香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF DATA SCIENCE
數據科學學院

Ack: Prof. Jignesh Patel @ CMU
Prof. Andy Pavlo @CMU

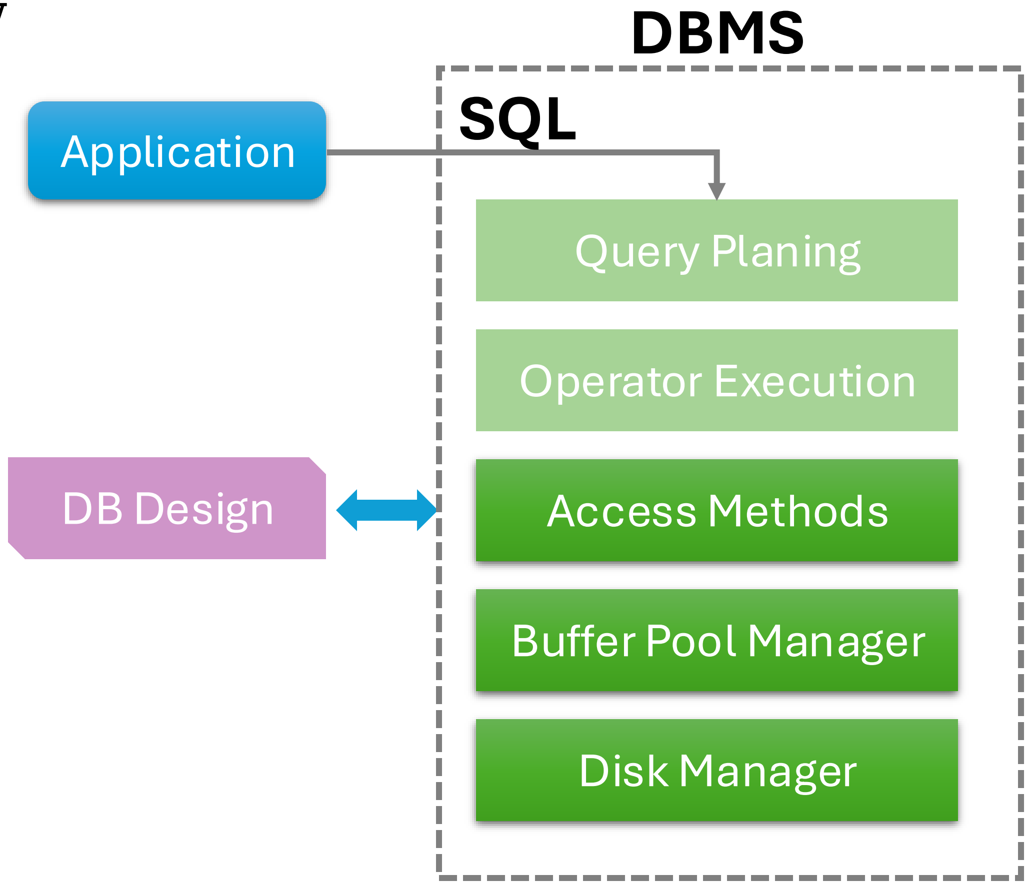
CSC3170
10: Sorting & Aggregations

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

Course Status

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

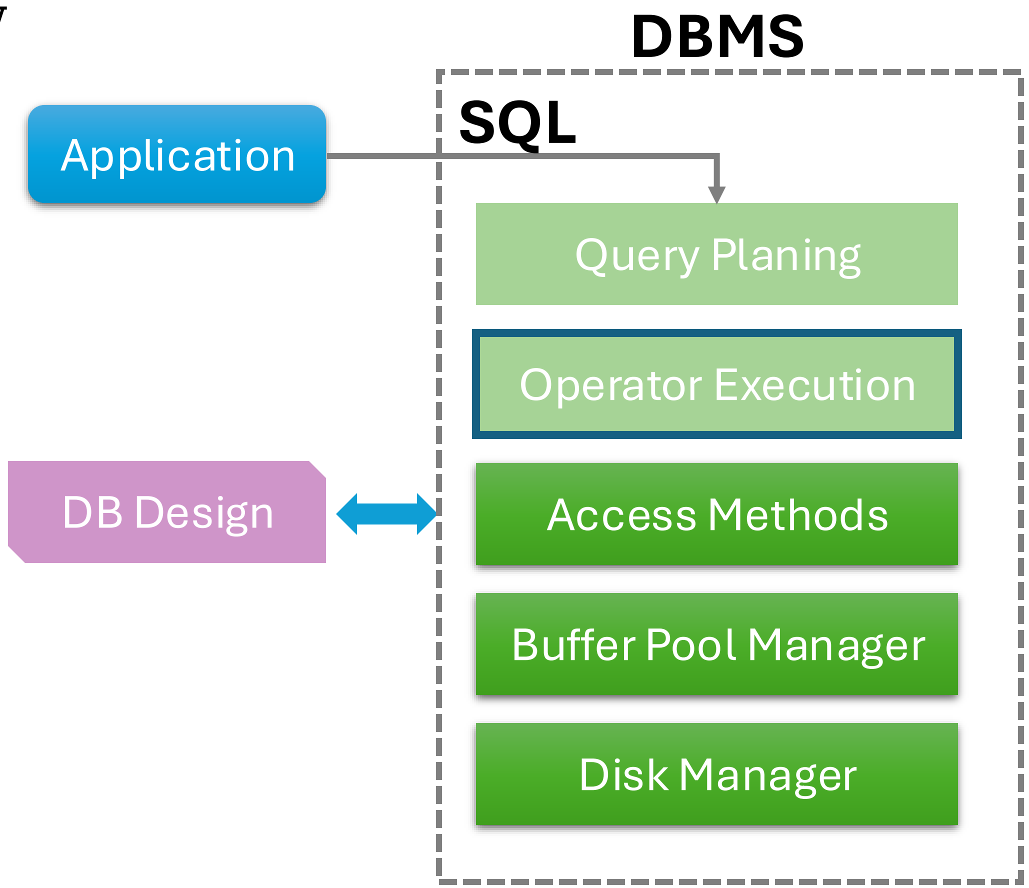
- Next several lectures:
  - Operator Algorithms
  - Query Processing Models
  - Runtime Architectures



Course Status

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

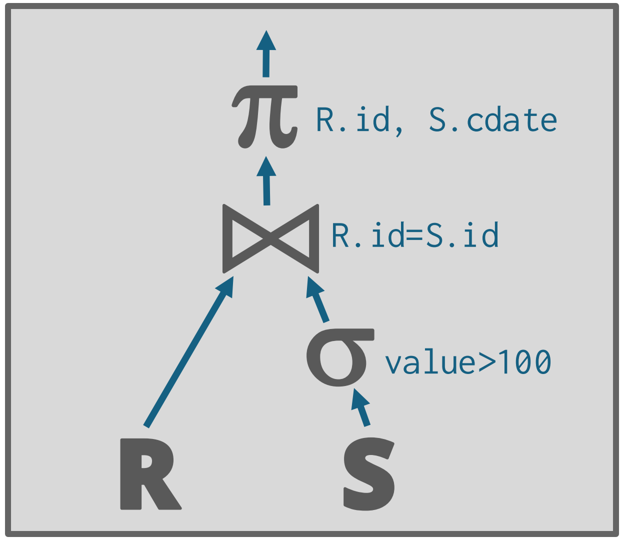
- Next several lectures:
  - Operator Algorithms
  - Query Processing Models
  - Runtime Architectures



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

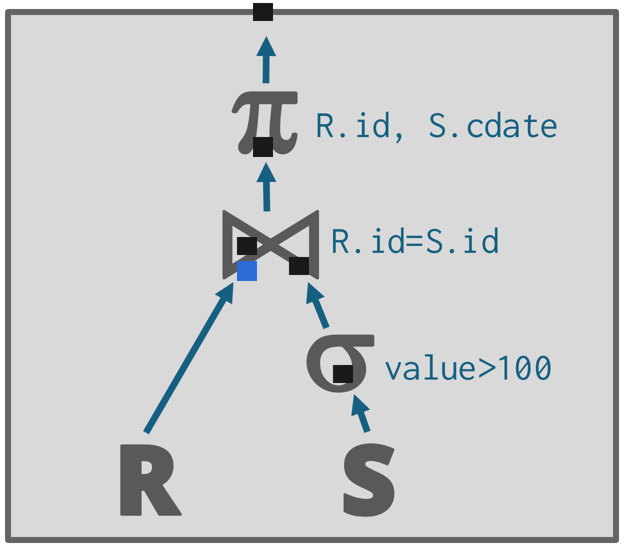
```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



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 later.
- 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).

SCHOOL OF DATA SCIENCE

In-Memory Sorting

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

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.

https://www.toptal.com/developers/sorting-algorithms
https://visualgo.net/en/sorting

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

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.

https://www.toptal.com/developers/sorting-algorithms
https://visualgo.net/en/sorting

- 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 …

This Lecture

- 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.

SCHOOL OF DATA SCIENCE

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
```

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

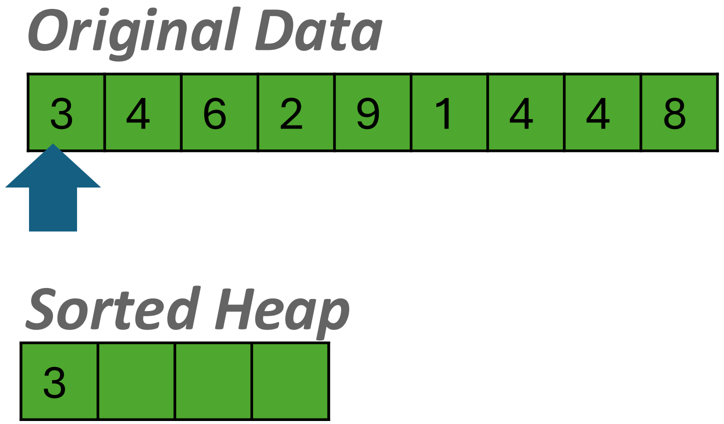
Sorted Heap

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
```

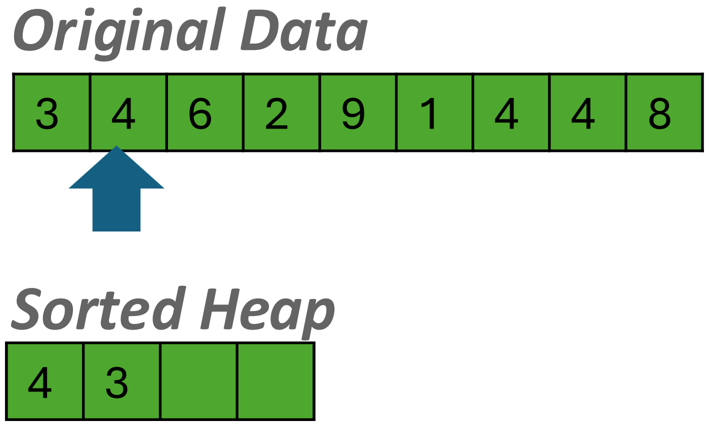


Top-N Heap Sort

- 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
```

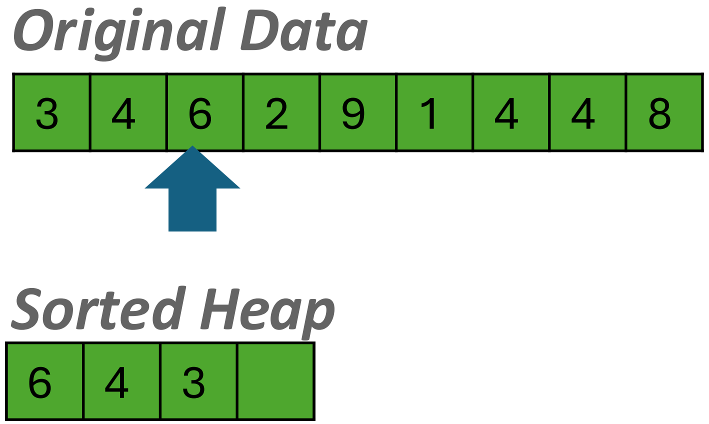


Top-N Heap Sort

- 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
```



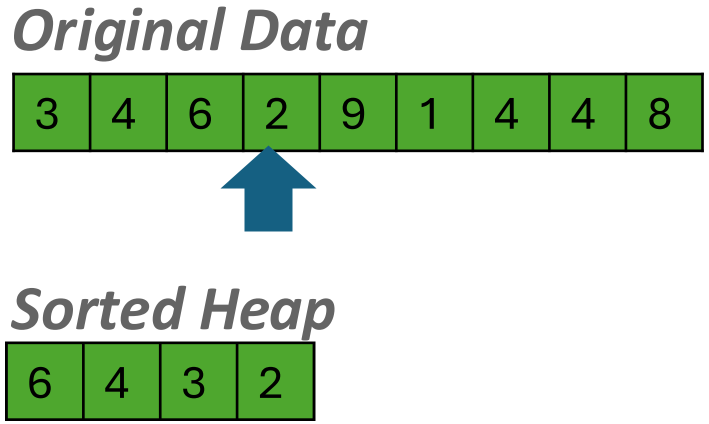
SCHOOL OF
DATA SCIENCE

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
```

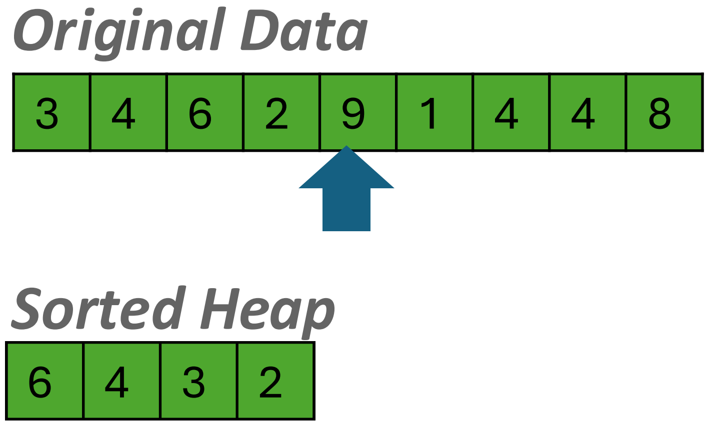


Top-N Heap Sort

- 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
```

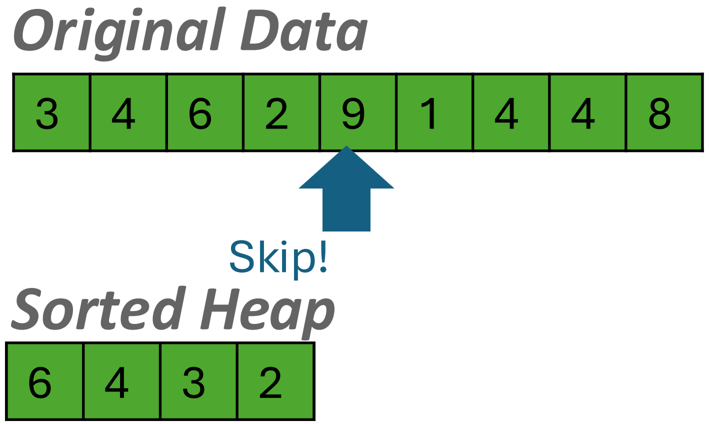


Top-N Heap Sort

- 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
```



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

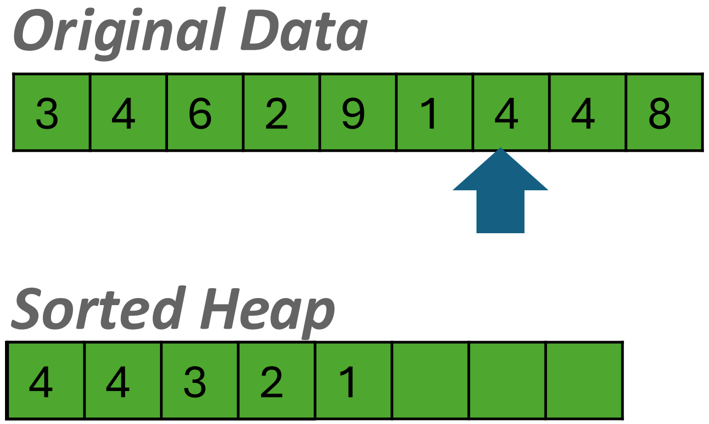
4	3	2	1

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
```

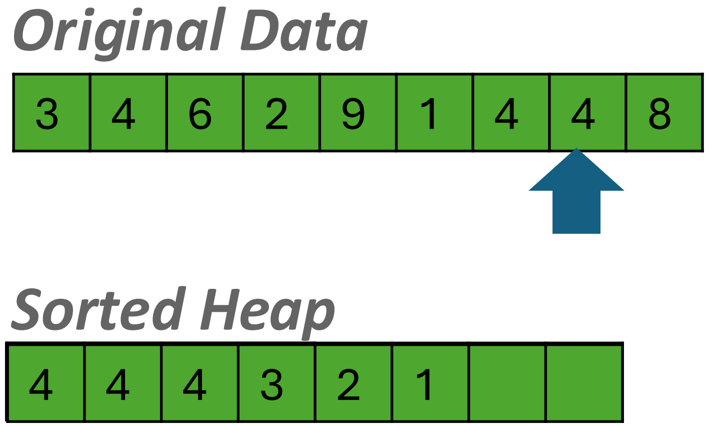


Top-N Heap Sort

- 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
```



