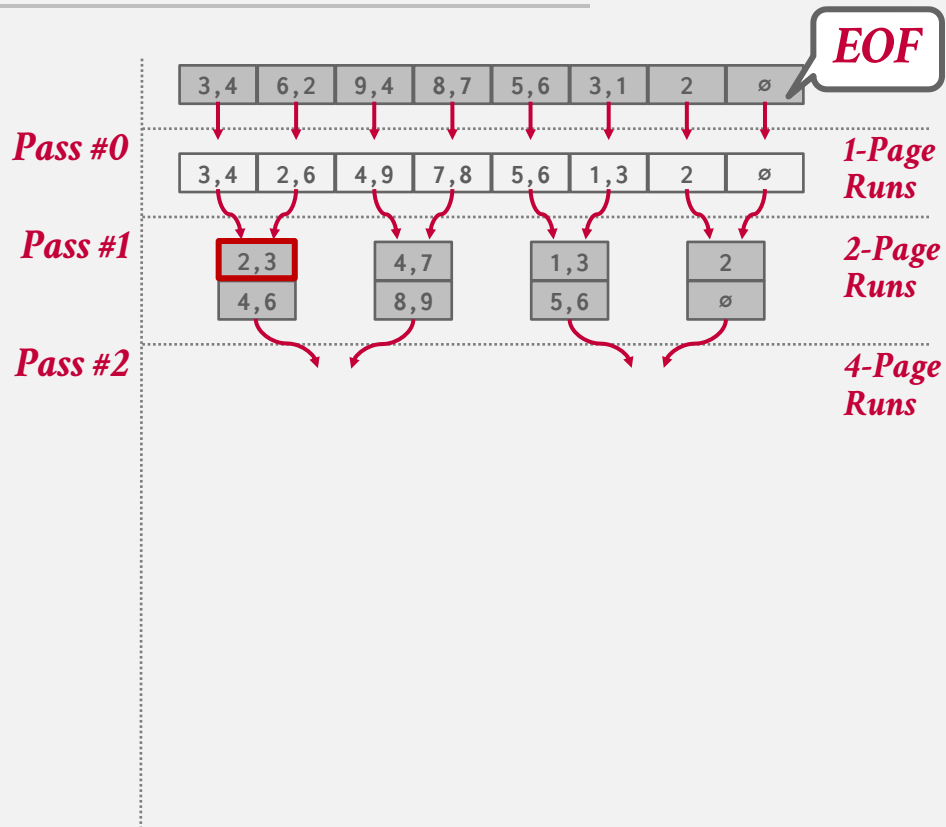
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

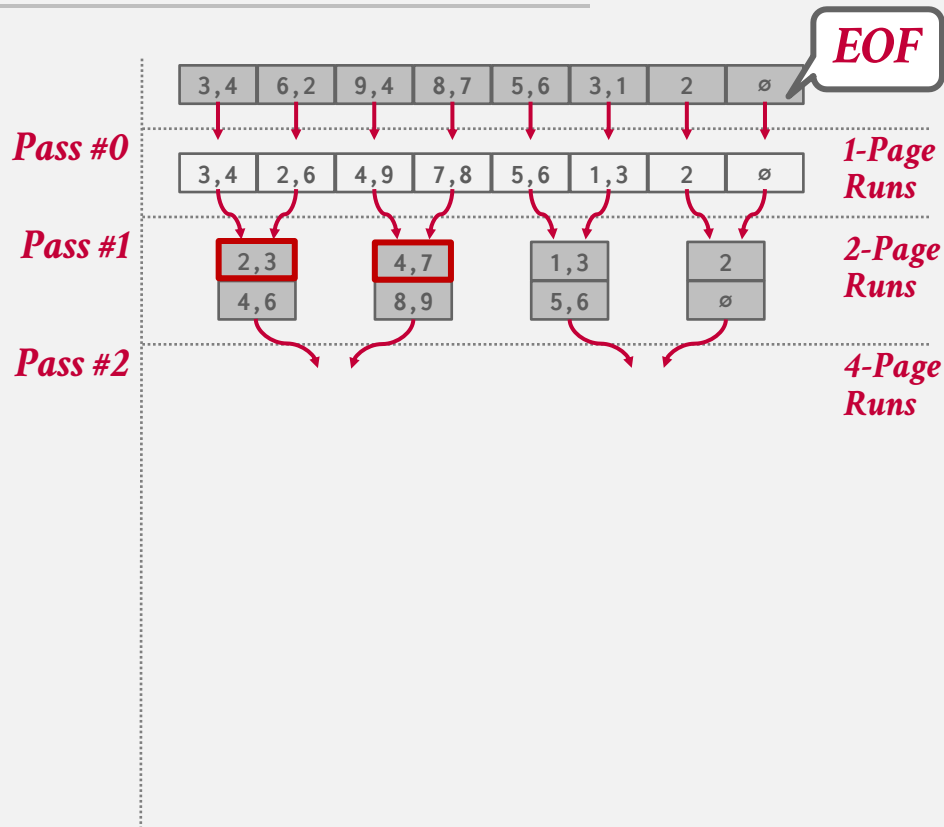
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

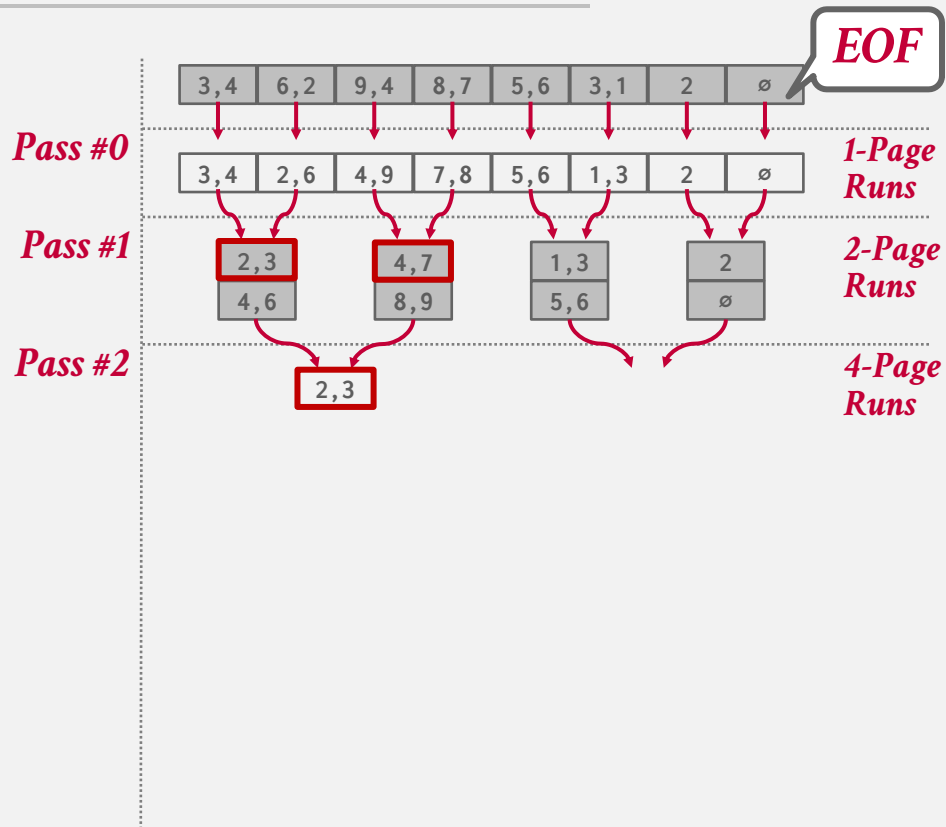
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

