

Carnegie Mellon University

# Database Systems

15-445/645 SPRING 2026

ANDY PAVLO

JIGNESH PATEL

Lecture #11

## Sorting & Aggregation Algorithms



# ADMINISTRIVIA

---



**Homework #3** is due Sunday Feb 22<sup>nd</sup> @ 11:59pm

**Mid-Term Exam** is on Wednesday Feb 25<sup>th</sup> @ 2:00pm

→ Lectures #01–11 (inclusive)

→ Study guide will be released later today (for real this time).

**Project #2** is due Sunday March 15<sup>th</sup> @ 11:59pm

# COURSE OUTLINE

---

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

---

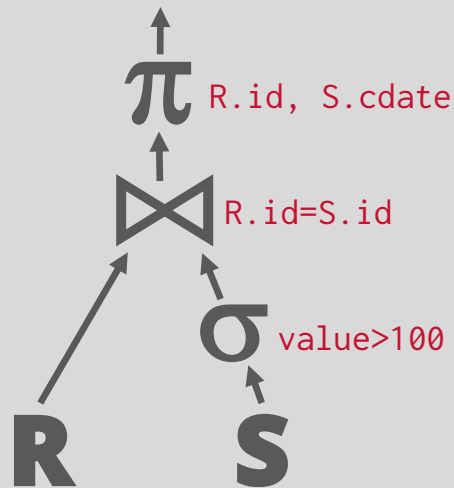
The operators are arranged in a tree.

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:

- Duplicate elimination (**DISTINCT**).
- Bulk loading sorted tuples into a B+Tree index is faster.
- Aggregations (**GROUP BY**), Window Functions

# SORTING ALGORITHMS



For a given input **run** (i.e., list of key/value pairs), sort it based on a comparison function and sorting parameters.

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

**Value:** Two choices

→ Tuple (early materialization).

→ Record ID (late materialization).

## Early Materialization

$K_1$	<i>&lt;Tuple Data&gt;</i>
$K_2$	<i>&lt;Tuple Data&gt;</i>

⋮

## Late Materialization



*Record ID /  
Offset*

# SORTING ALGORITHMS



For a given input run (i.e. key/value pairs), sort it with a comparison function and parameters.

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

**Value:** Two choices

- Tuple (early materialization).
- Record ID (late materialization).



# TODAY'S AGENDA

---



External Merge Sort

B+Tree Sorting

Top-N Heap Sort

Collations

Aggregations

↪ **DB Flash Talk: MotherDuck**

# TODAY



External Merge Sort

B+Tree Sorting

Top-N Heap Sort

Collations

Aggregations

↳ DB Flash Talk: Moth

## Redesigning DuckDB's Sort, Again



Laurens Kuiper

2025-09-24 · 16 min

**TL;DR:** After four years, we've decided to redesign DuckDB's sort implementation, again. In this post, we present and evaluate the new design.

DuckDB v1.4.0 was [just released](#), which includes a complete redesign of DuckDB's sort implementation. We [redesigned DuckDB's sort just four years ago](#), which allowed DuckDB to sort more data than fits in main memory, in parallel, with highly efficient comparisons. This implementation served us well, but since then we've implemented larger-than-memory query processing for more operators, such as the [hash join](#) and [hash aggregation](#), which both use a new and improved [spillable page layout](#). We presented this layout in an [earlier blog post](#). We decided to integrate this layout in DuckDB's sort, and [completely redesigned the implementation](#).

Not interested in the implementation? [Jump straight to the benchmark!](#)

# OBSERVATION

---

If data fits in memory, then the DBMS can use a standard sorting algorithm.

- Optimized algorithms if data is mostly sorted (VergeSort).
- Otherwise use your favorite (QuickSort, TimSort, RadixSort).
- Many online visualization tools.

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 ...

# 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.
- Pick your favorite in-memory sorting algorithm.

## **Phase #2 – Merging**

- Combine sorted runs into larger chunks.