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

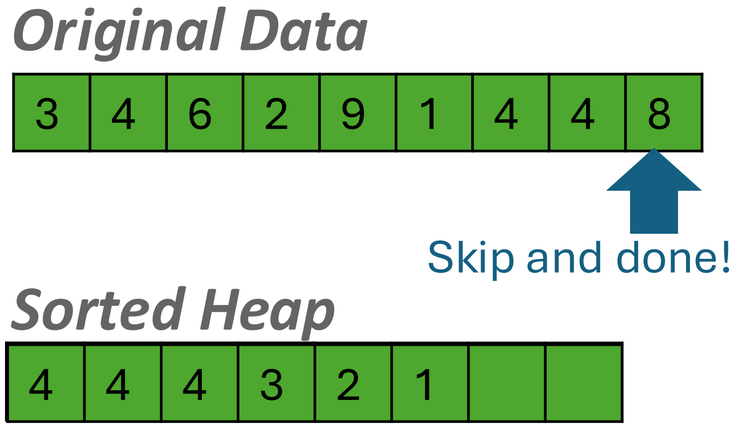
4	4	4	3	2	1		

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
```



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

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.

SCHOOL OF DATA SCIENCE

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***).

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

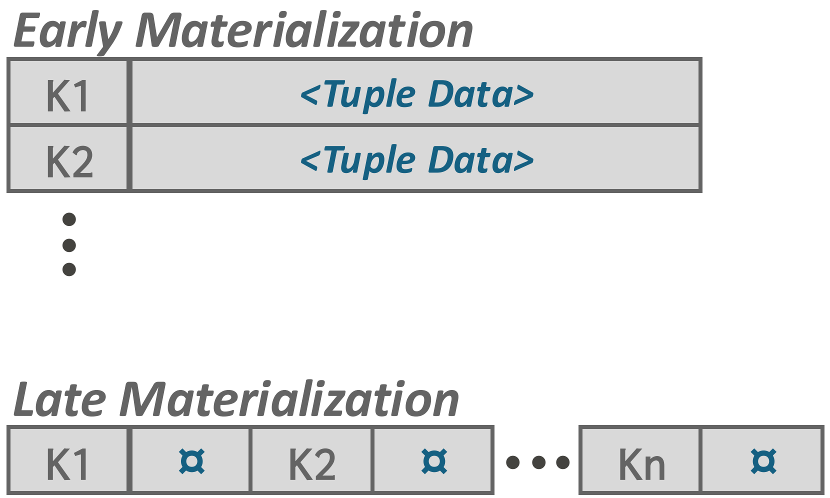
K1	*<Tuple Data>*
K2	*<Tuple Data>*

⋮

SCHOOL OF DATA SCIENCE

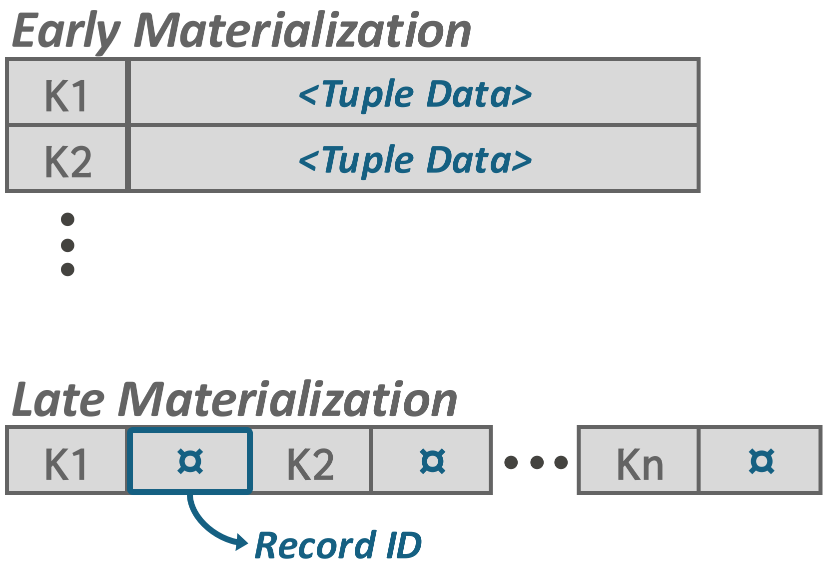
Sorted Run

- 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***).



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***).



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.

SCHOOL OF DATA SCIENCE

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.

- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost $= 2N \cdot (\text{\# of passes})$

SCHOOL OF DATA SCIENCE

Simplified 2-Way External Merge Sort

3,4	6,2	9,4	8,7	5,6	3,1	2	∅

- 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

3,4	6,2	9,4	8,7	5,6	3,1	2	∅

EOF

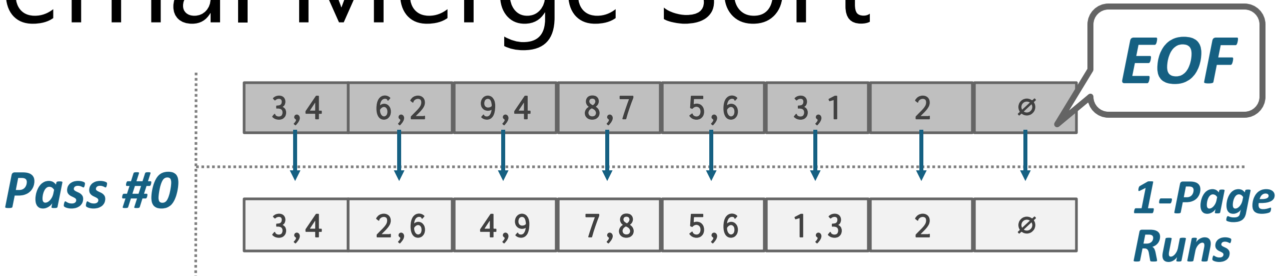
- In each pass, we read and write every page in the file.

- 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.

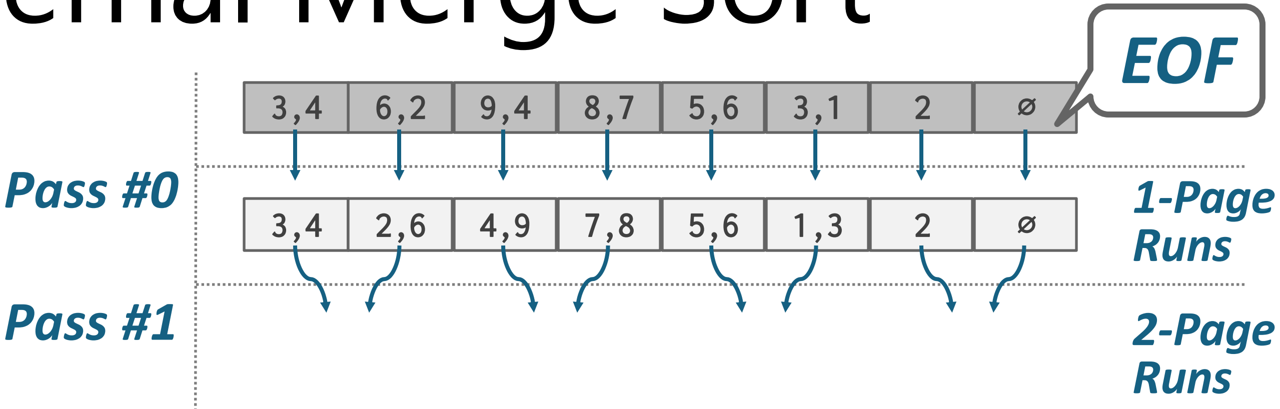
- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost $= 2N \cdot (\text{\# of passes})$



Simplified 2-Way External Merge Sort

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

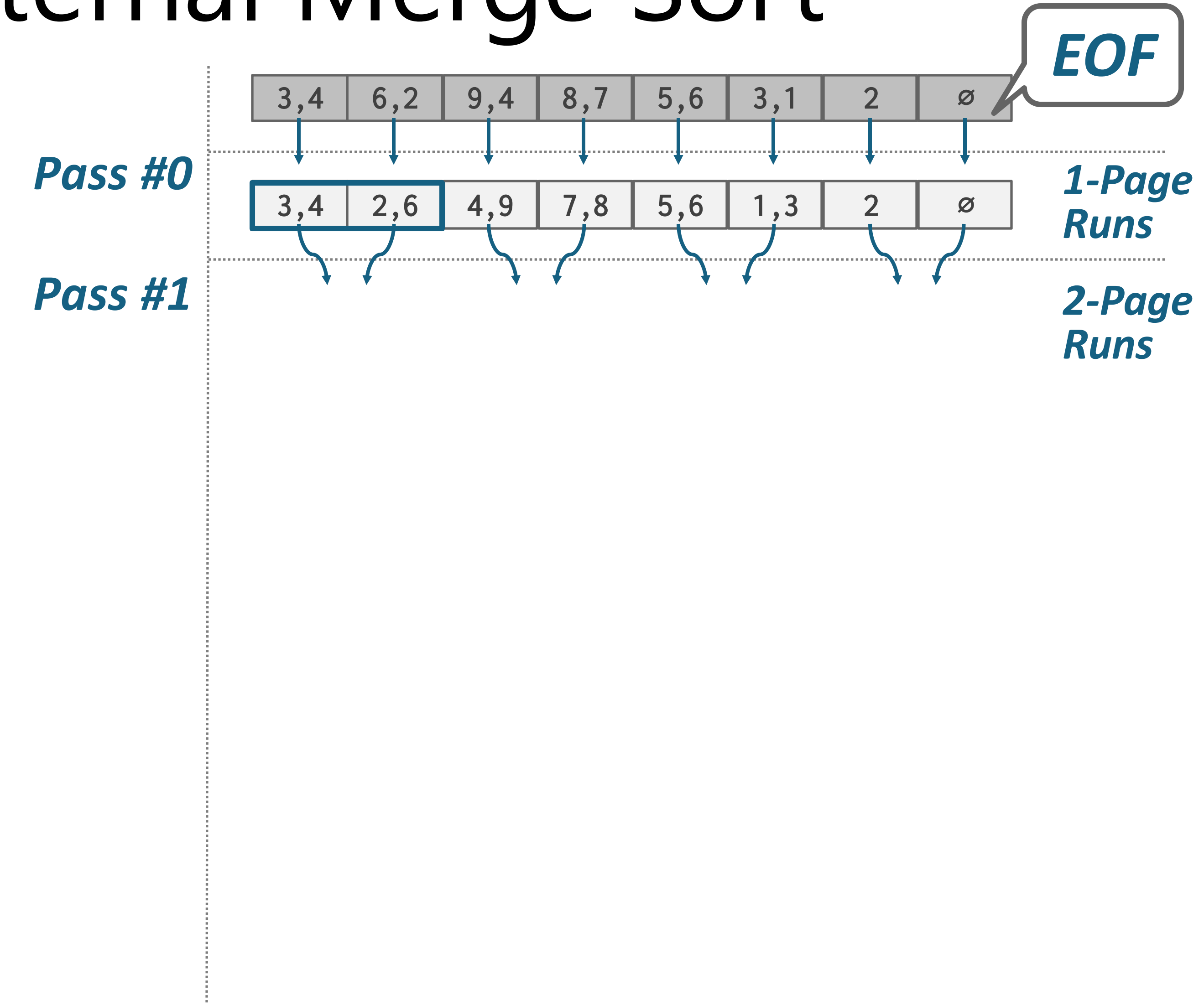
- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost $= 2N \cdot$ (# of passes)



Simplified 2-Way External Merge Sort

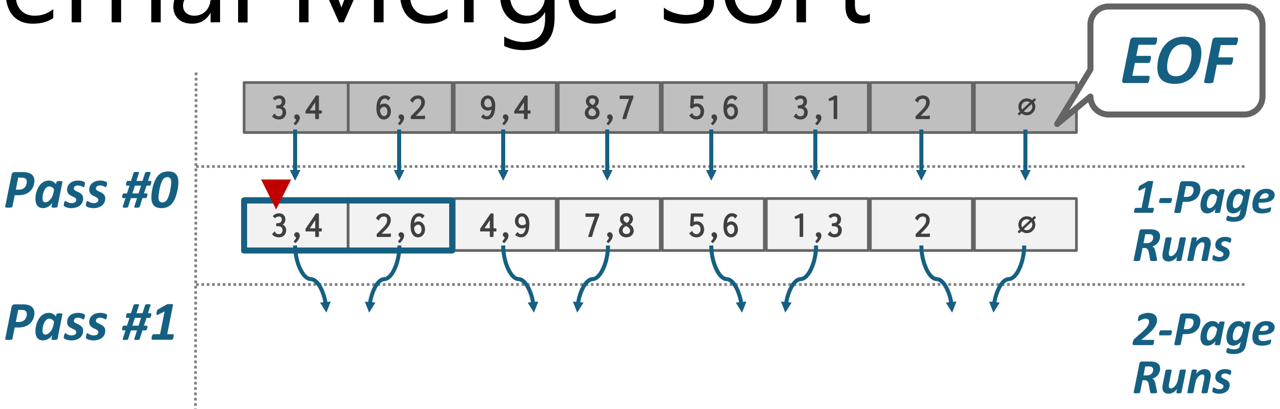
- In each pass, we read and write every page in the file.

- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost $= 2N \cdot$ (# of passes)



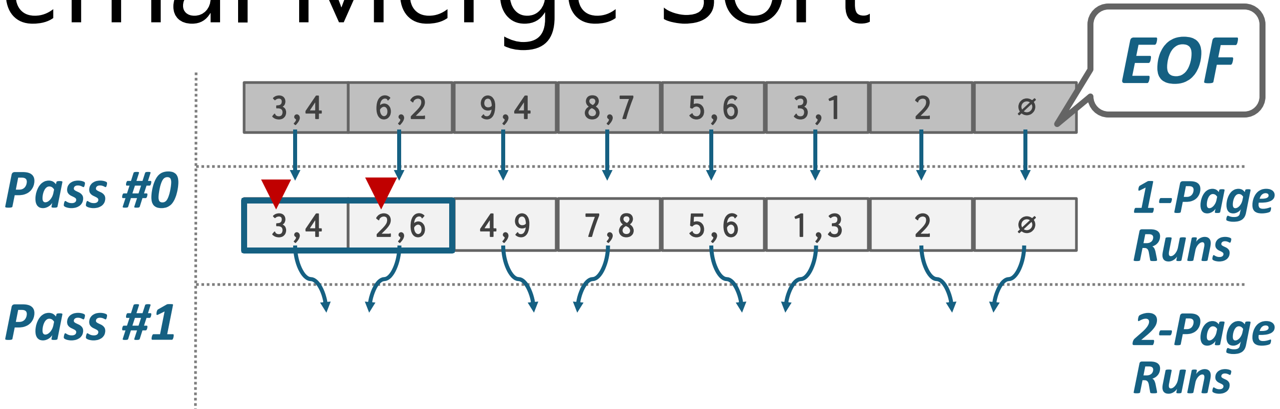
Simplified 2-Way External Merge Sort

- 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 $= 2N \cdot$ (# of passes)



Simplified 2-Way External Merge Sort