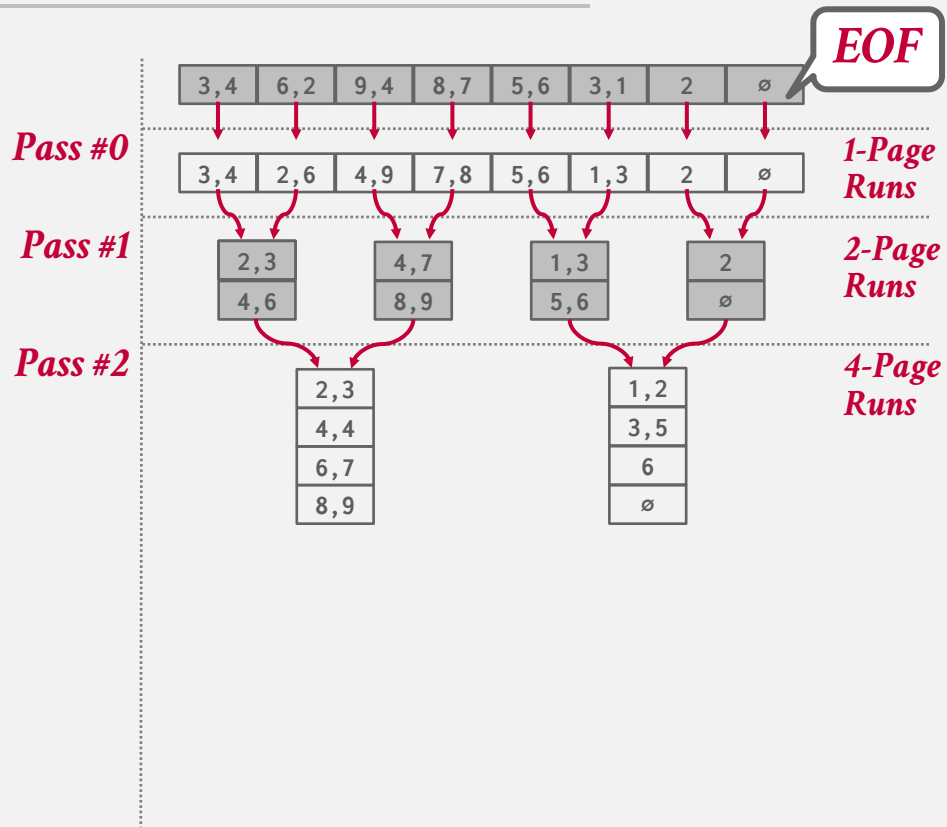
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

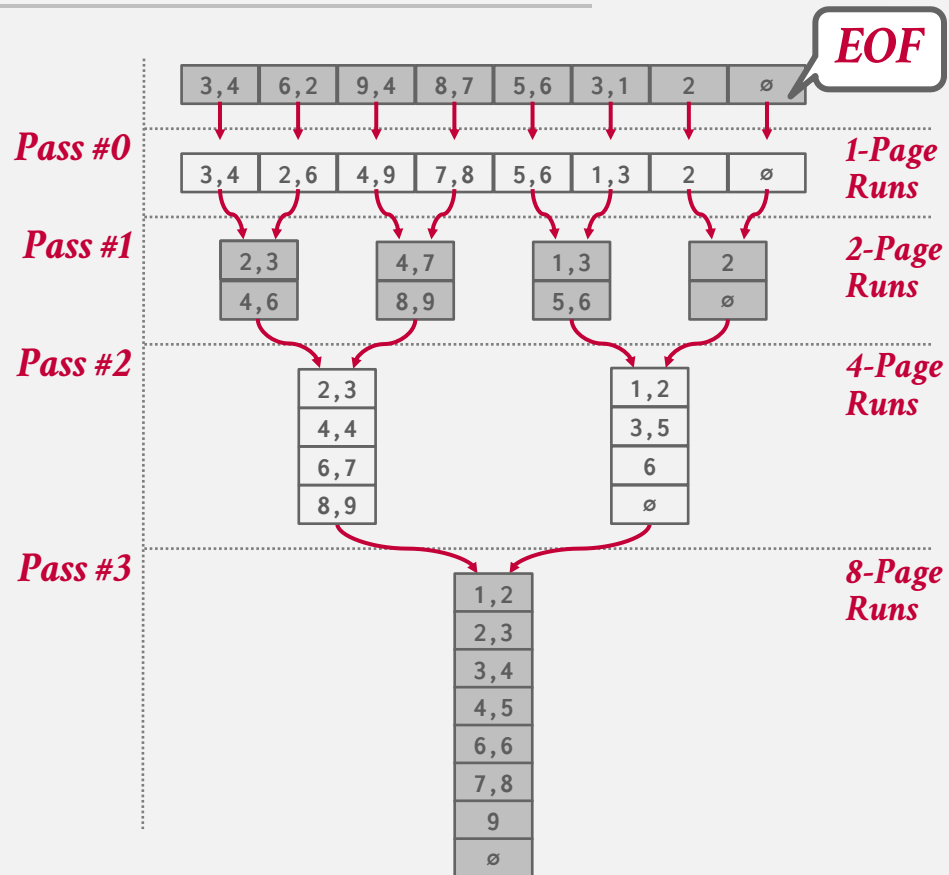
In each pass, we read and write every page in the file.

Number of passes

$$= 1 + \lceil \log_2 N \rceil$$

Total I/O cost

$$= 2N \cdot (\# \text{ of passes})$$



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

---

This algorithm only requires three buffer pool pages to perform the sorting ( $B=3$ ).

→ Two input pages, one output page

But even if we have more buffer space available ( $B>3$ ), it does not effectively utilize them if the worker must block on disk I/O...

# GENERAL EXTERNAL MERGE SORT

---

## Pass #0

- Use  $B$  buffer pages
- Produce  $\lceil N/B \rceil$  sorted runs of size  $B$

## Pass #1,2,3,...

- Merge  $B-1$  runs (i.e.,  $B$ -way merge)

Number of passes =  $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

Total I/O Cost =  $2N \cdot (\# \text{ of passes})$



# EXAMPLE

---

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages:  $N=108$ ,  $B=5$

- **Pass #0:**  $\lceil N / B \rceil = \lceil 108 / 5 \rceil = 22$  sorted runs of 5 pages each (last run is only 3 pages).
- **Pass #1:**  $\lceil N' / B-1 \rceil = \lceil 22 / 4 \rceil = 6$  sorted runs of 20 pages each (last run is only 8 pages).
- **Pass #2:**  $\lceil N'' / B-1 \rceil = \lceil 6 / 4 \rceil = 2$  sorted runs, first one has 80 pages and second one has 28 pages.
- **Pass #3:** Sorted file of 108 pages.

$$1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil = 1 + \lceil \log_4 22 \rceil = 1 + \lceil 2.229... \rceil = 4 \text{ passes}$$