# 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 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 <sup>12</sup>

---

## 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, the DBMS reads and writes every page in the file.

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

Number of passes

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

Total I/O cost

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

# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

13

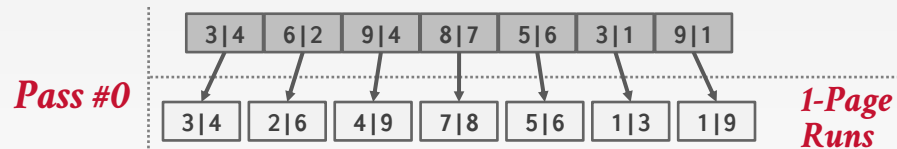
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

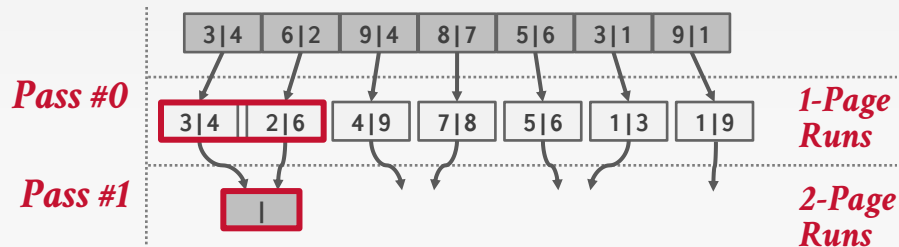
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

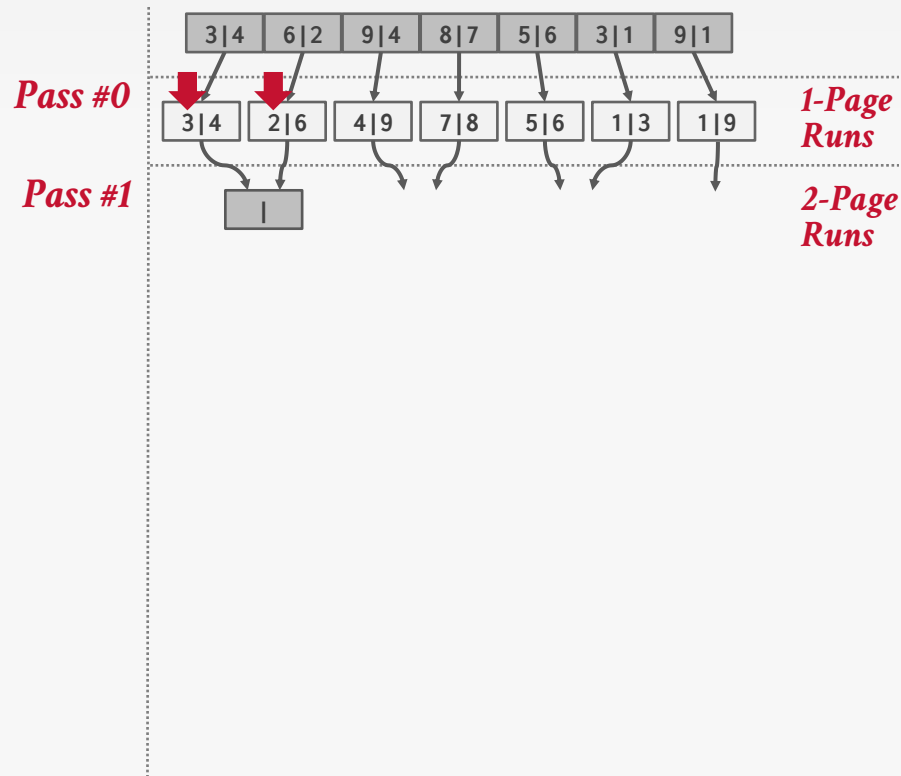
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