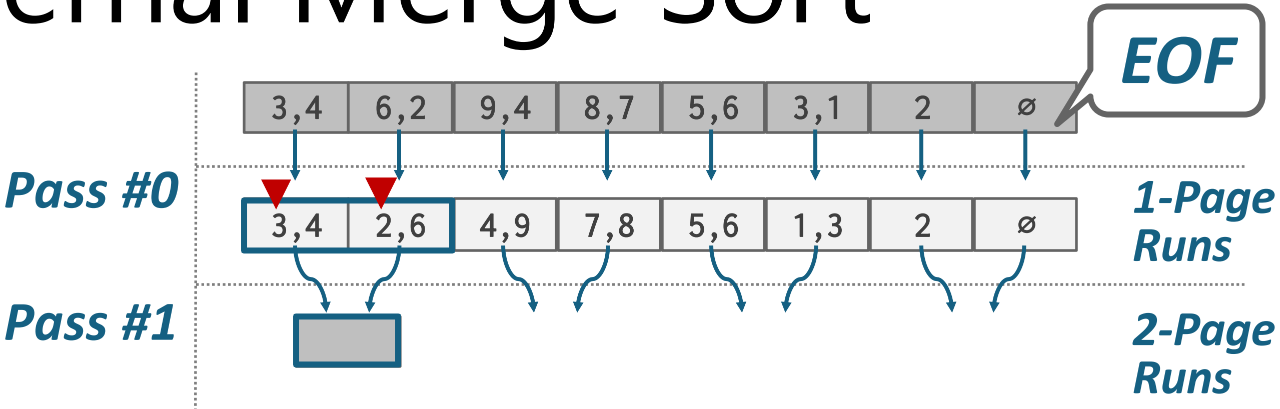
- In each pass, we read and write every page in the file.
- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost $= 2N \cdot (\text{\# of passes})$



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 $= 2N \cdot$ (# of passes)

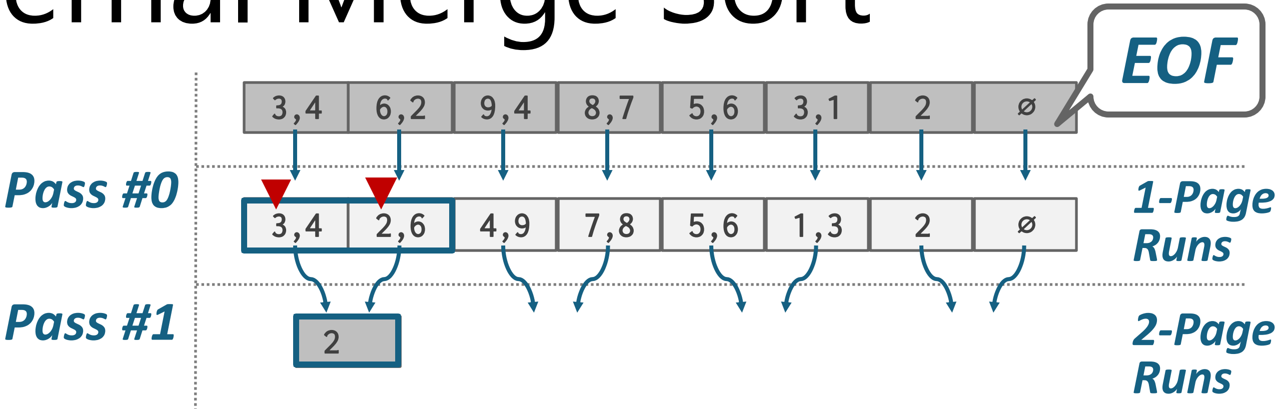


Simplified 2-Way External Merge Sort

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

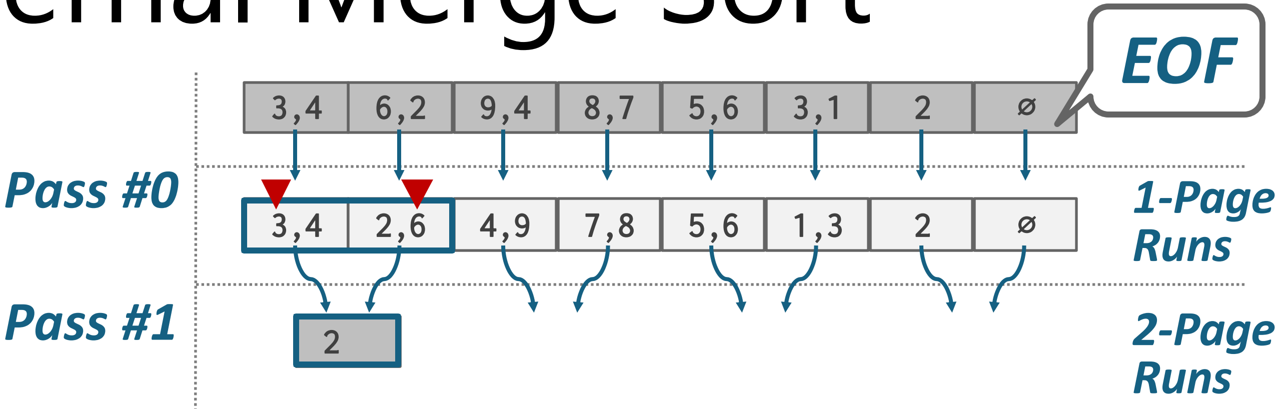
- Number of passes
  $= 1 + \lceil \log_2 N \rceil$

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



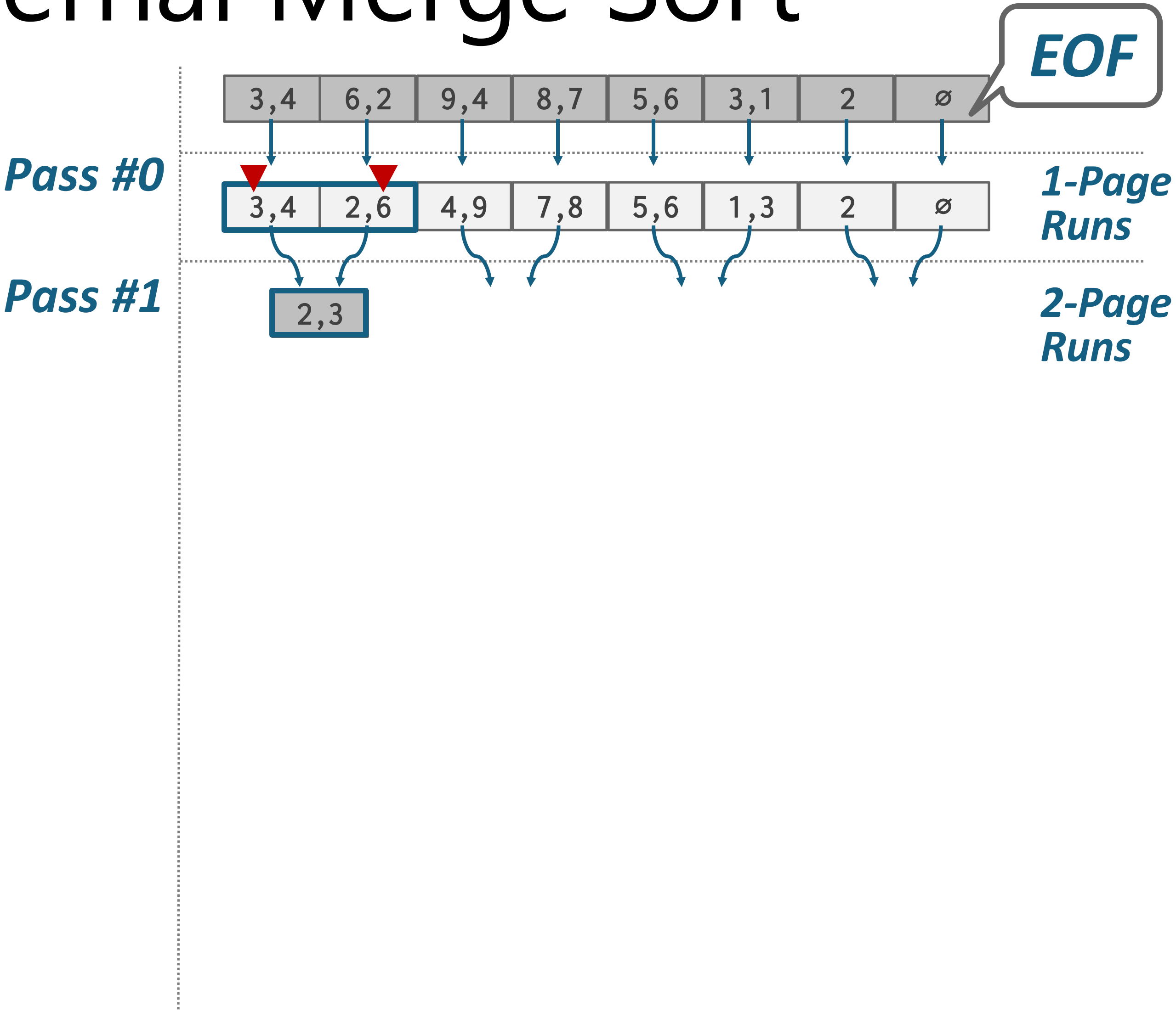
Simplified 2-Way External Merge Sort

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



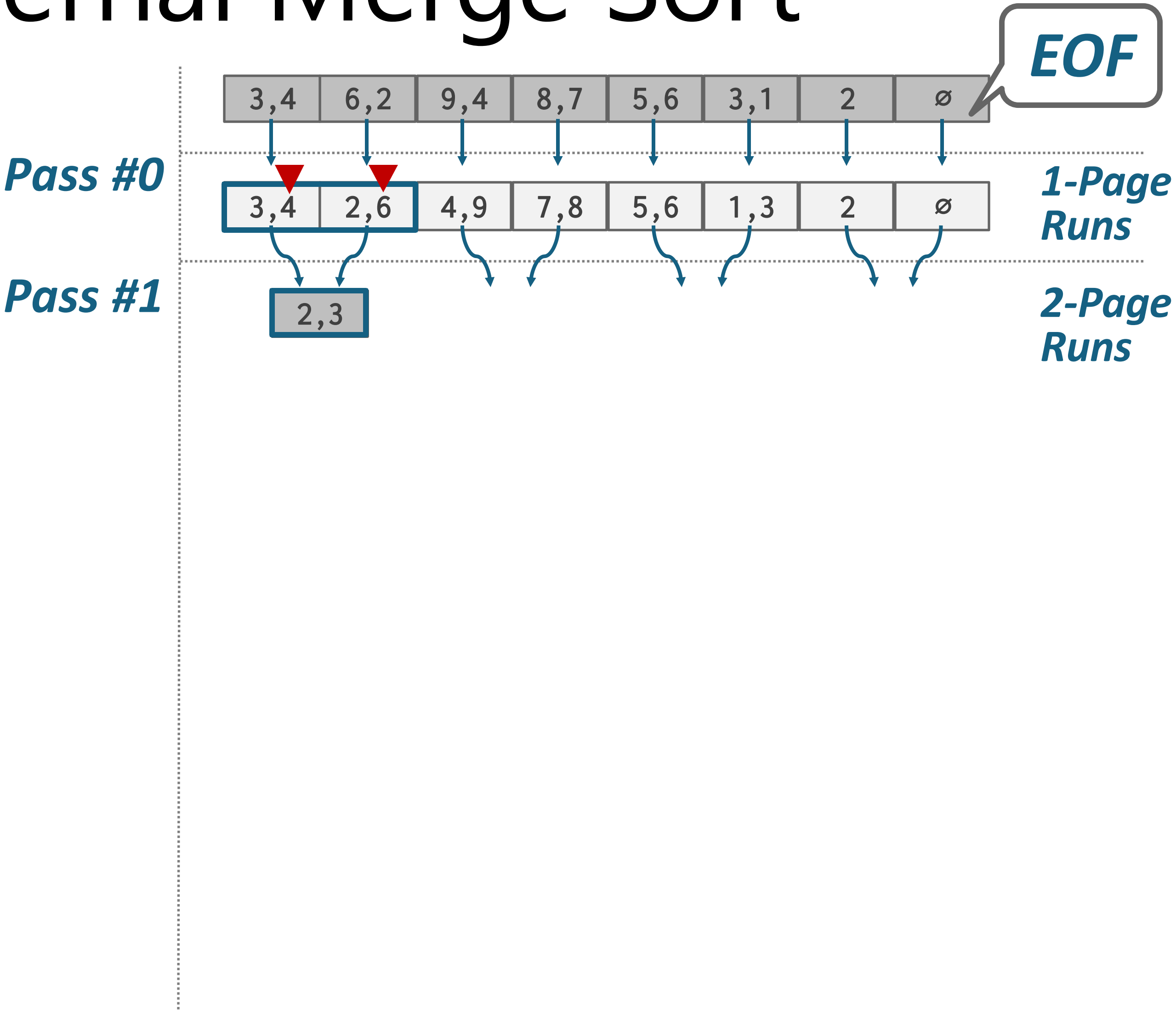
Simplified 2-Way External Merge Sort

- 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 $= 2N \cdot$ (# of passes)



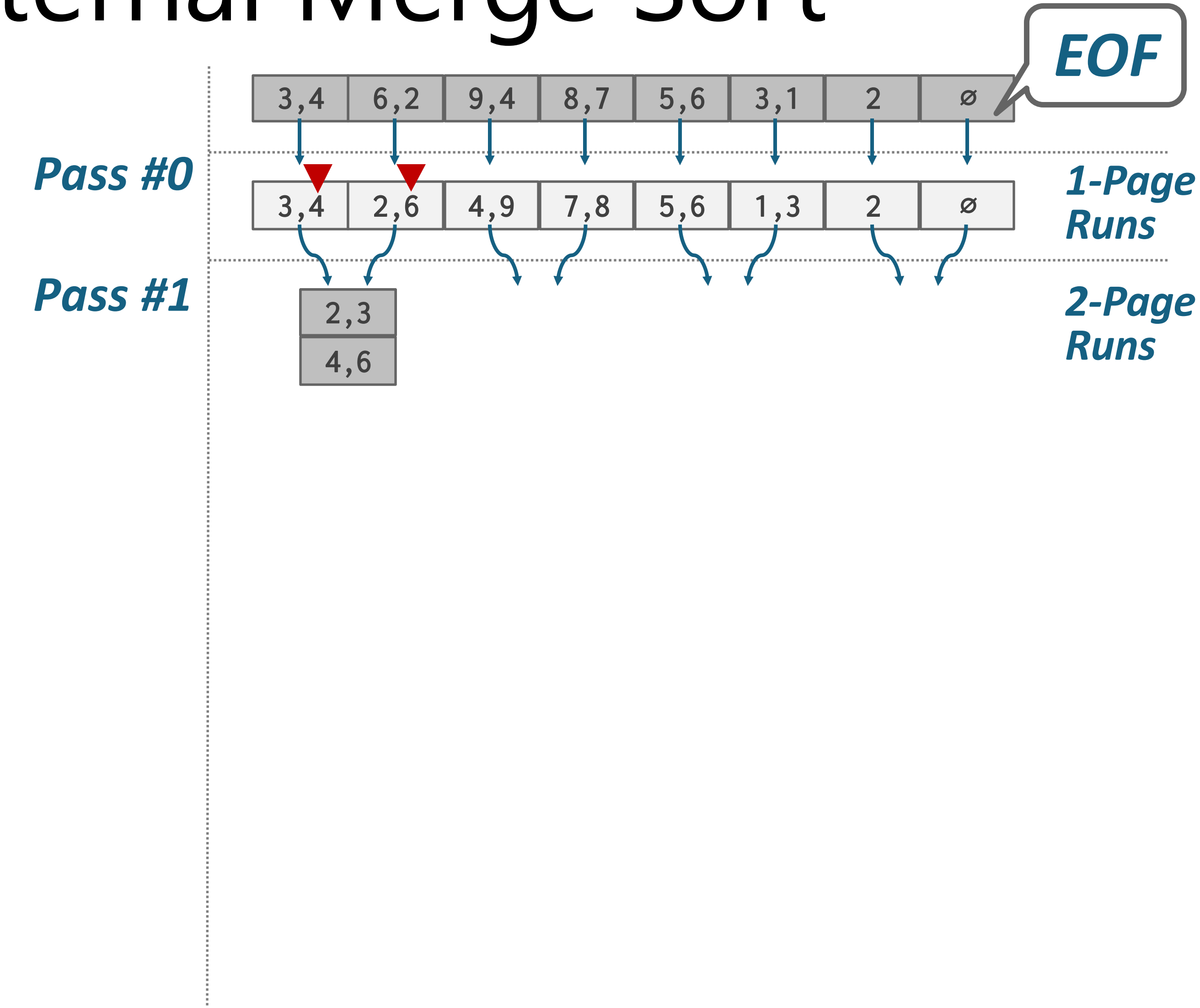
Simplified 2-Way External Merge Sort

- 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 $= 2N \cdot$ (# of passes)



Simplified 2-Way External Merge Sort

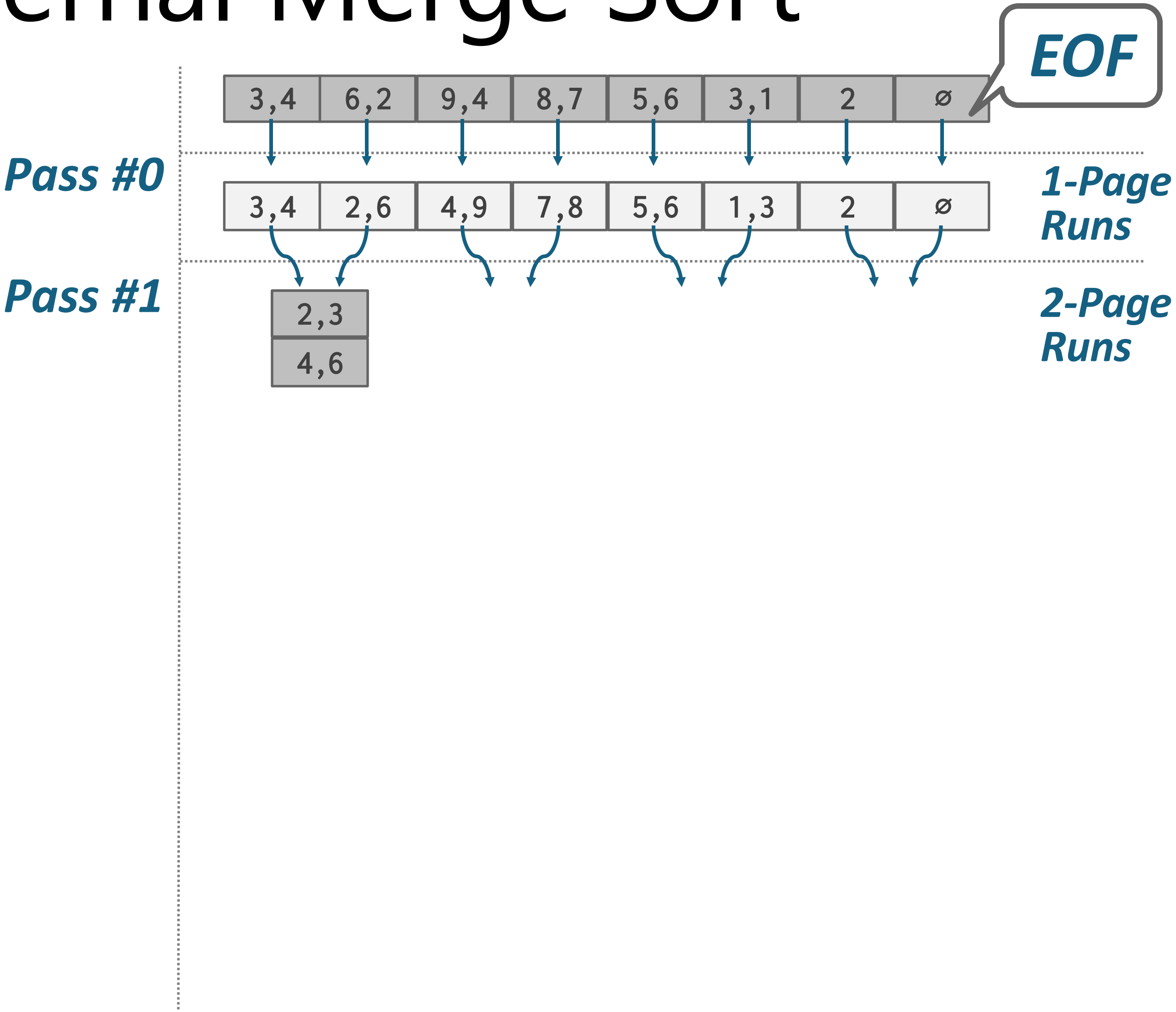
- In each pass, we read and write every page in the file.
- Number of passes
  $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost
  $= 2N \cdot$ (# of passes)



Simplified 2-Way External Merge Sort

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

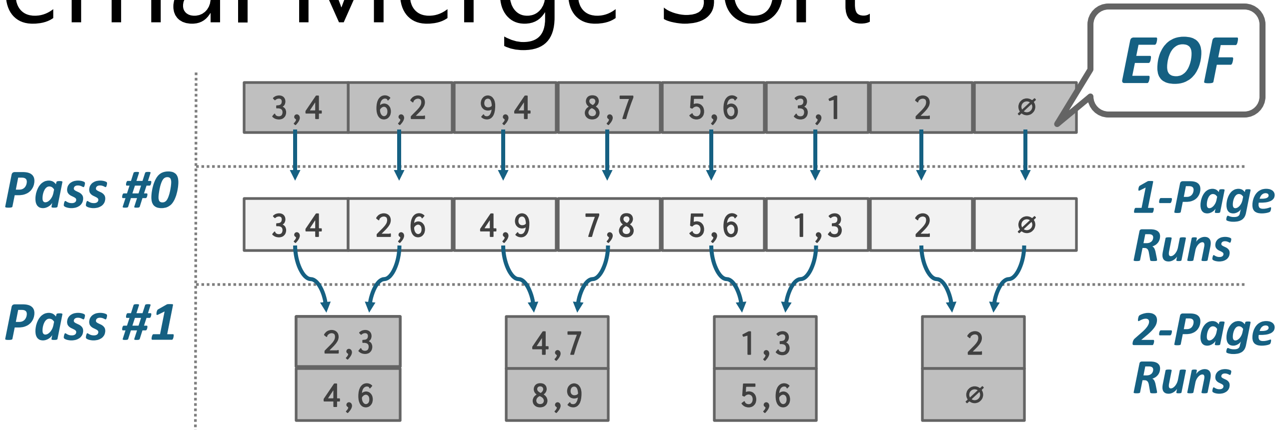
- Number of passes
  $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost
  $= 2N \cdot (\#\ \text{of passes})$



Simplified 2-Way External Merge Sort

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

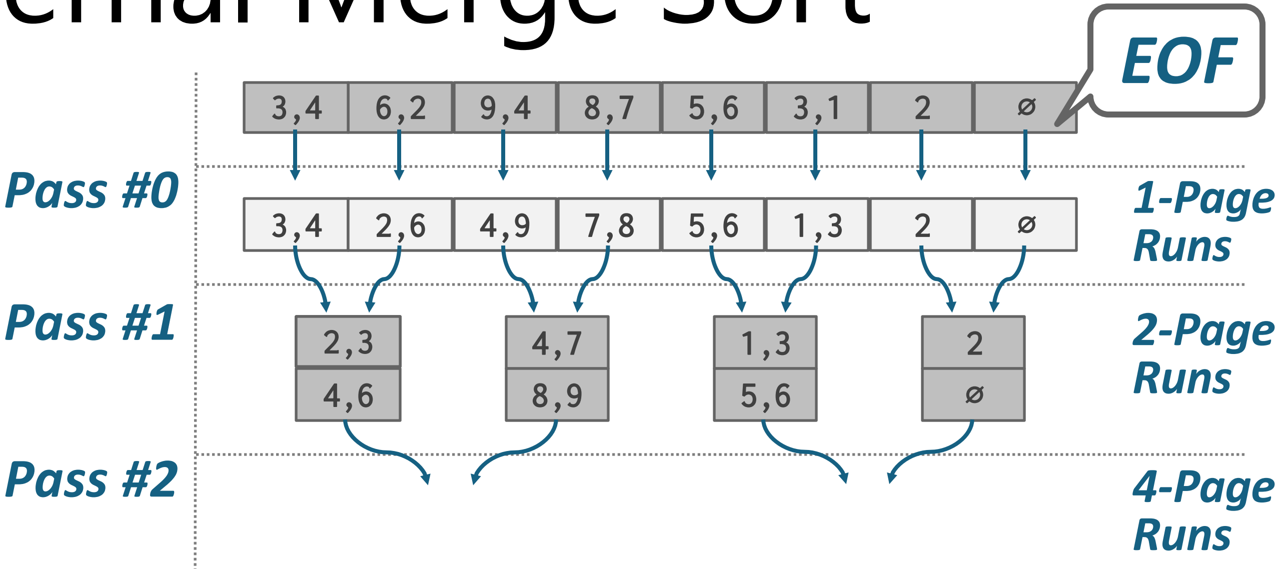
- Number of passes
  $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost
  $= 2N \cdot (\# \text{ of passes})$



Simplified 2-Way External Merge Sort

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

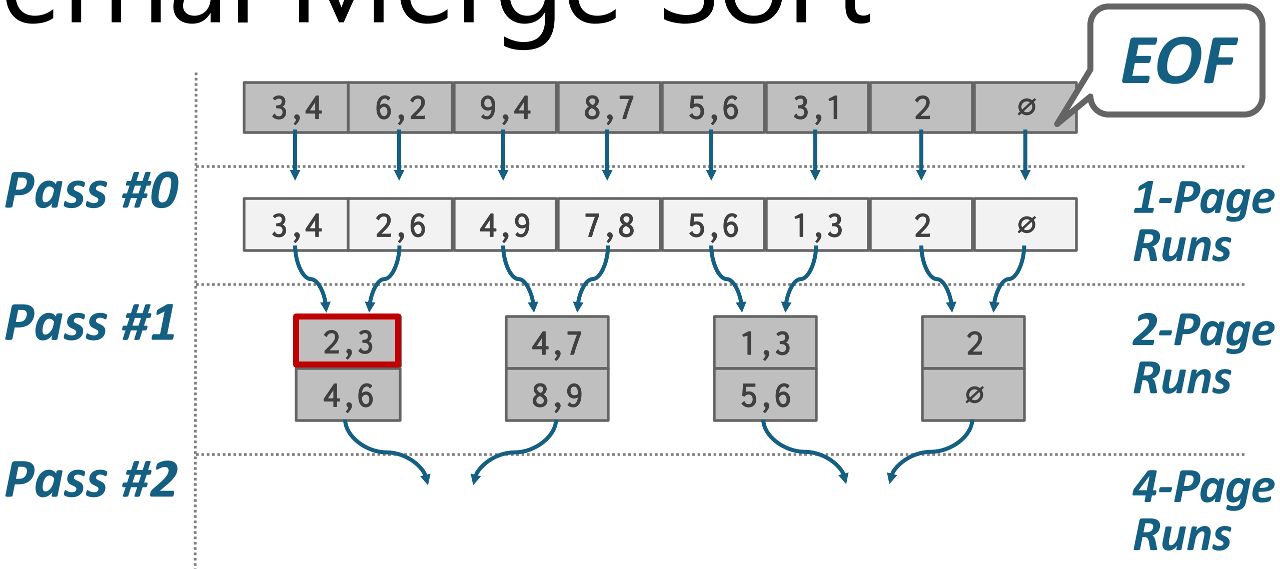
- Number of passes
  $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost
  $= 2N \cdot (\text{\# of passes})$