# DOUBLE BUFFERING OPTIMIZATION

---

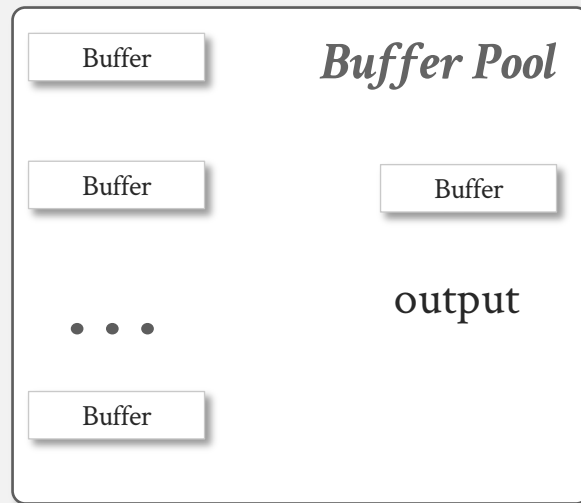
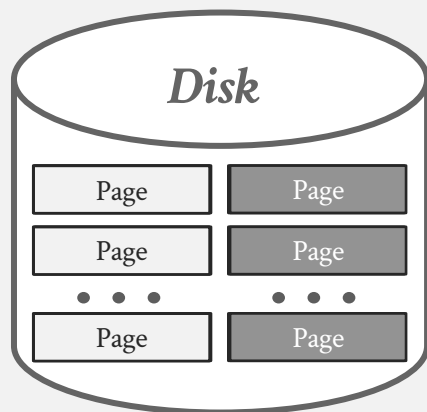
Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.

# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

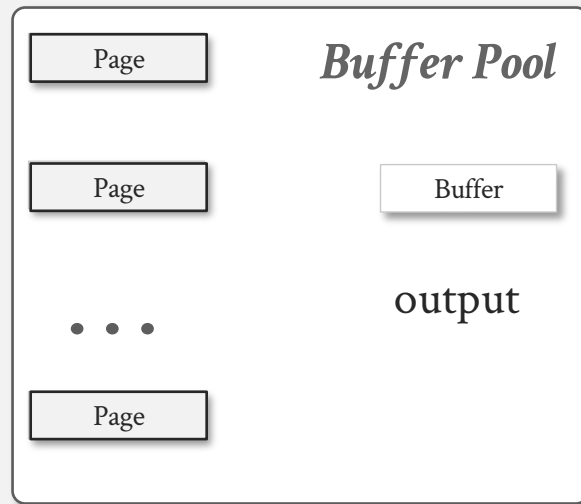
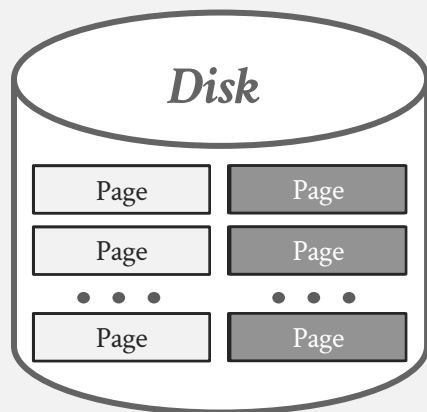
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

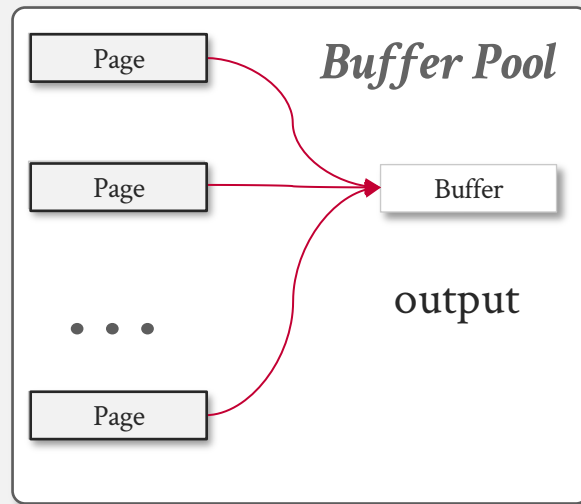
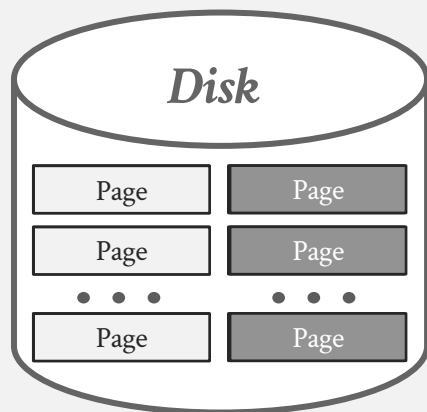
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

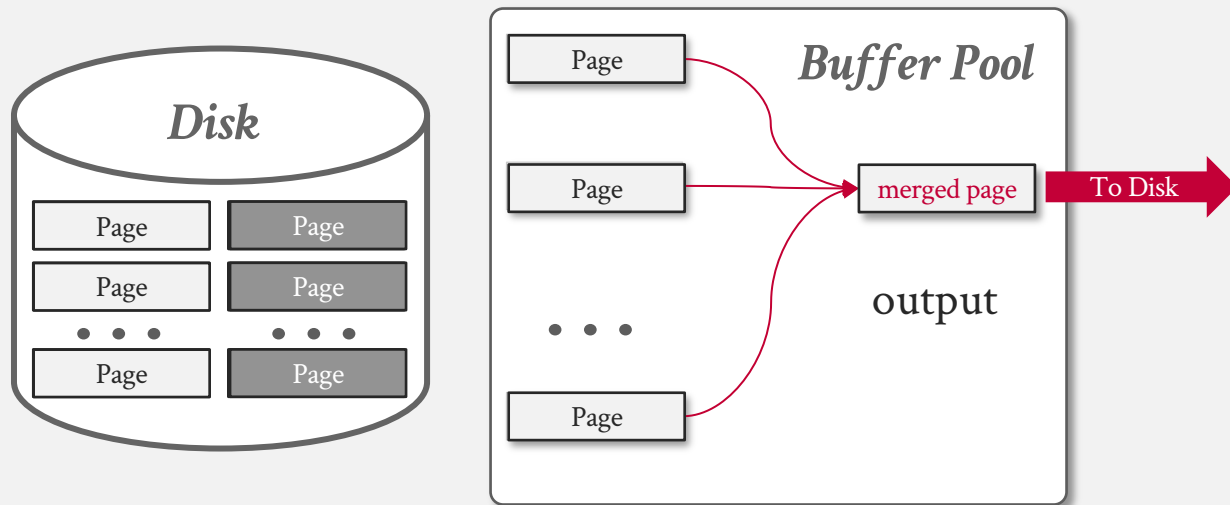
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