13

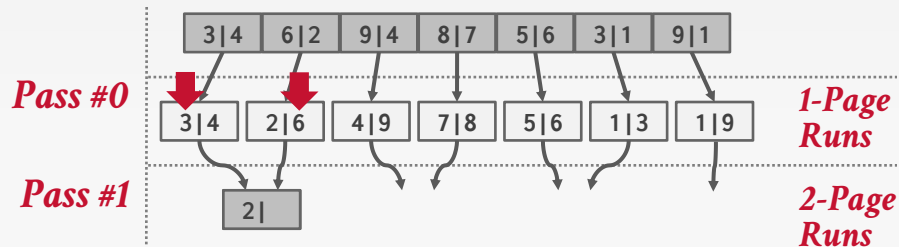
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

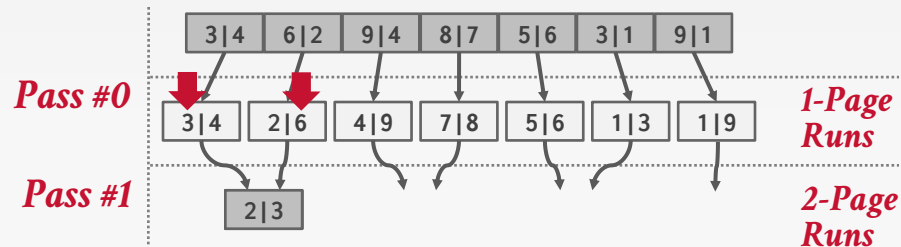
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT 13

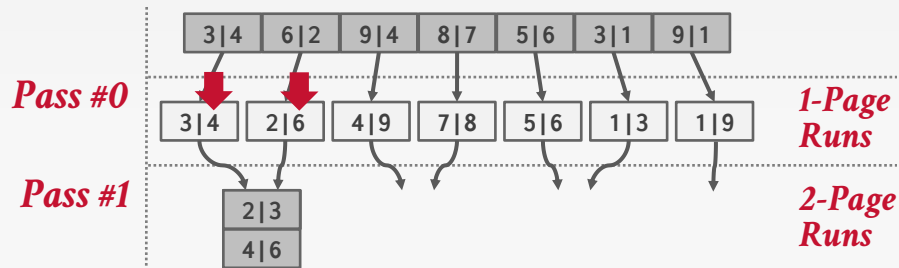
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

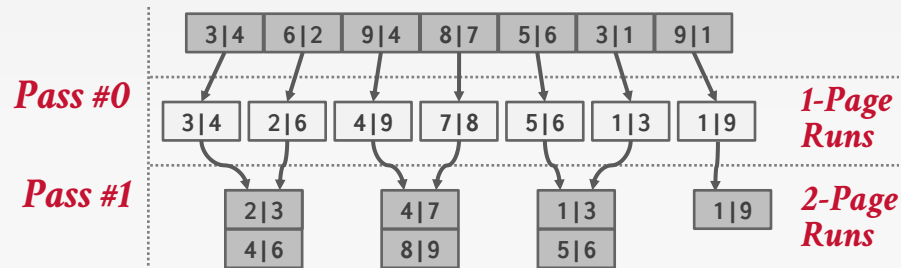
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

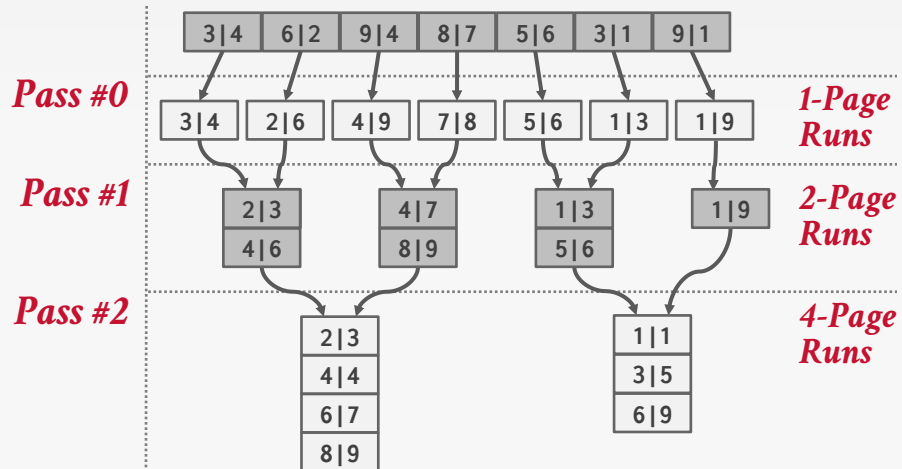
In each pass, the DBMS reads and writes every page in the file.