Simplified 2-Way External Merge Sort

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

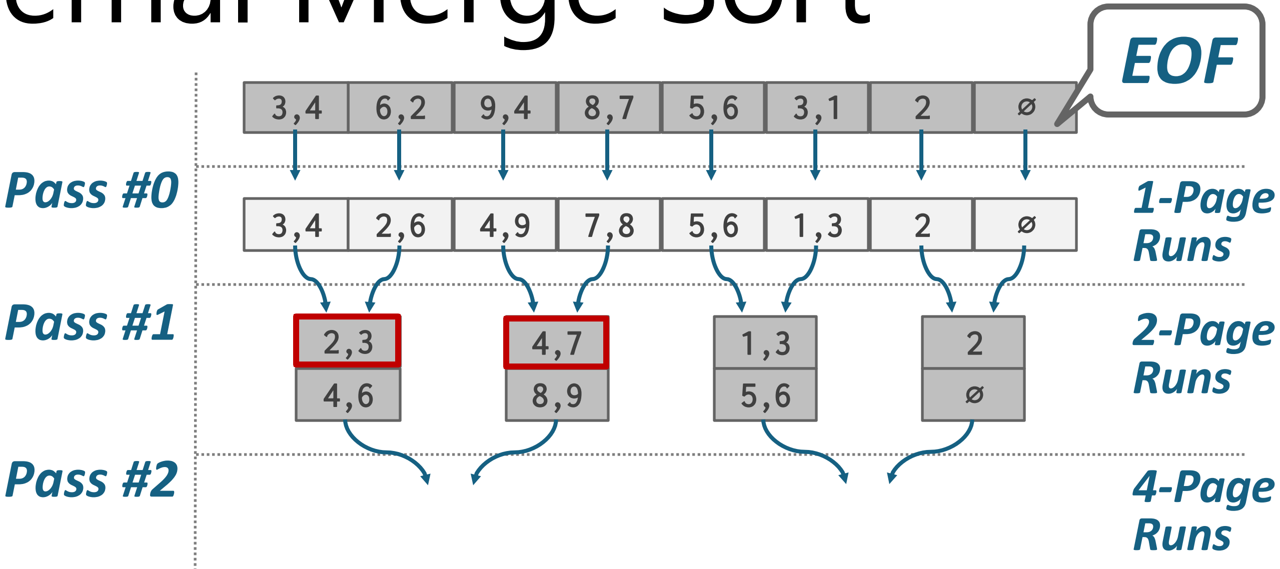
- Number of passes $= 1 + \lceil \log_2 N \rceil$
- Total I/O cost $= 2N \cdot$ (# of passes)



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

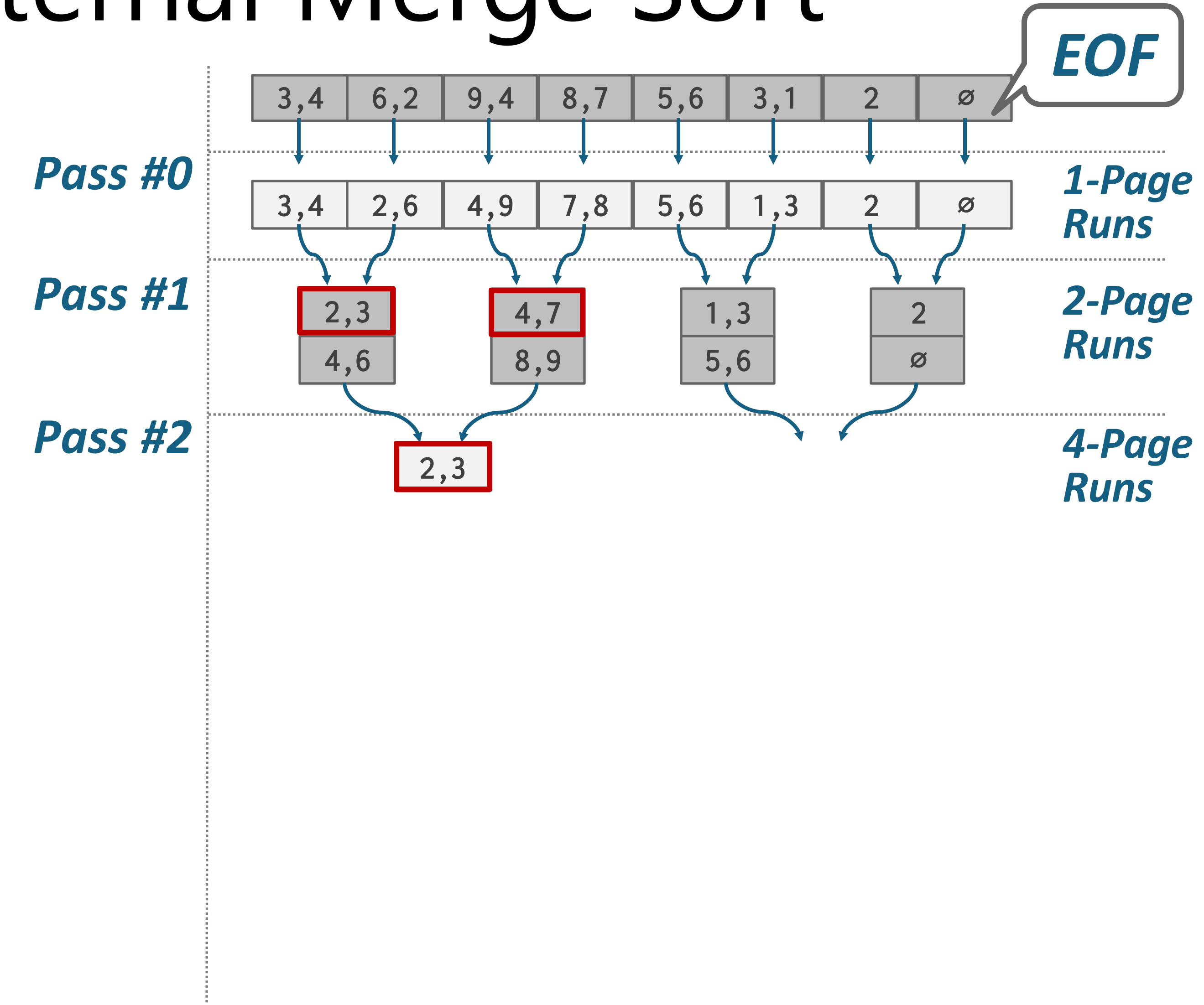


Simplified 2-Way External Merge Sort

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

- Number of passes $= 1 + \lceil \log_2 N \rceil$

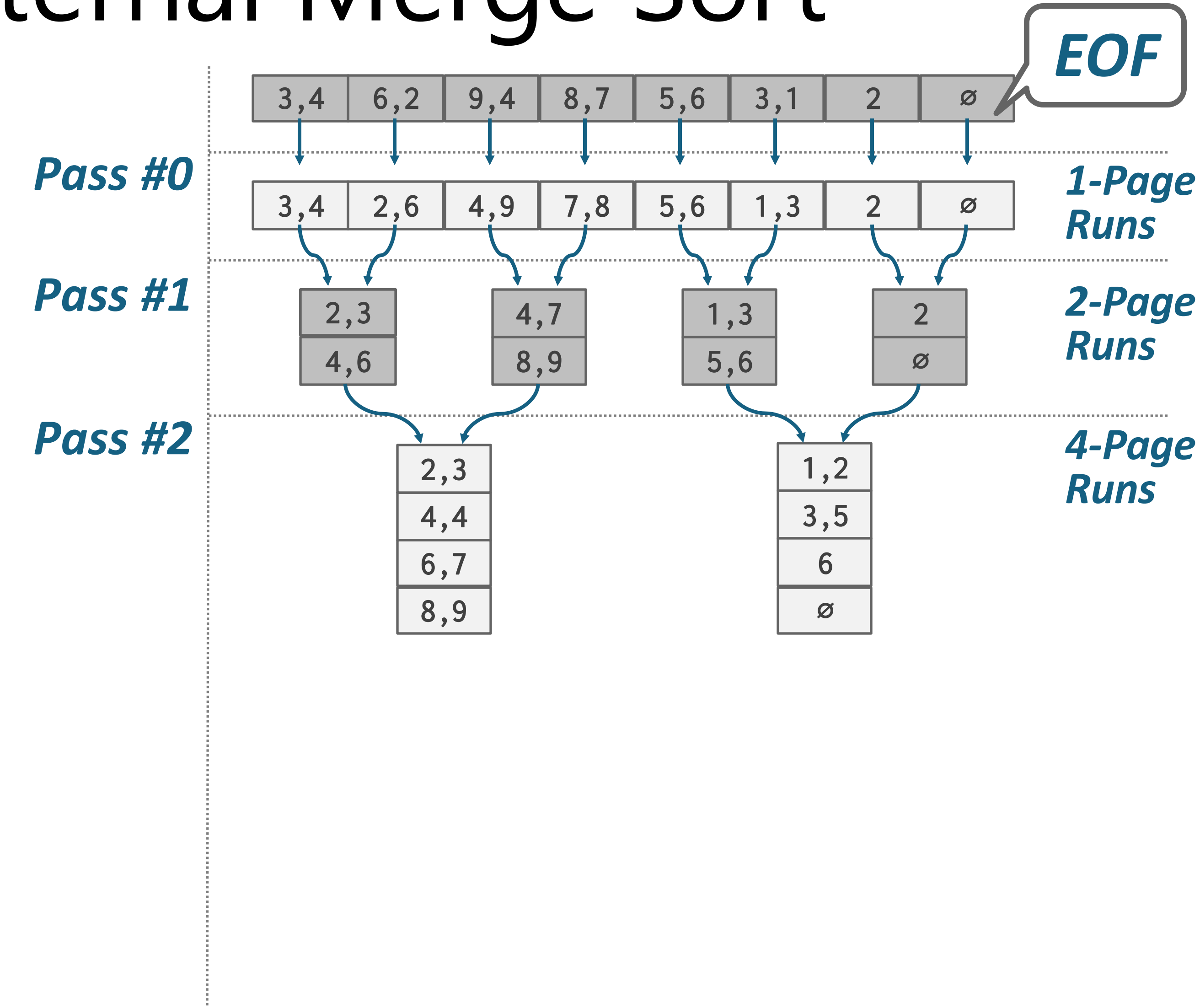
- Total I/O cost $= 2N \cdot$ (# of passes)



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
  = $2N \cdot$ (# of passes)

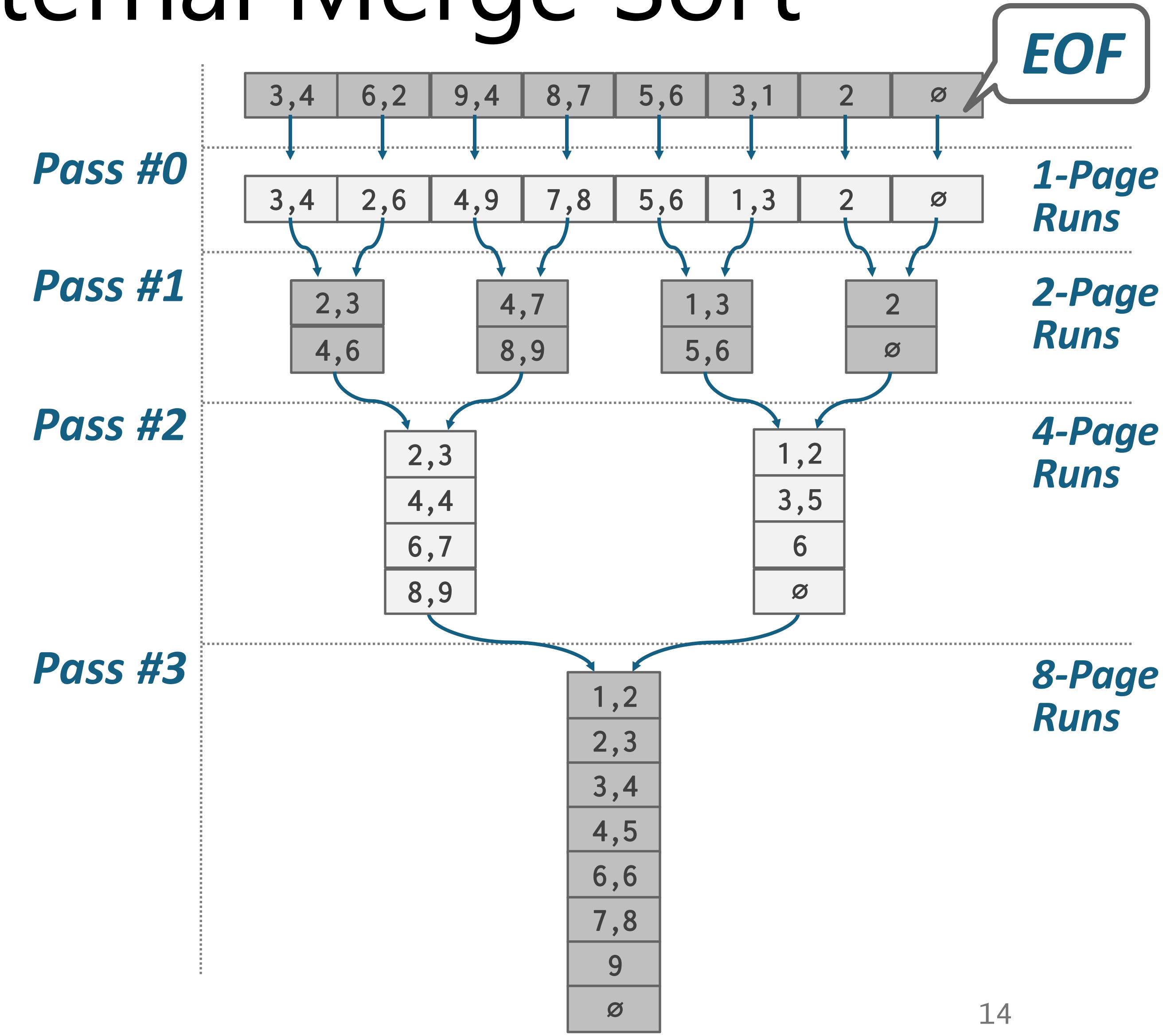


SCHOOL OF DATA SCIENCE

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 $= 2N \cdot$ (# 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., K-way merge)

- Number of passes = $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Total I/O Cost = $2N \cdot$ **(# of passes)**

Example

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

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).

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:** $\lceil N' / B\text{-}1 \rceil$ = $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages).

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

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

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:** $\lceil N' / B\text{-}1 \rceil = \lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages).
  - **Pass #2:** $\lceil N'' / B\text{-}1 \rceil = \lceil 6 / 4 \rceil = 2$ sorted runs, first one has 80 pages and second one has 28 pages.
  - **Pass #3:** Sorted file of 108 pages.
- $1+\lceil \log_{B-1} \lceil N / B \rceil \rceil = 1+\lceil \log_4 22 \rceil = 1+\lceil 2.229... \rceil$
  $= \mathbf{4\ passes}$

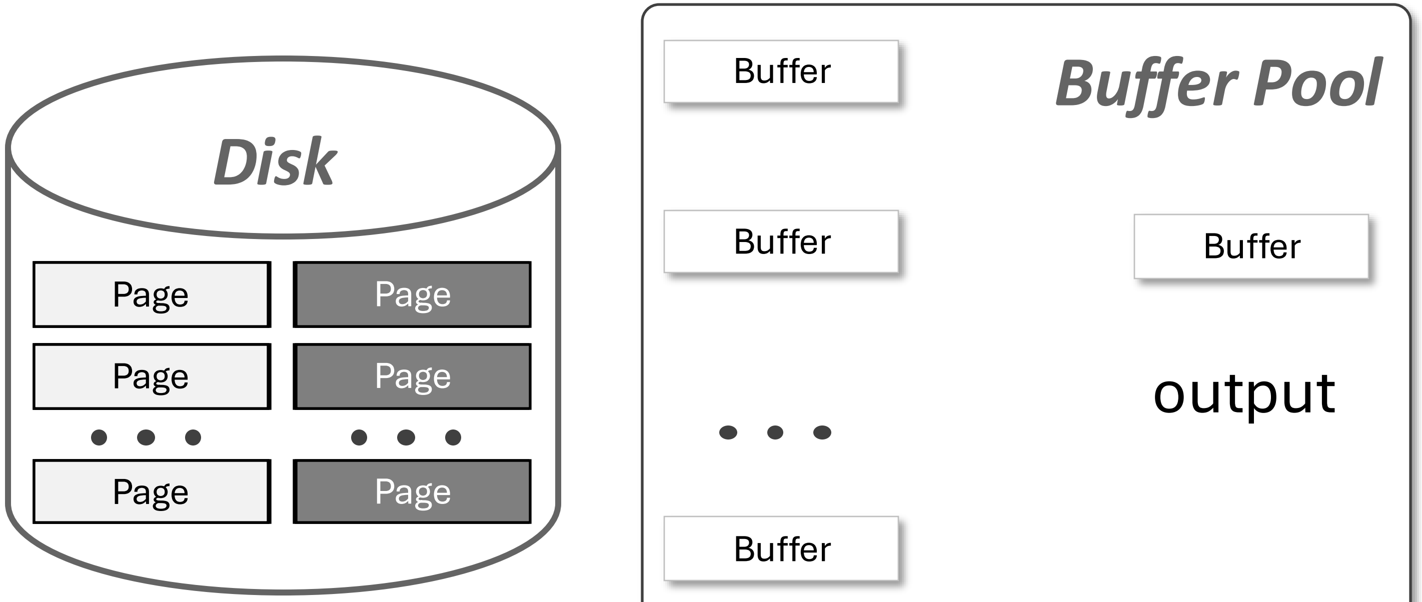
SCHOOL OF DATA SCIENCE

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.

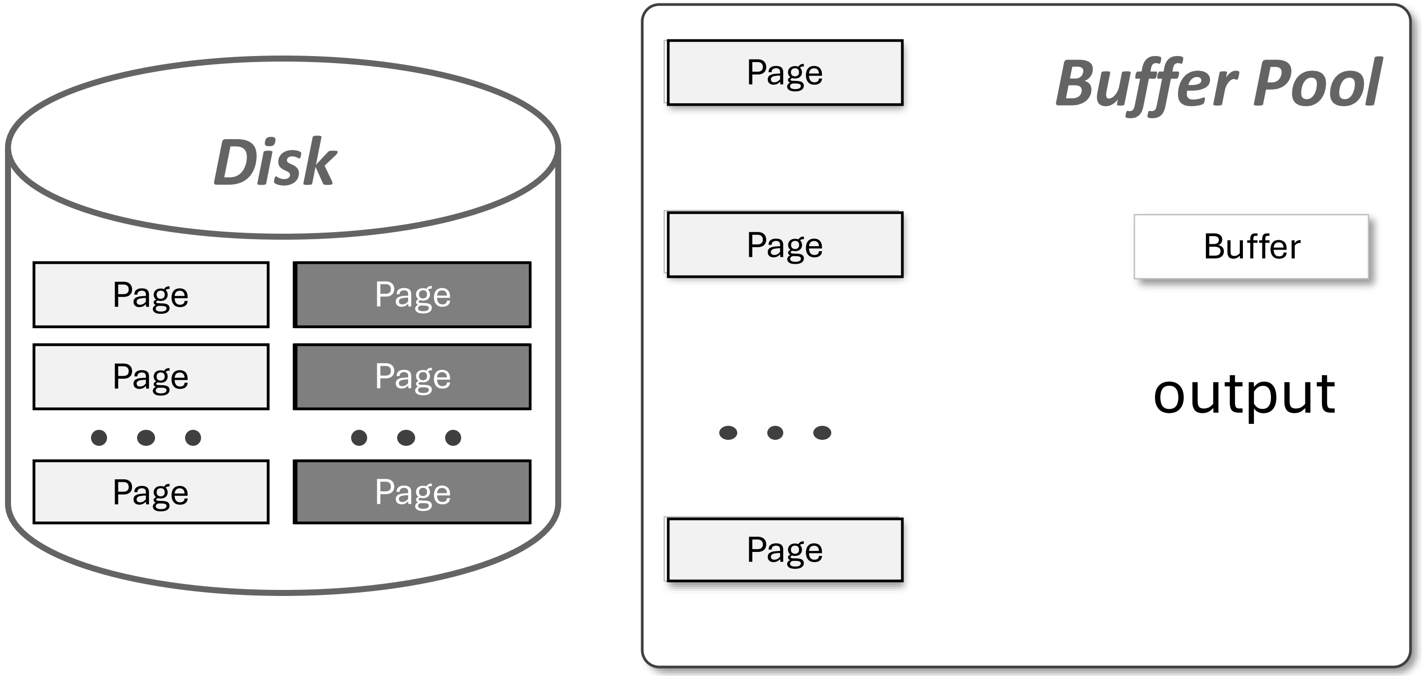
Double Buffering

- 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