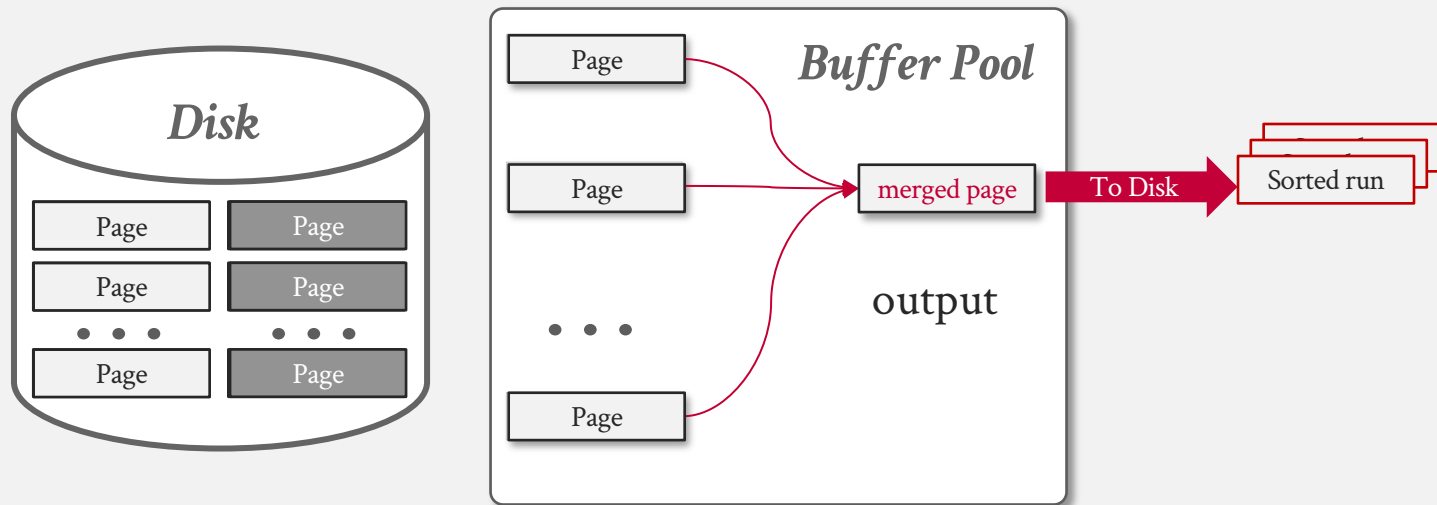
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

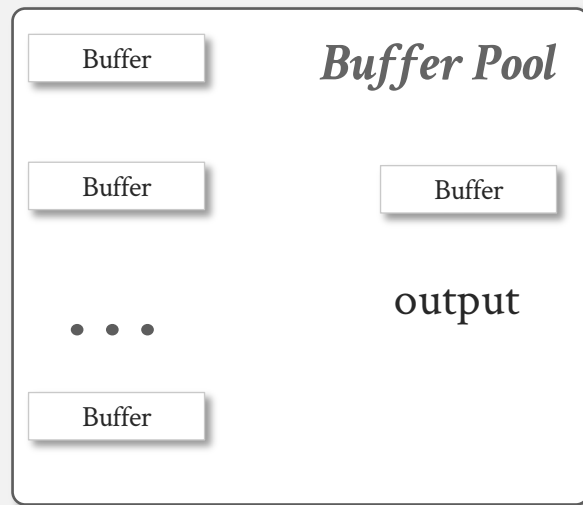
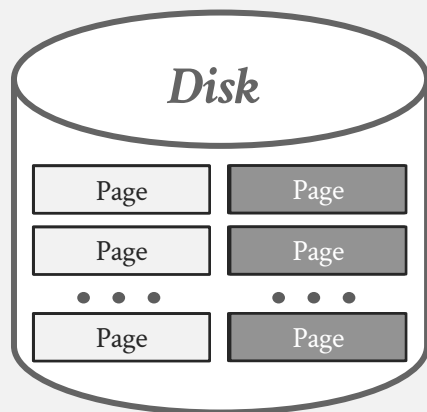
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

→ Overlap CPU and I/O operations

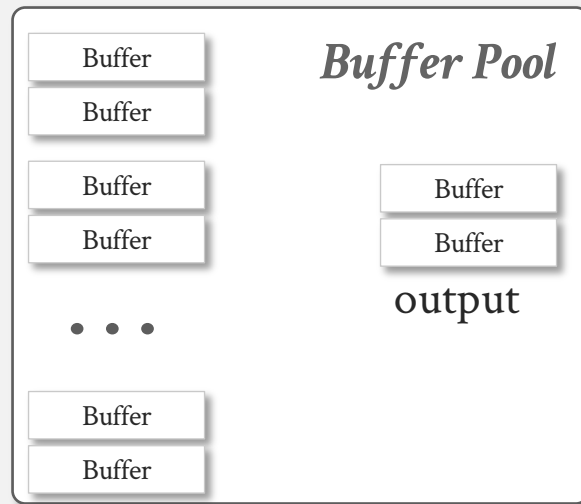
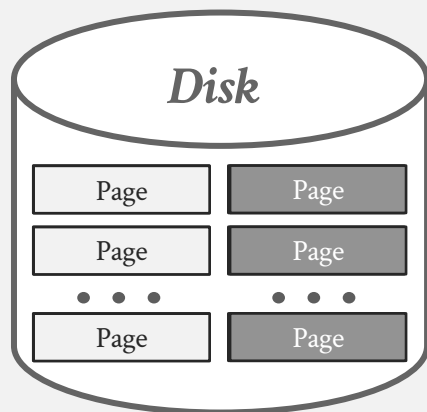




# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

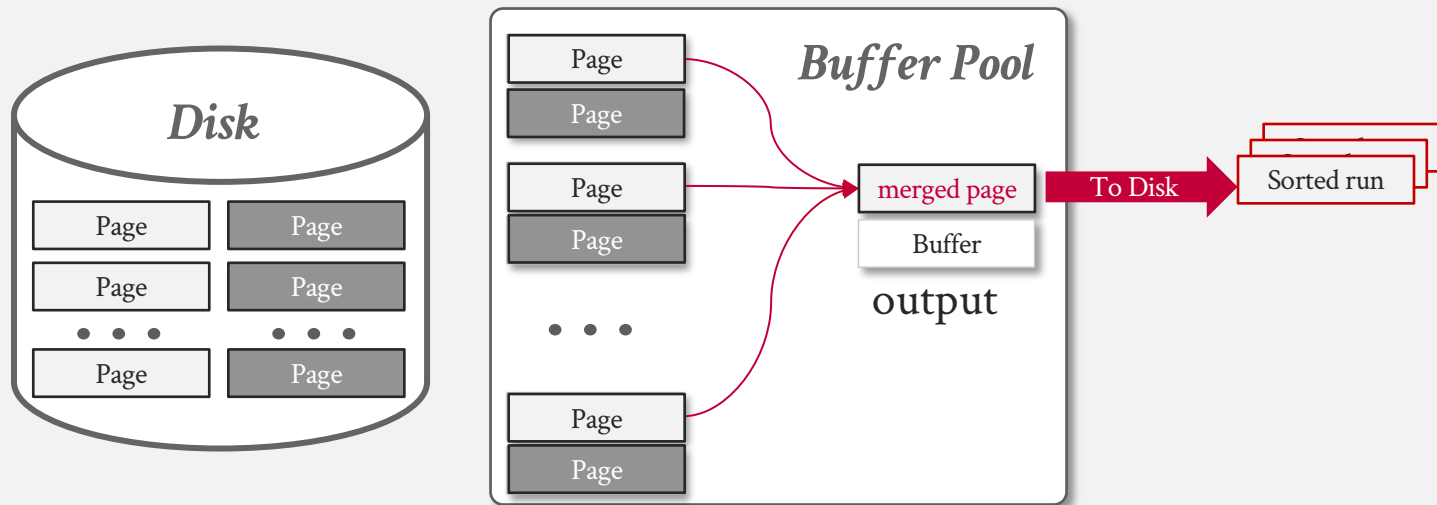
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

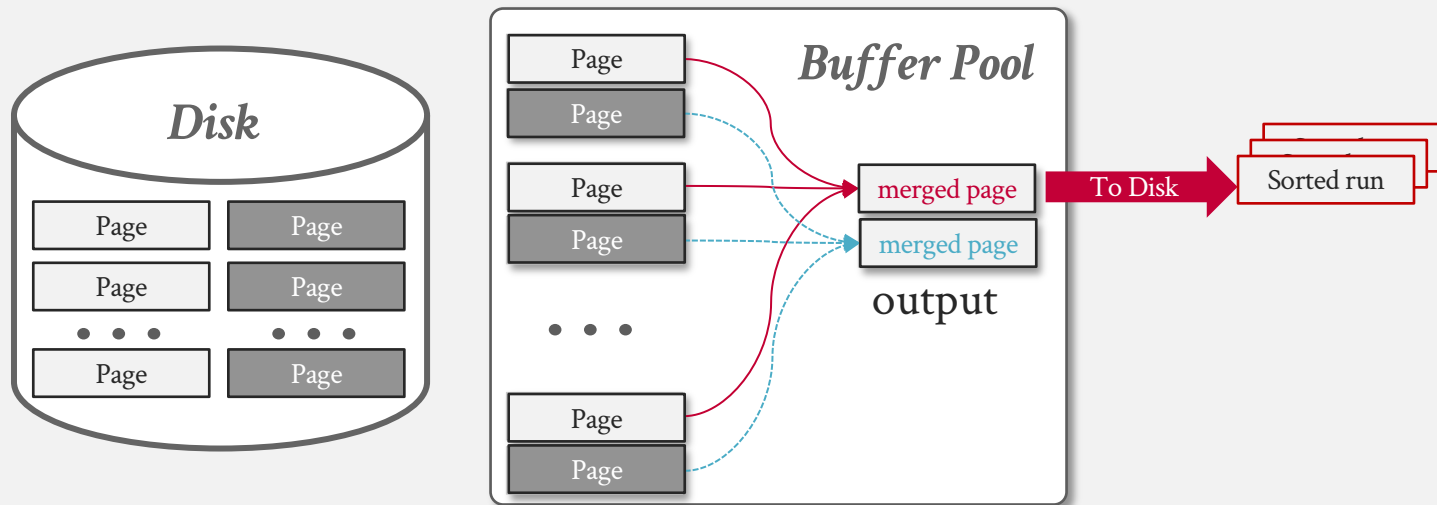
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

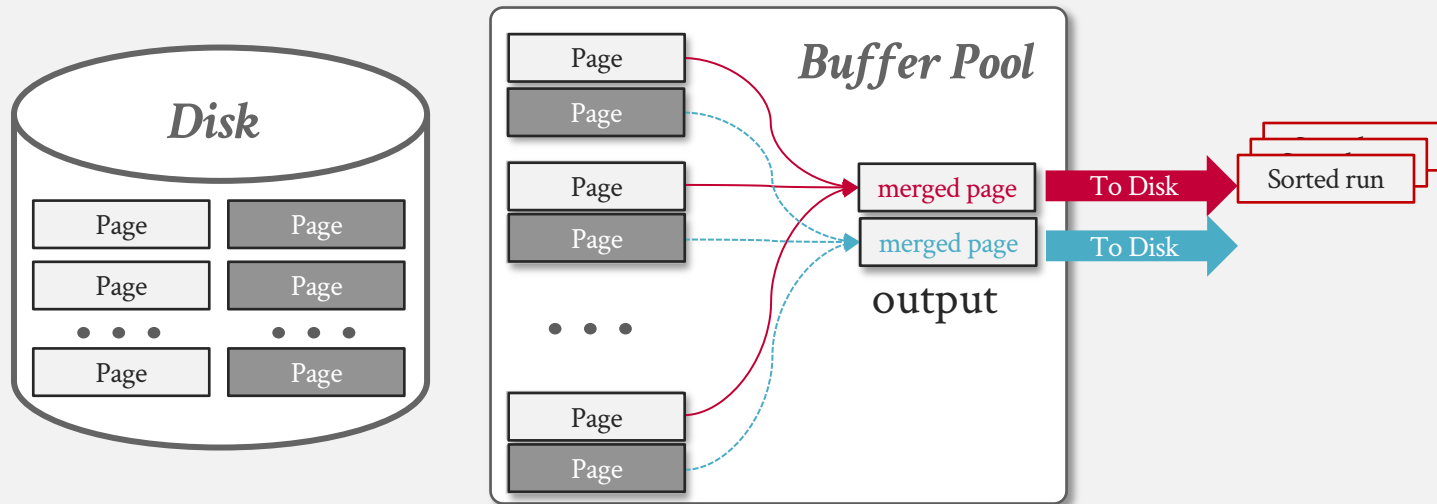
→ Overlap CPU and I/O operations



# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

→ Overlap CPU and I/O operations



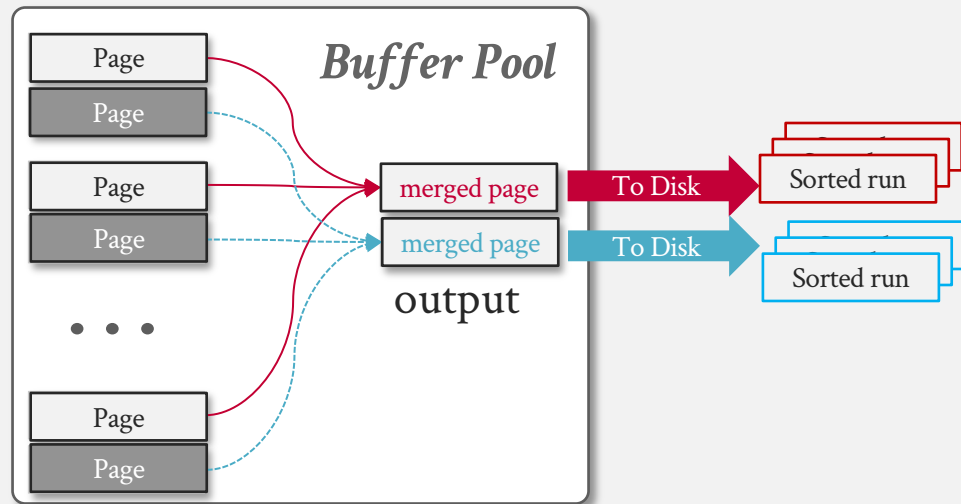
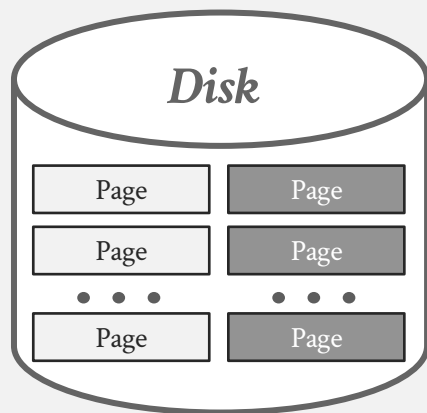
# DOUBLE BUFFERING

Prefetch the next run in the background and store it in a second buffer while the system is processing the current run.