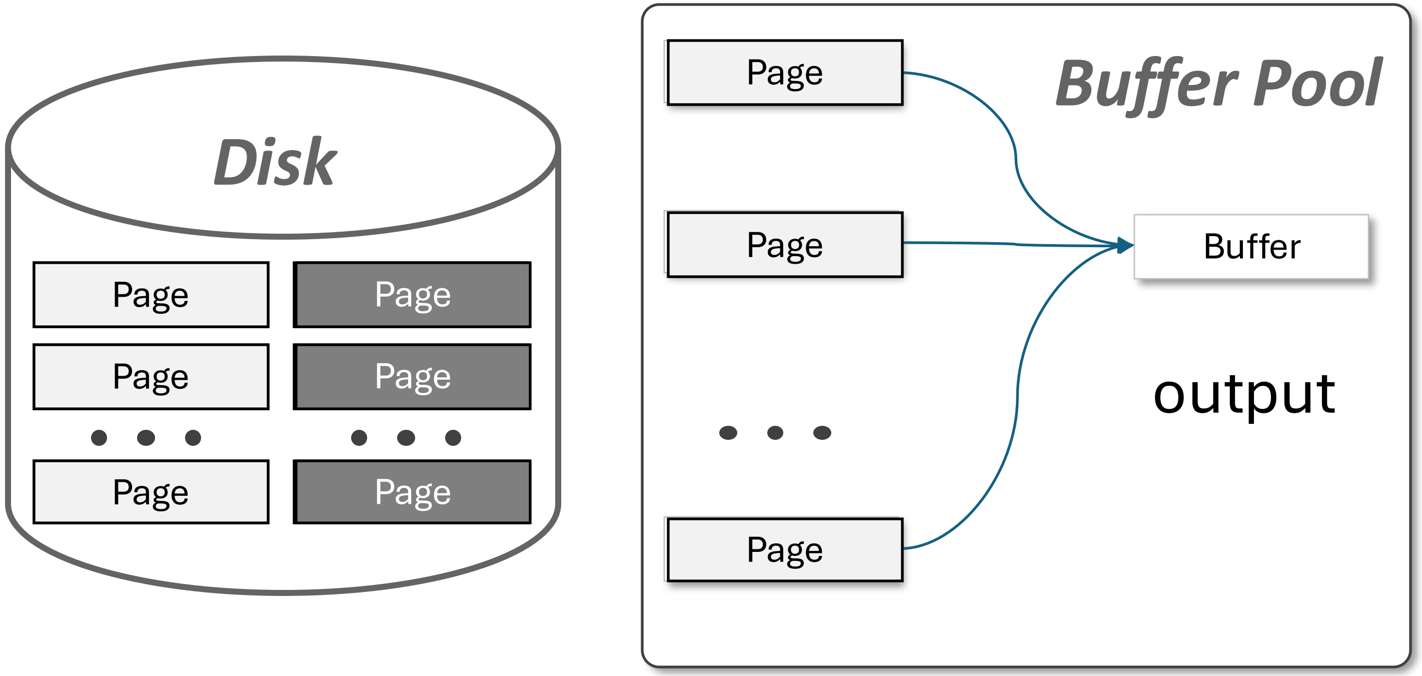
Double Buffering

- 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



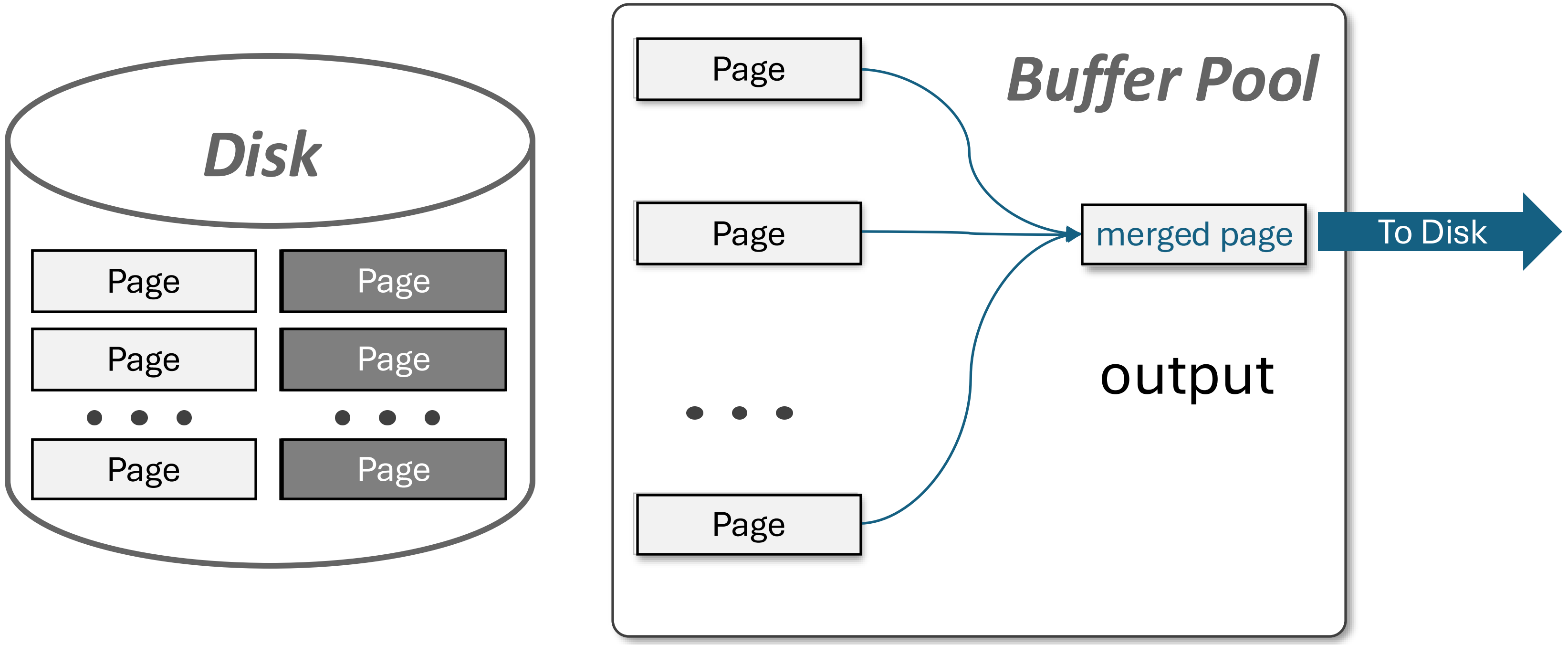
Double Buffering

- 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



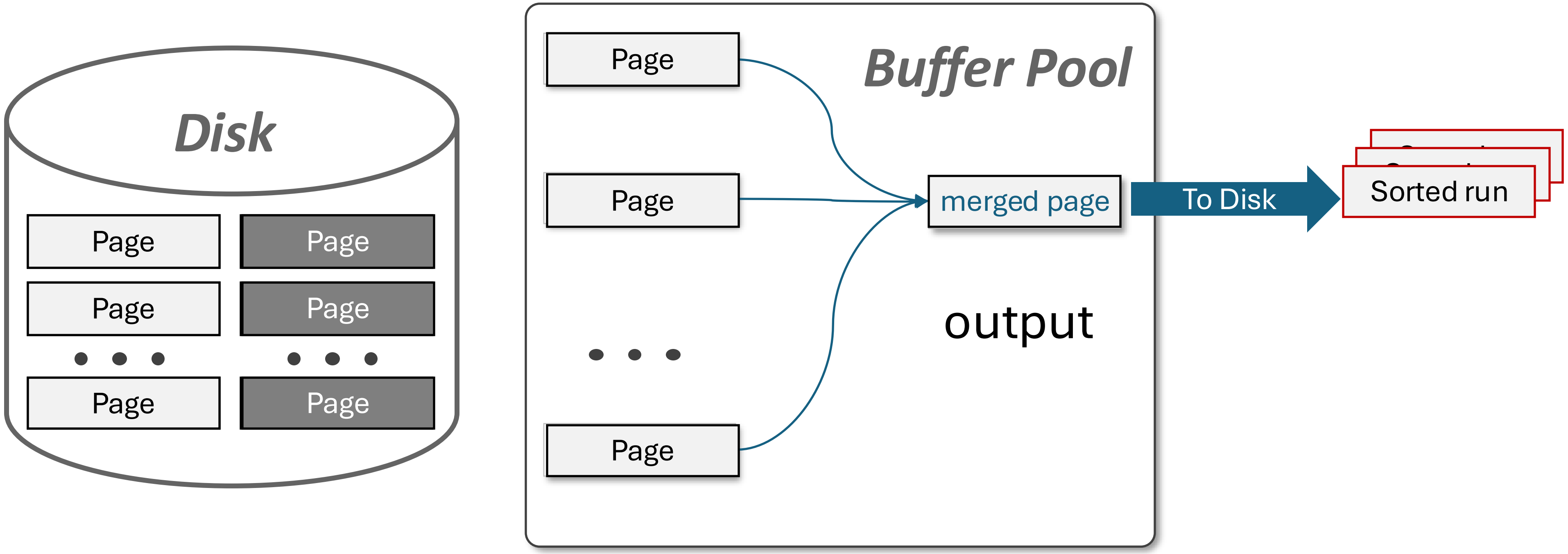
Double Buffering

- 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



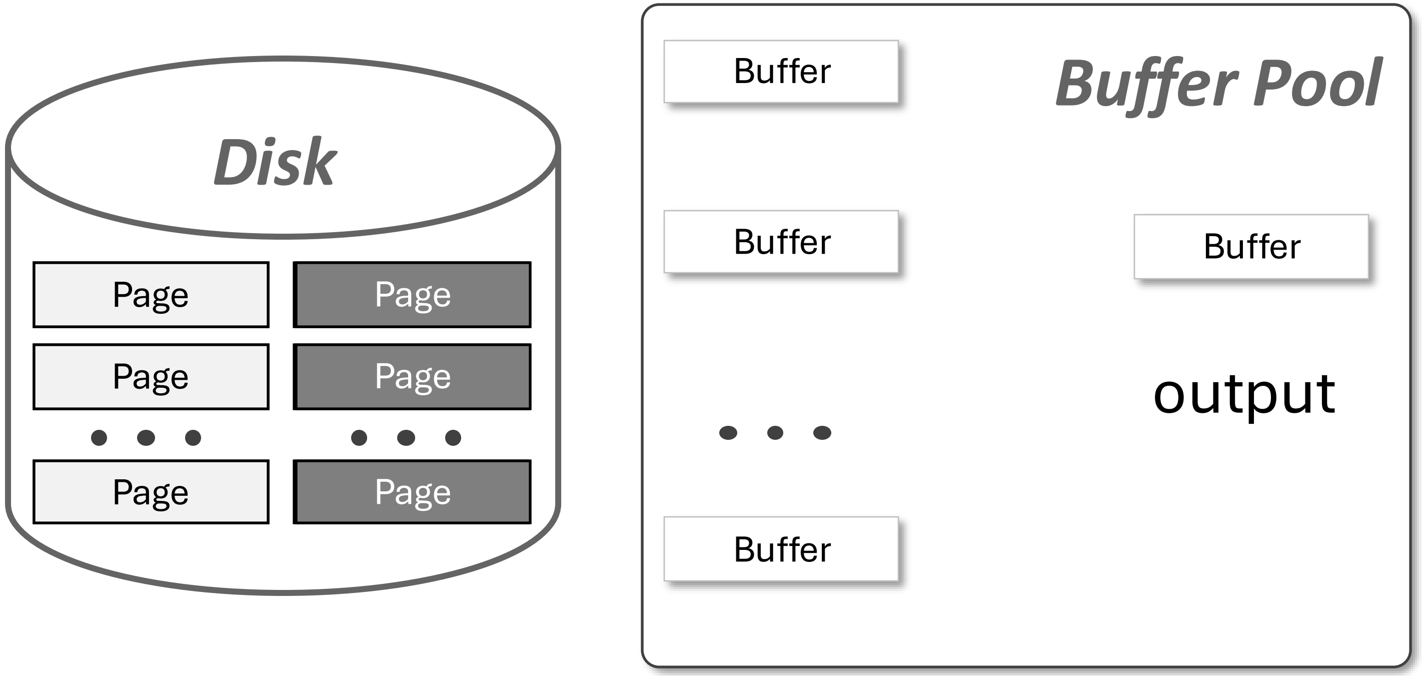
Double Buffering

- 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



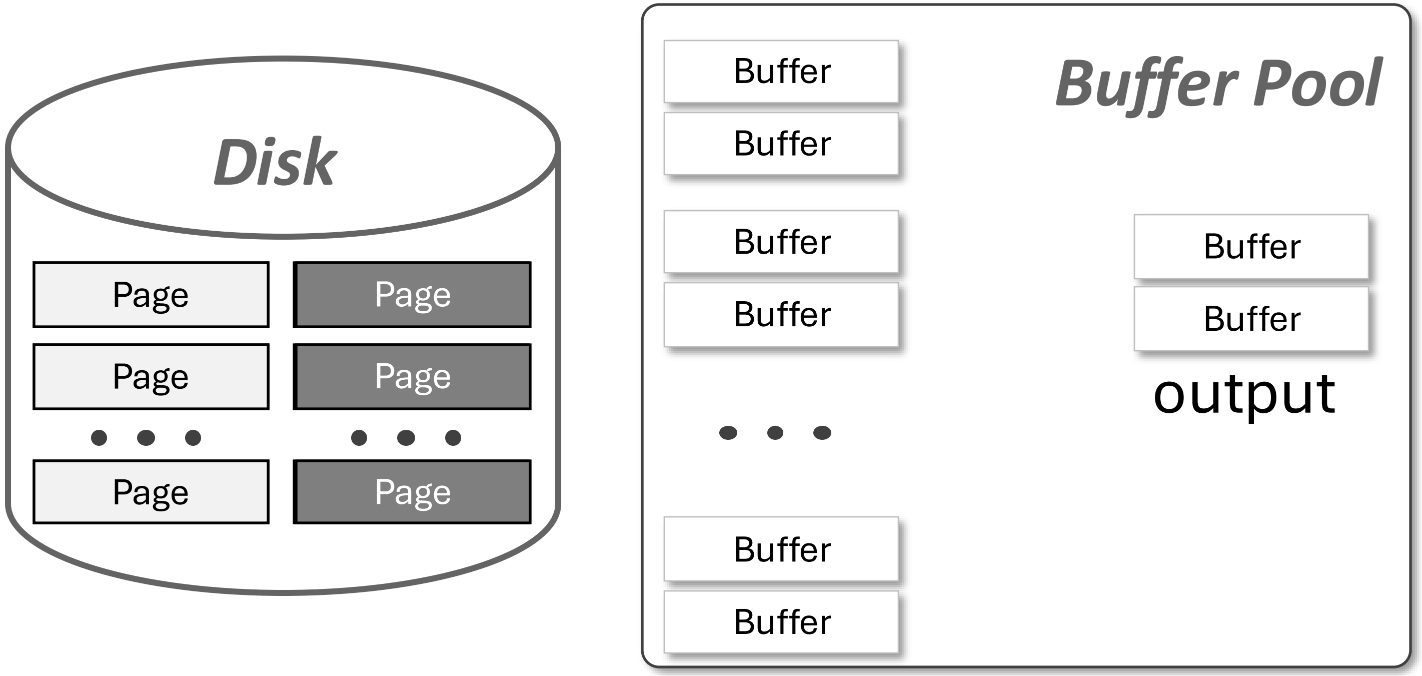
Double Buffering

- 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



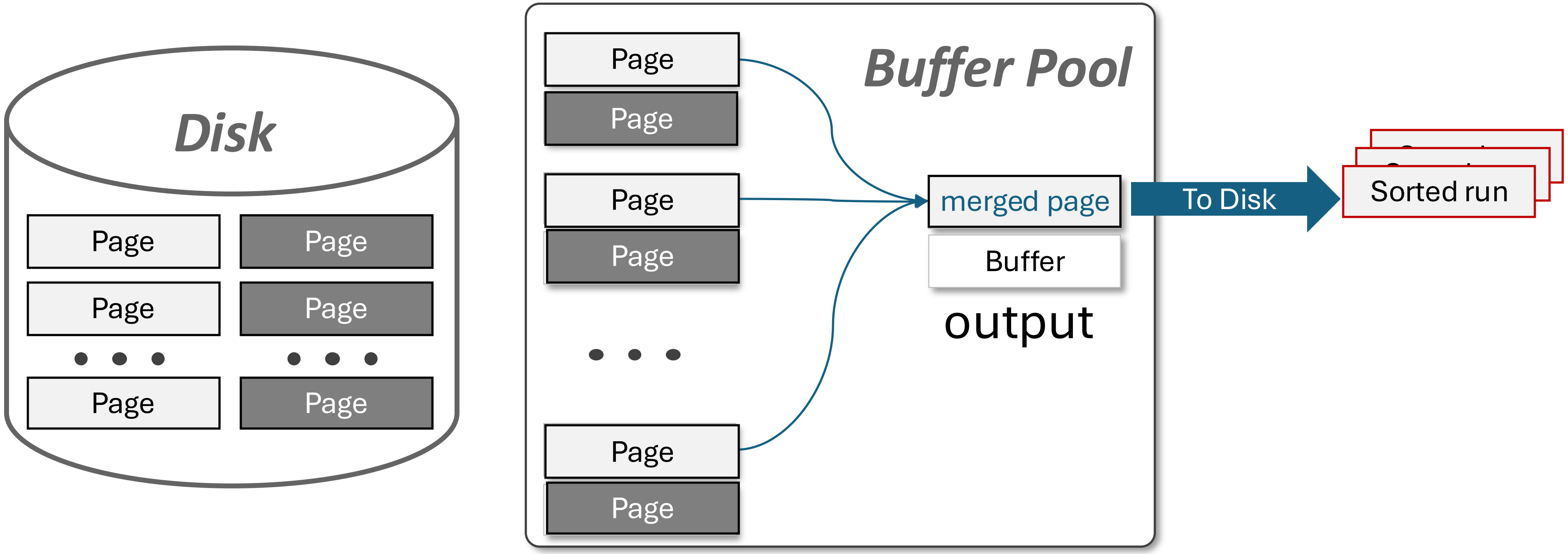
Double Buffering

- 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



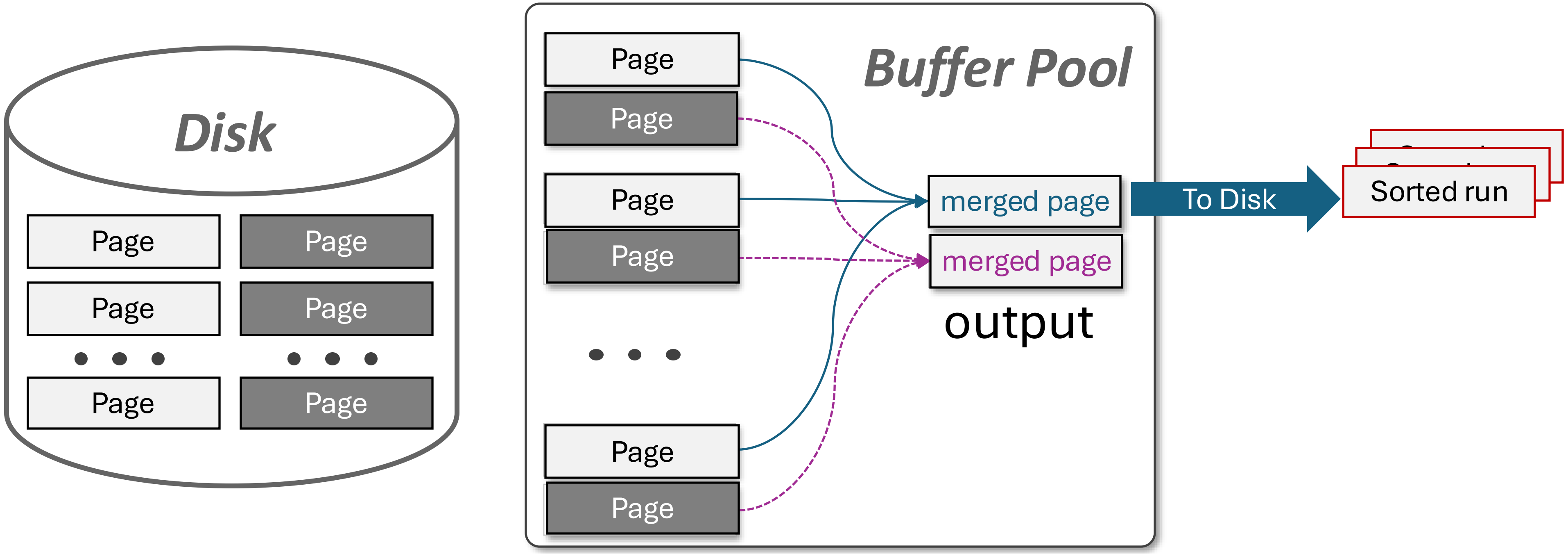
Double Buffering

- 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



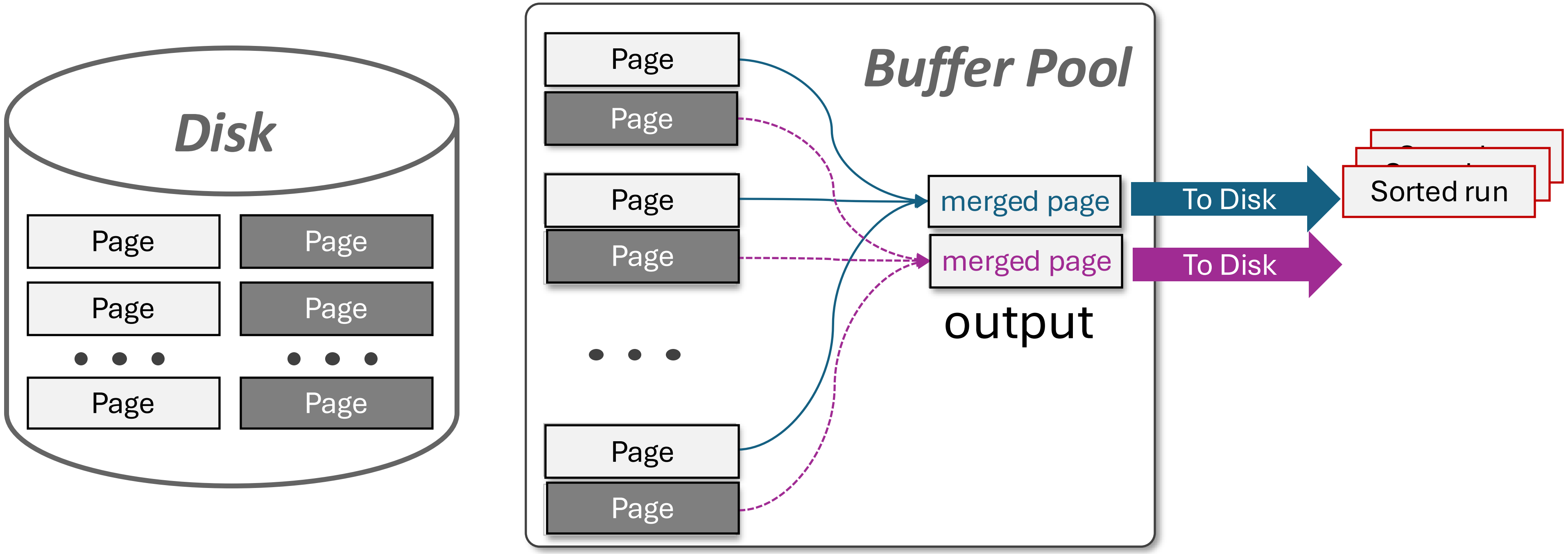
Double Buffering

- 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



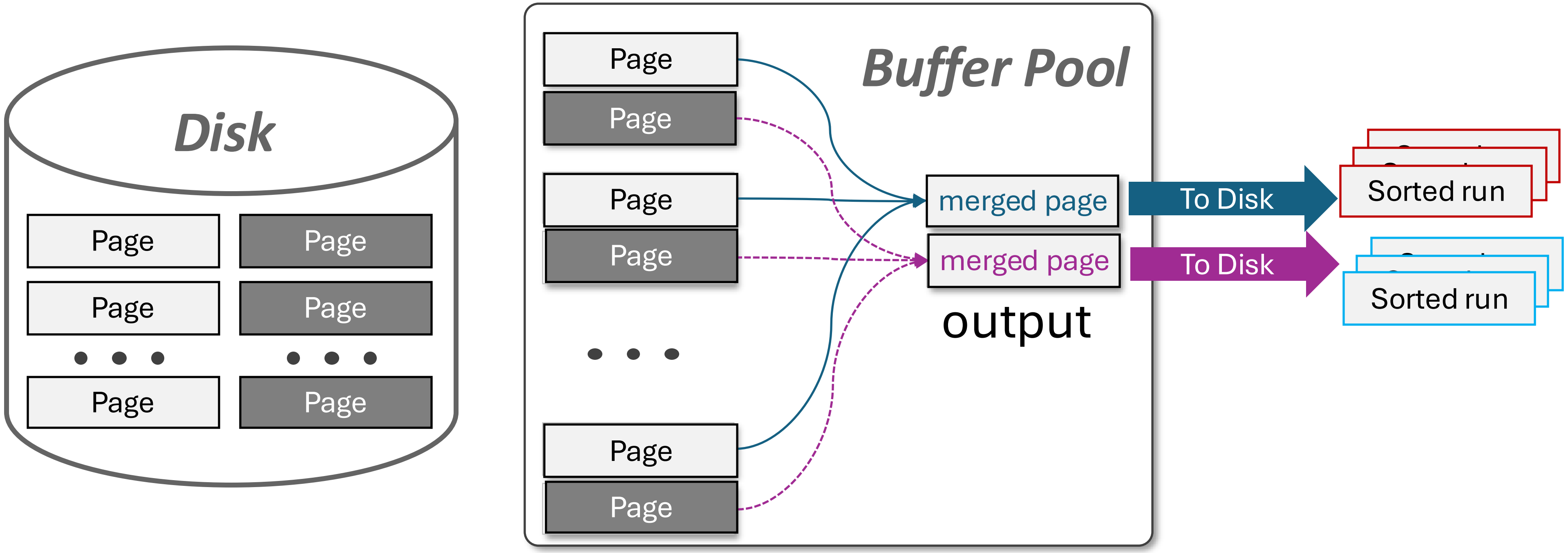
Double Buffering

- 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

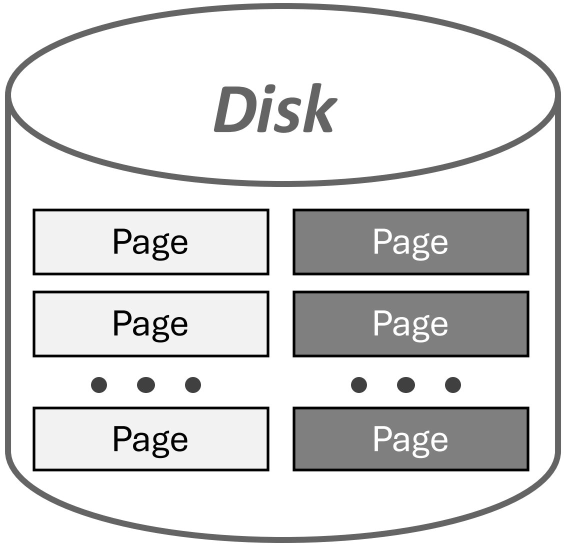


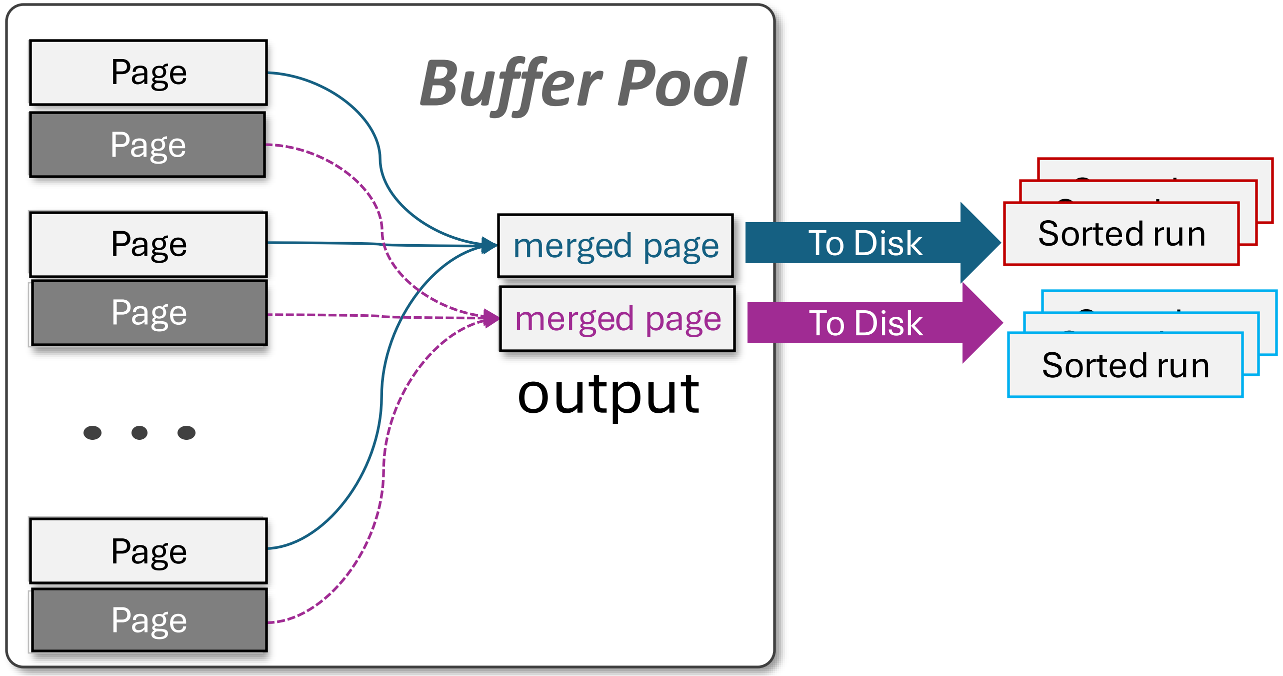
Double Buffering

- 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.





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

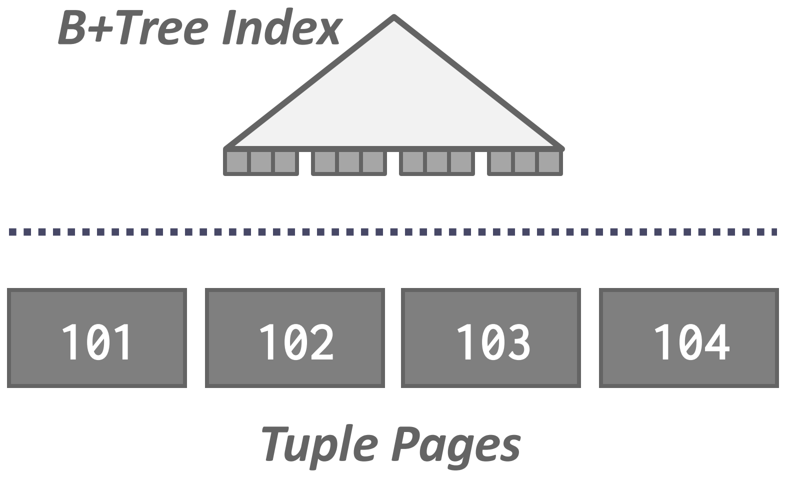
- If the table that must be sorted already has a B+Tree index on the sort attribute(s), then we can use that to accelerate sorting.

- 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

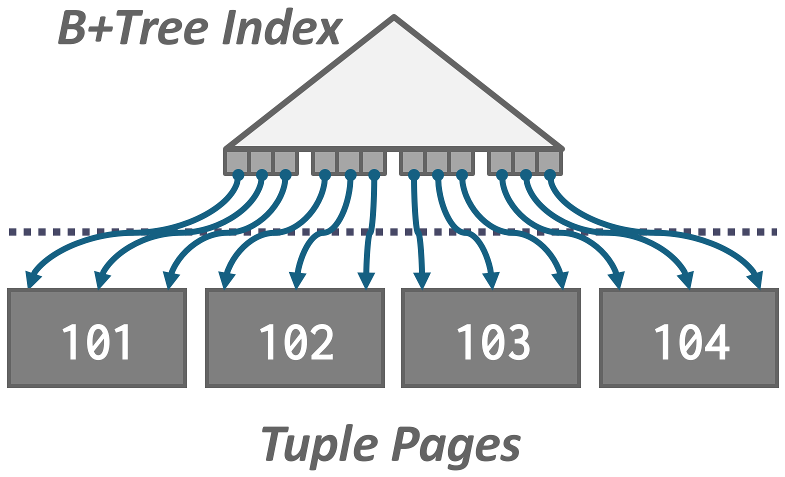
- Traverse to the left-most leaf page, and then retrieve tuples from all leaf pages.

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



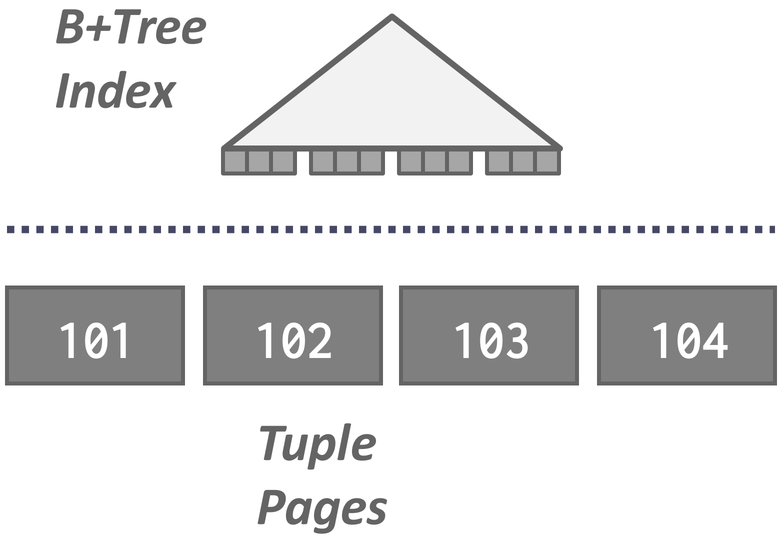
Case #1: Clustered B+Tree

- 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.



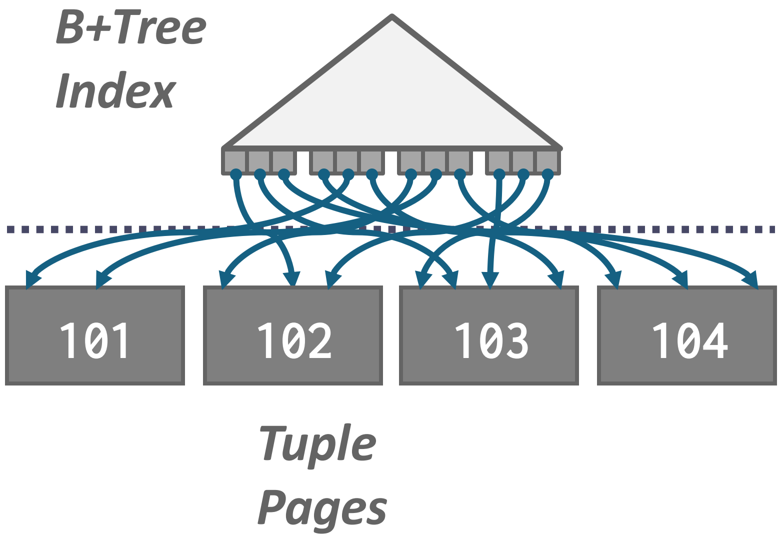
Case #2: Unclustered B+Tree

- 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.



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