→ Overlap CPU and I/O operations

Impact: reduces the effective “B” by half.

Reduces response time. What about throughput?



# COMPARISON OPTIMIZATIONS

---

## Approach #1: Code Specialization

→ Instead of providing a comparison function as a pointer to sorting algorithm, create a hardcoded version of sort that is specific to a key type.

## Approach #2: Suffix Truncation

→ First compare a binary prefix of long **VARCHAR** keys instead of slower string comparison. Fallback to slower version if prefixes are equal.

# USING B+TREES FOR SORTING

---

If the table that must be sorted already has a B+Tree index on the sort attribute(s), then we can use that to accelerate sorting.

Retrieve tuples in desired sort order by simply traversing the leaf pages of the tree.

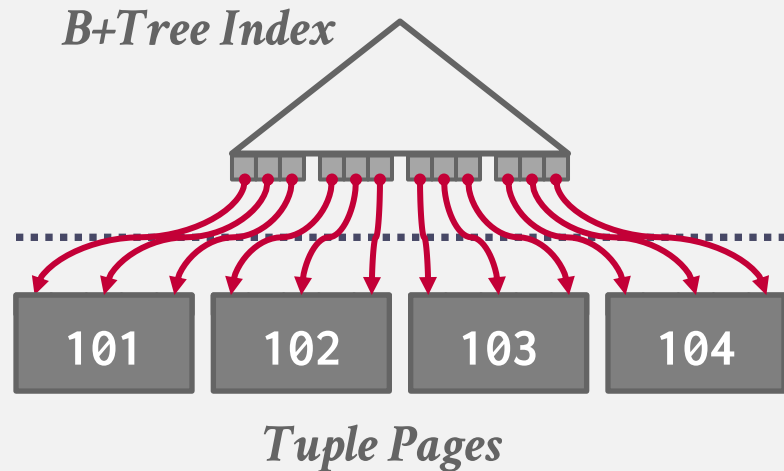
Cases to consider:

- Clustered B+Tree
- Unclustered B+Tree

# CASE #1 - CLUSTERED B+TREE

Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.

This is always better than external sorting because there is no computational cost, and all disk access is sequential.

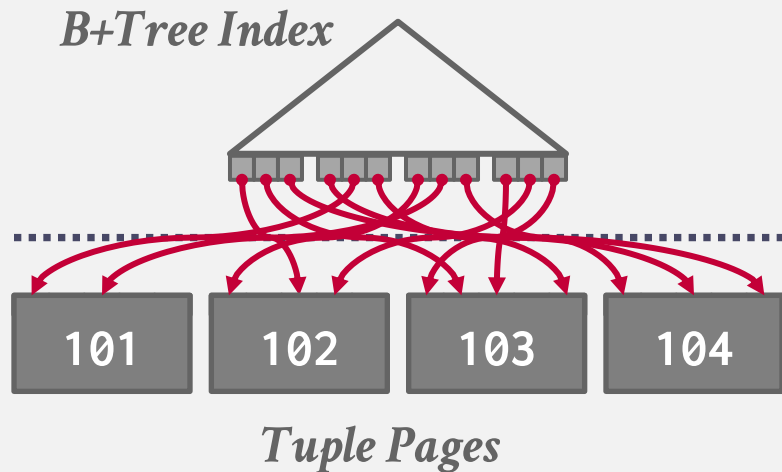




## CASE #2 - UNCLUSTERED B+TREE

Chase each pointer to the page that contains the data.

This is almost always a bad idea.  
In general, one I/O per data record.



# AGGREGATIONS

---

Collapse values for a single attribute from multiple tuples into a single scalar value.

The DBMS needs a way to quickly find tuples with the same distinguishing attributes for grouping.

Two implementation choices:

- Sorting
- Hashing

# SORTING AGGREGATION

enrolled (sid, cid, grade)

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
ORDER BY cid
```

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

  
*Filter*

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

  
*Remove  
Columns*

cid
15-445
15-826
15-721
15-445

  
*Sort*

cid
15-445
15-445
15-721
15-826

# SORTING AGGREGATION

enrolled (sid, cid, grade)

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
ORDER BY cid
```

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

  
*Filter*


sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

  
*Remove  
Columns*

cid
15-445
15-826
15-721
15-445

  
*Sort*

cid
15-445
15-445
15-721
15-826

  
*Eliminate  
Duplicates*

# SORTING AGGREGATION

enrolled (sid, cid, grade)

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
ORDER BY cid
```

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

  
*Filter*


sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

  
*Remove  
Columns*

cid
15-445
15-826
15-721
15-445

  
*Sort*

cid
15-445
<del>15-445</del>
15-721
15-826

  
*Eliminate  
Duplicates*

# ALTERNATIVES TO SORTING

---

What if we do not need the data to be ordered?

- Forming groups in **GROUP BY** (no ordering)
- Removing duplicates in **DISTINCT** (no ordering)

Hashing is a better alternative in this scenario.

- Only need to remove duplicates, no need for ordering.
- Can be computationally cheaper than sorting.

# HASHING AGGREGATE

---

Populate an ephemeral hash table as the DBMS scans the table. For each record, check whether there is already an entry in the hash table:

- **DISTINCT**: Discard duplicate
- **GROUP BY**: Perform aggregate computation

If everything fits in memory, then this is easy.

If the DBMS must spill data to disk, then we need to be smarter...

# EXTERNAL HASHING AGGREGATE

---

## Phase #1 – Partition

- Divide tuples into buckets based on hash key
- Write them out to disk when they get full

## Phase #2 – ReHash

- Build in-memory hash table for each partition and compute the aggregation



# PHASE #1 - PARTITION

---

Use a hash function  $h_1$  to split tuples into partitions on disk.

- A partition is one or more pages that contain the set of keys with the same hash value.
- Partitions are “spilled” to disk via output buffers.

Assume that we have  $B$  buffers.

We will use  $B-1$  buffers for the partitions and  $1$  buffer for the input data.

# PHASE #1 - PARTITION

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

⋮

*B-1 partitions*

**Filter**

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

⋮

**Remove Columns**

cid
15-445
15-826
15-721
15-445

⋮



15-445	15-445
15-445	15-312
15-312	15-445

15-826
15-210

⋮

15-721
--------

# PHASE #1 - PARTITION

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

⋮

*B-1 partitions*

**Filter**

sid	cid	grade
53666	15-445	C
53688	15-826	B
53666	15-721	C
53655	15-445	C

⋮

**Remove Columns**

cid
15-445
15-826
15-721
15-445

⋮



15-445	15-445
15-445	15-312
15-312	15-445

15-826
15-210

⋮

15-721
--------

# PHASE #1 - PARTITION

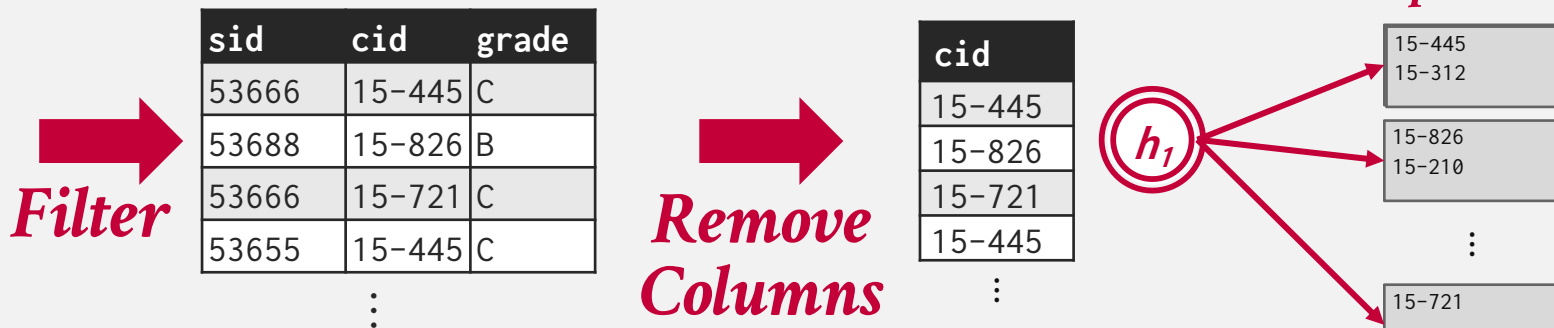
```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