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

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

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

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

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

SCHOOL OF DATA SCIENCE

Sorting Aggregation

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

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

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

SCHOOL OF DATA SCIENCE

Sorting Aggregation

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

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

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

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

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

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

SCHOOL OF DATA SCIENCE

Alternatives To Sorting

- What if we do <u>not</u> 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

⋮

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

⋮

Filter

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

⋮

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

⋮

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

⋮

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

⋮

Filter

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

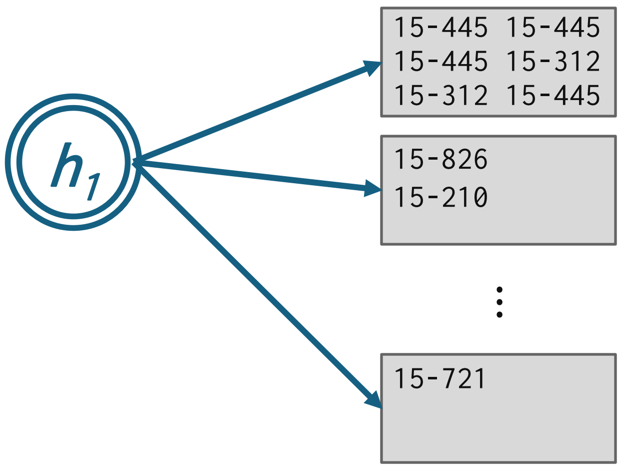
⋮

Remove Columns

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

⋮

B-1 partitions



SCHOOL OF DATA SCIENCE

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

⋮

Filter

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

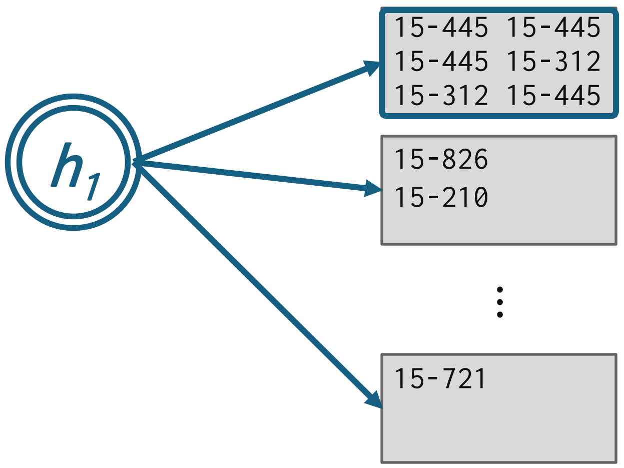
⋮

Remove Columns

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

⋮

B-1 partitions



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

⋮

Filter

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

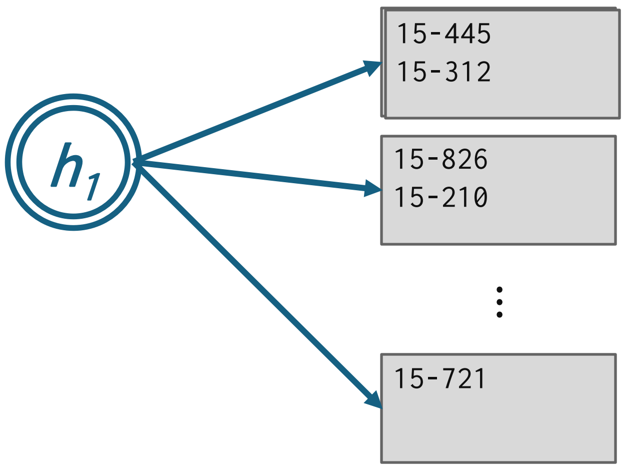
⋮

Remove Columns

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

⋮

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')
```

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

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

15-826
15-826

⋮

SCHOOL OF
DATA SCIENCE

Phase #2: ReHash

```sql
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

B-1 Partitions

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

15-826
15-826

⋮

Phase #2: ReHash

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

Phase #1 Buckets

```
15-445 15-445
15-445 15-445
15-445 15-445
```

```
15-826
15-826
```

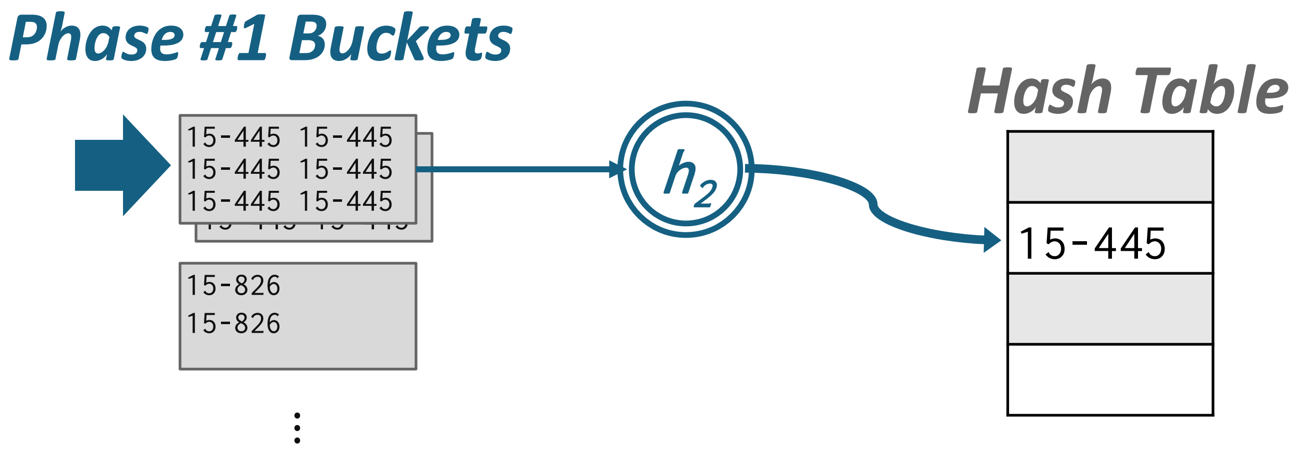
⋮

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')
```



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

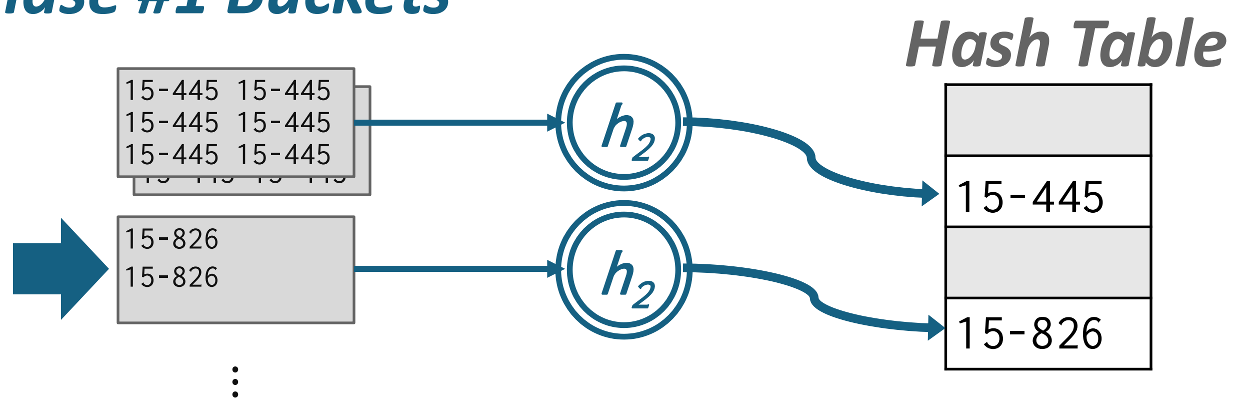
Phase #2: ReHash

```
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

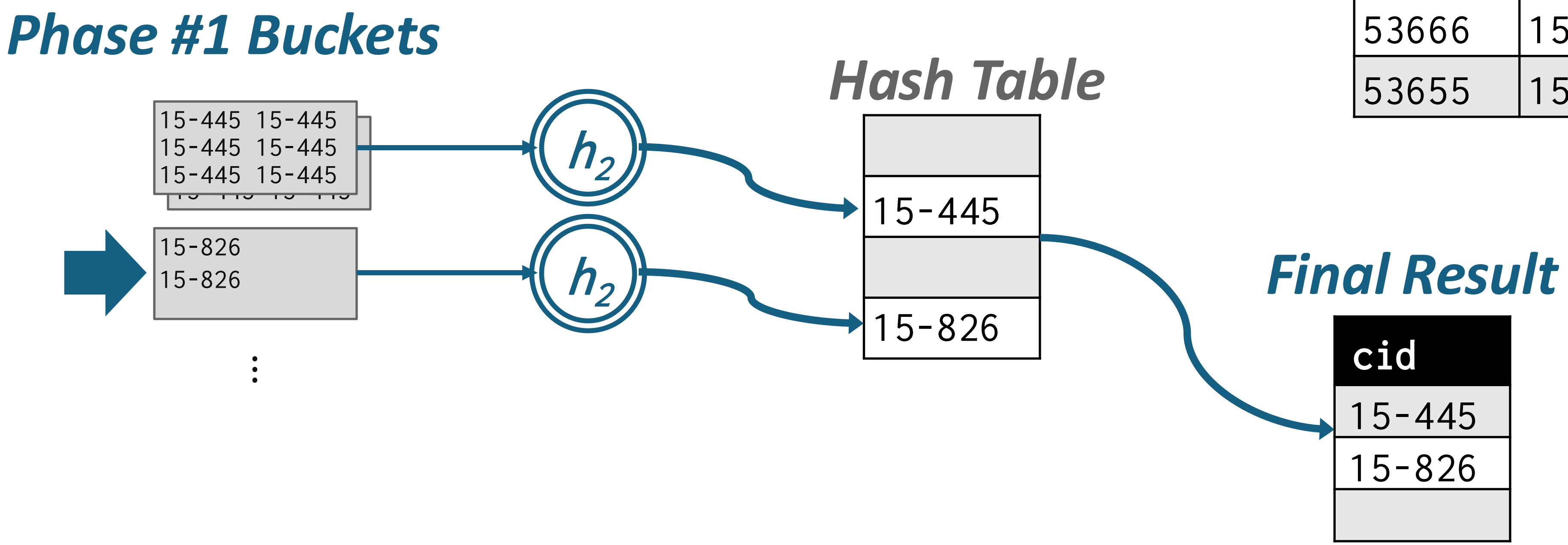


Phase #2: ReHash

```sql
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

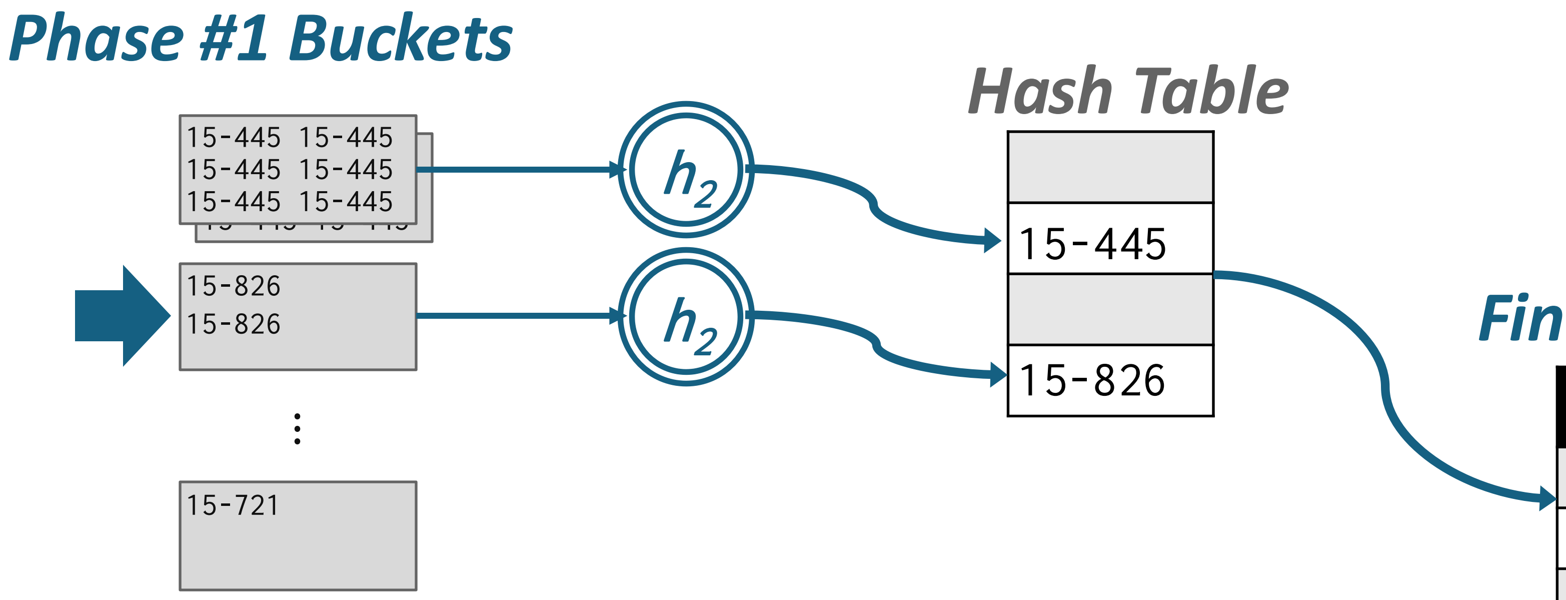


Phase #2: ReHash

```
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



cid
15-445
15-826

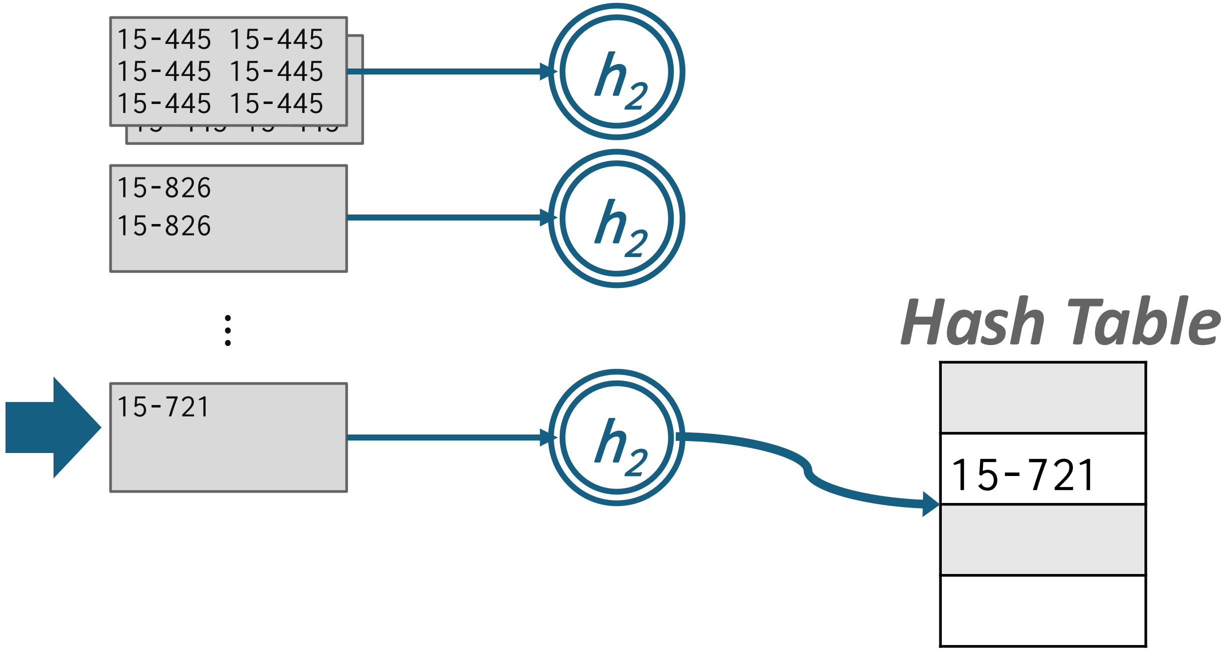
Phase #2: ReHash