⋮

*B-1 partitions*



## PHASE #2 - REHASH

---

For each partition on disk:

- Read it into memory and build an in-memory hash table based on a second hash function  $h_2$ .
- Then go through each bucket of this hash table to bring together matching tuples.

This assumes that each partition fits in memory.

# PHASE #2 - REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

## Phase #1 Buckets

*B-1  
Partitions*

15-445 15-445  
15-445 15-445  
15-445 15-445  
15-445 15-445

15-826  
15-826

⋮

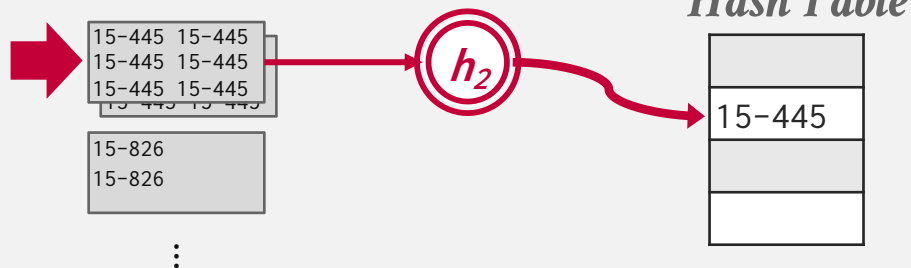
enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

# PHASE #2 - REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

*Phase #1 Buckets*



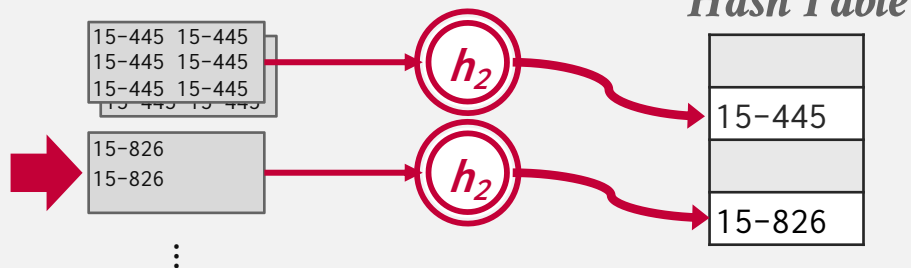
**enrolled (sid, cid, grade)**

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

# PHASE #2 - REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

*Phase #1 Buckets*



**enrolled (sid, cid, grade)**

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C



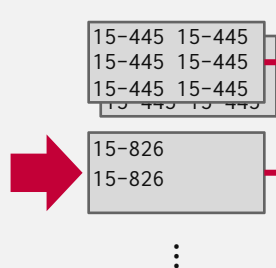
# PHASE #2 - REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
```

enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

*Phase #1 Buckets*



*Hash Table*

15-445
15-826

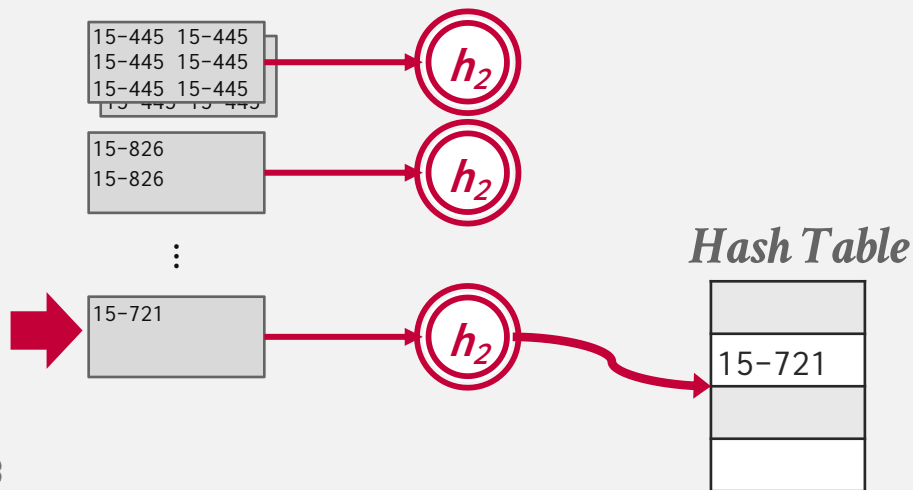
*Final Result*

cid
15-445
15-826

# PHASE #2 - REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

## Phase #1 Buckets



## enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

## Final Result

cid
15-445
15-826

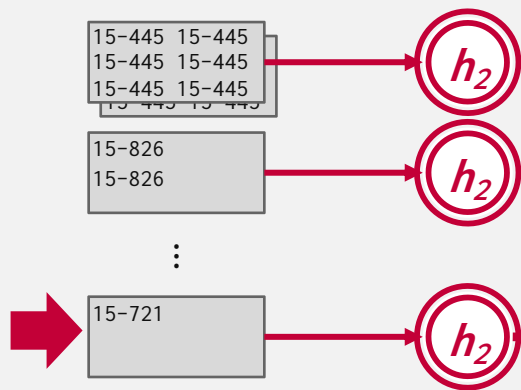
# PHASE #2 - REHASH