```
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



Final Result

cid
15-445
15-826

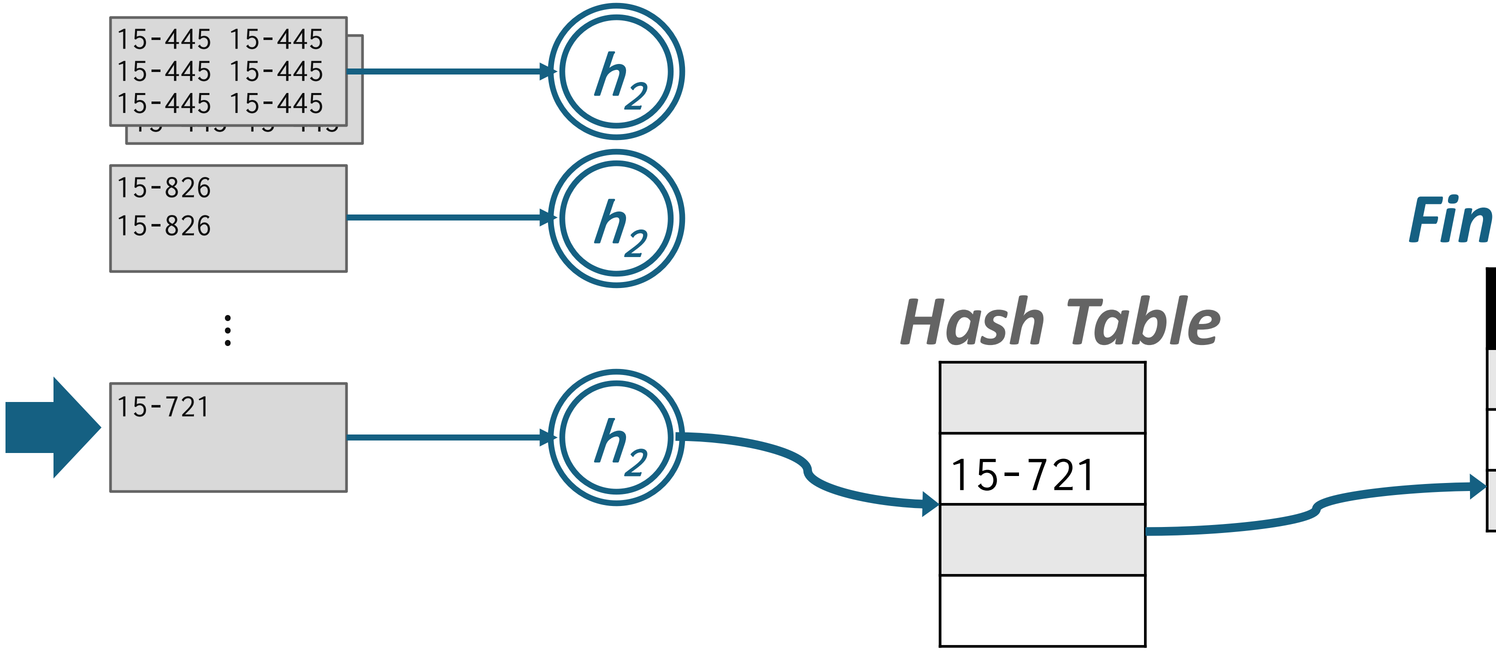
Phase #2: ReHash

```sql
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



Final Result

cid
15-445
15-826
15-721

SCHOOL OF DATA SCIENCE

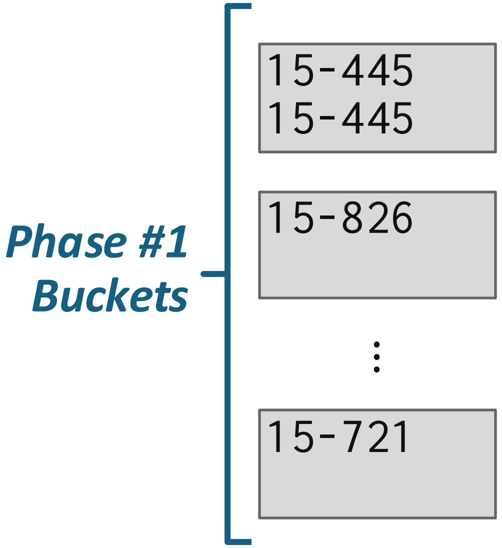
Hashing Summarization

- 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`

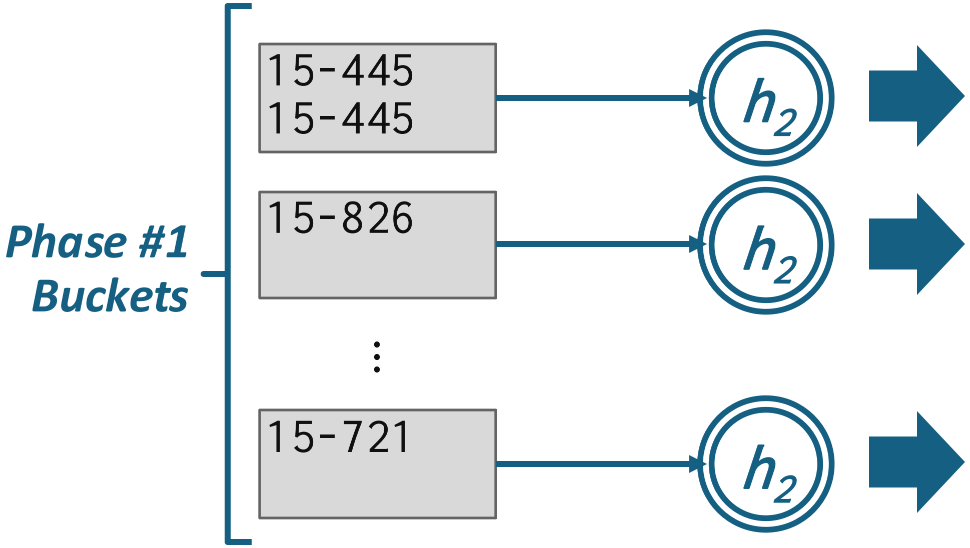
Hashing Summarization

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



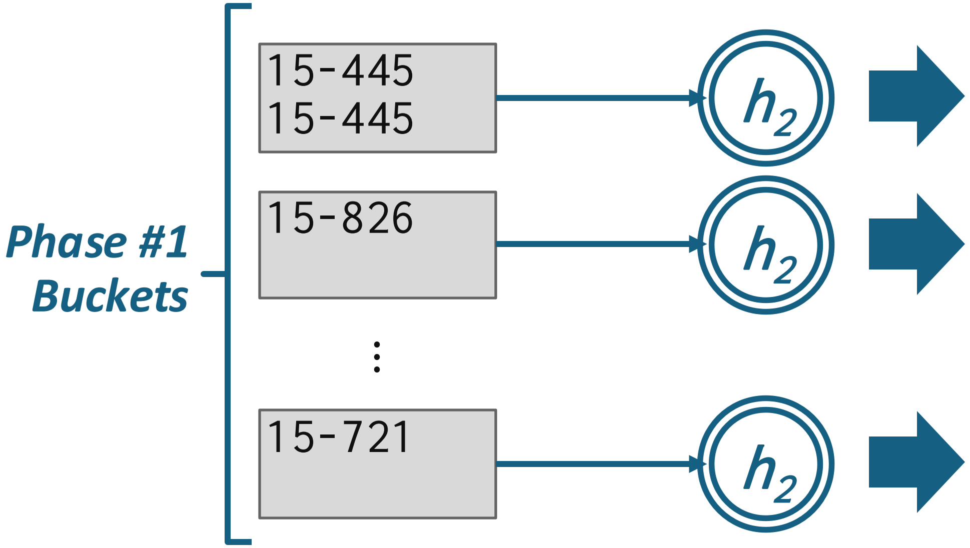
Hashing Summarization

```
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
```

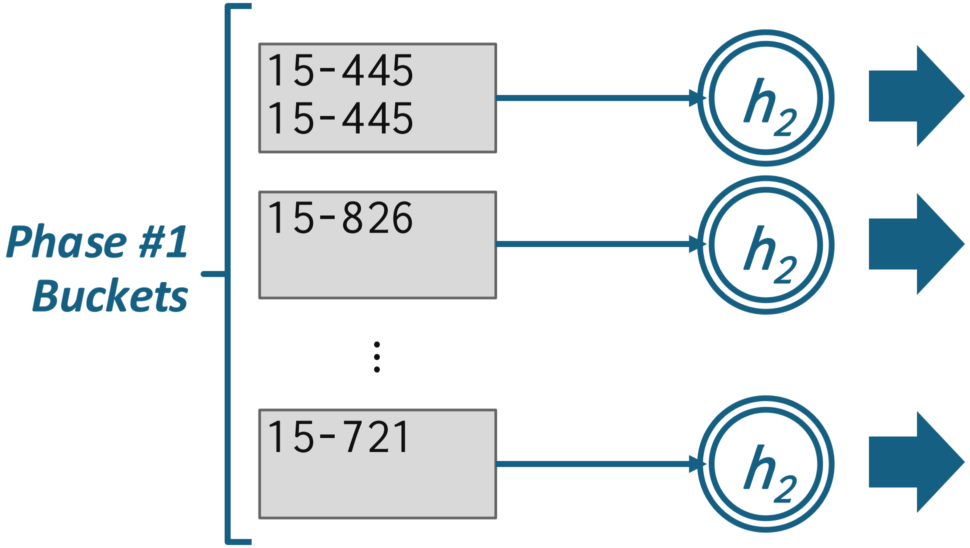


Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Hashing Summarization

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



Hash Table

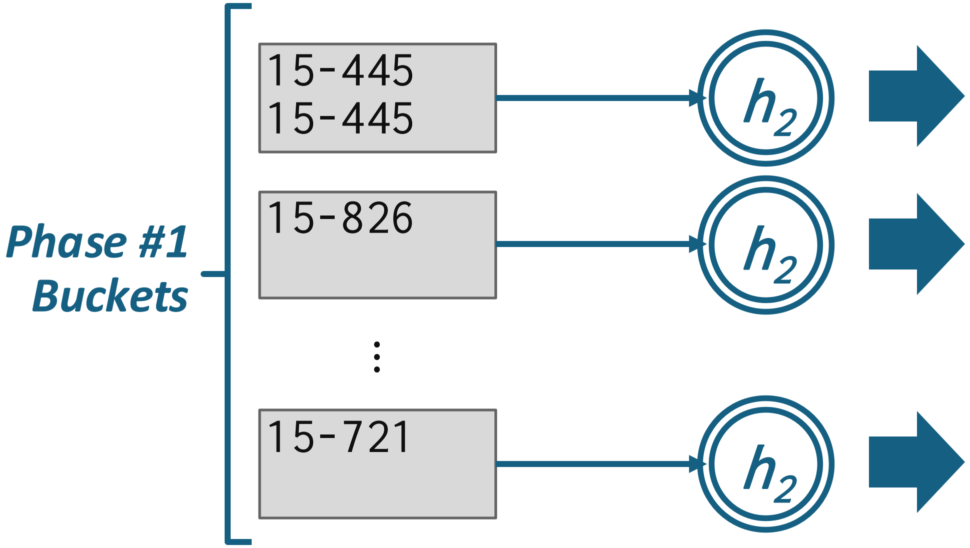
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Hashing Summarization

```sql
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)



Hash Table

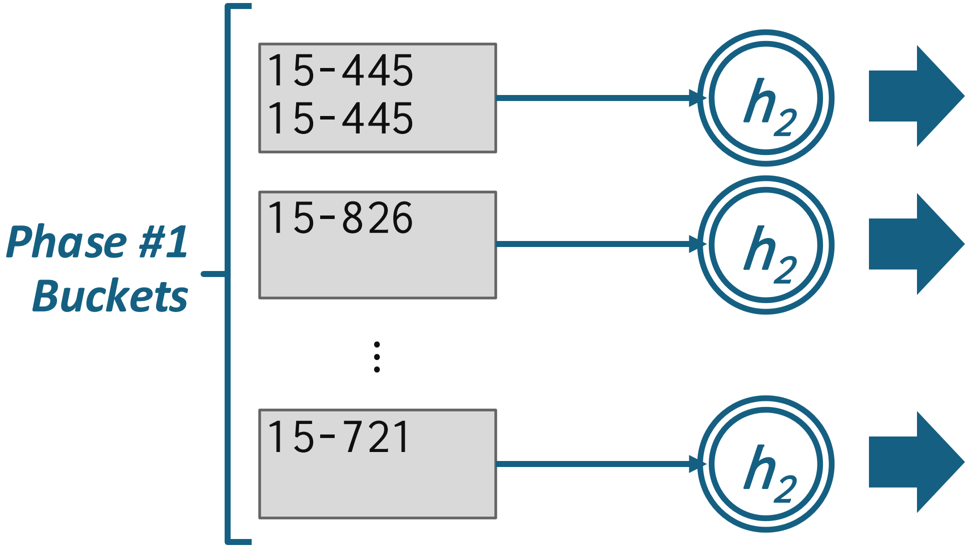
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Hashing Summarization

```sql
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)



Hash Table

key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

Final Result

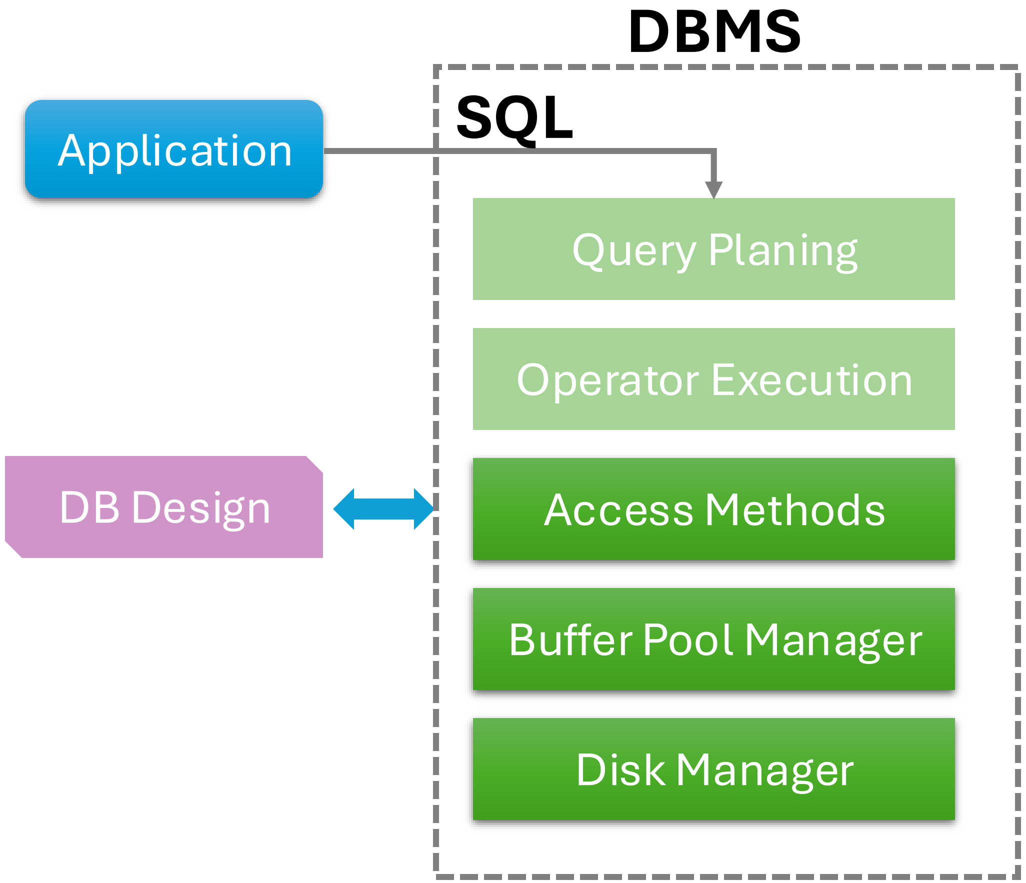
cid	AVG(gpa)
15-445	3.66
15-826	3.33
15-721	2.89

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 Lecture

- Nested Loop Join
- Sort-Merge Join
- Hash Join



Next Lecture

- Nested Loop Join
- Sort-Merge Join
- Hash Join