```
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
```

enrolled (sid, cid, grade)

sid	cid	grade
53666	15-445	C
53688	15-721	A
53688	15-826	B
53666	15-721	C
53655	15-445	C

*Phase #1 Buckets*



*Hash Table*

15-721

*Final Result*

cid
15-445
15-826
15-721

# HASHING SUMMARIZATION

---

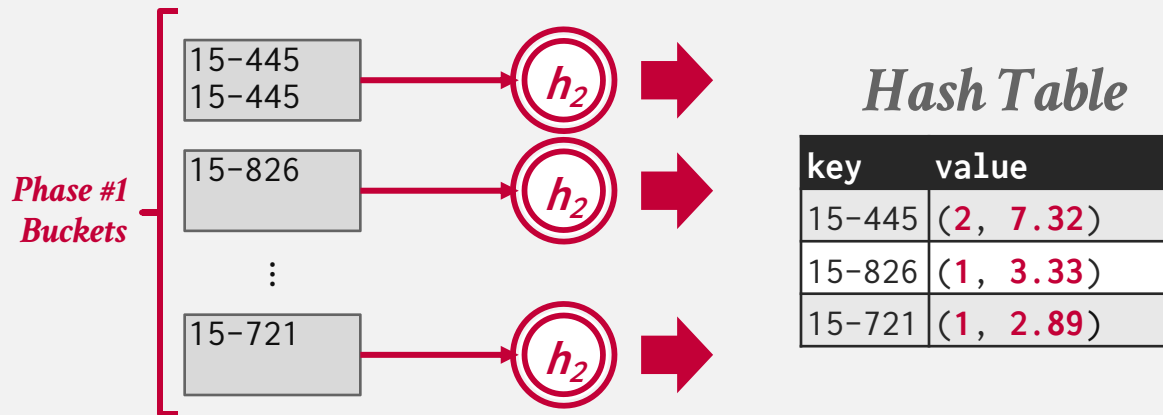
During the rehash phase, store pairs of the form  
(**GroupKey**→**RunningVal**)

When we want to insert a new tuple into the hash table:

- If we find a matching **GroupKey**, just update the **RunningVal** appropriately
- Else insert a new **GroupKey**→**RunningVal**

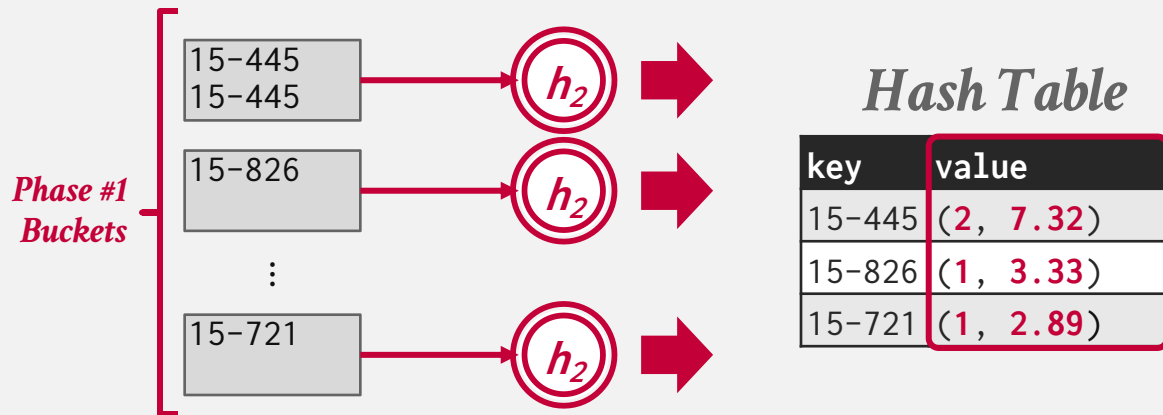
# HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```



# HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

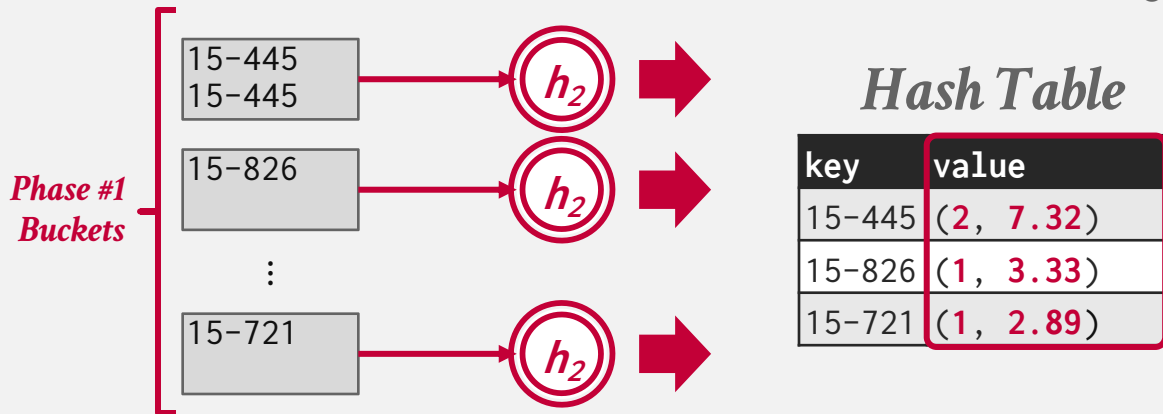


# HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

## Running Totals

AVG(col) → (COUNT, SUM)  
 MIN(col) → (MIN)  
 MAX(col) → (MAX)  
 SUM(col) → (SUM)  
 COUNT(col) → (COUNT)



# HASHING SUMMARIZATION

```
SELECT cid, AVG(s.gpa)
  FROM student AS s, enrolled AS e
 WHERE s.sid = e.sid
 GROUP BY cid
```

## Running Totals

AVG(col) → (COUNT, SUM)

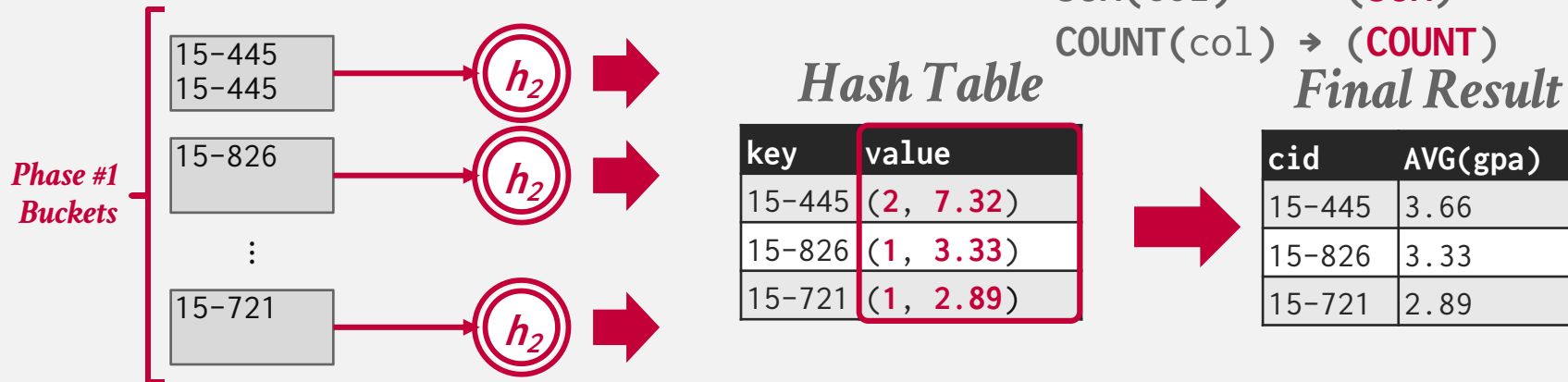
MIN(col) → (MIN)

MAX(col) → (MAX)

SUM(col) → (SUM)

COUNT(col) → (COUNT)

## Final Result





# CONCLUSION

---

Choice of sorting vs. hashing is subtle and depends on optimizations done in each case.

We already discussed the optimizations for sorting:

- Chunk I/O into large blocks to amortize costs
- Double-buffering to overlap CPU and I/O

# NEXT CLASS

---

Nested Loop Join

Sort-Merge Join

Hash Join