Number of passes

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

Total I/O cost

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



# SIMPLIFIED 2-WAY EXTERNAL MERGE SORT

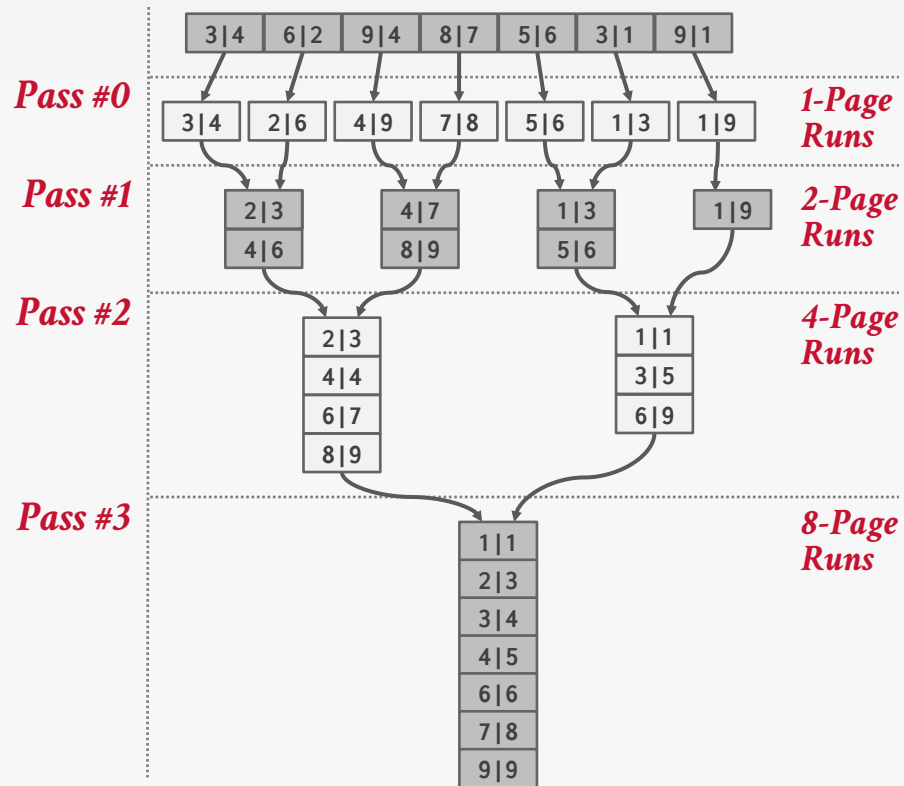
In each pass, the DBMS reads and writes every page in the file.

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...

# EXAMPLE

---

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

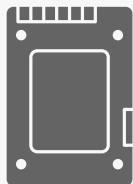
- **Pass #0:**  $\lceil N / B \rceil = \lceil 108 / 5 \rceil = 22$  sorted runs of 5 pages each (last run is only 3 pages).
- **Pass #1:**  $\lceil N' / B-1 \rceil = \lceil 22 / 4 \rceil = 6$  sorted runs of 20 pages each (last run is only 8 pages).
- **Pass #2:**  $\lceil N'' / B-1 \rceil = \lceil 6 / 4 \rceil = 2$  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

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

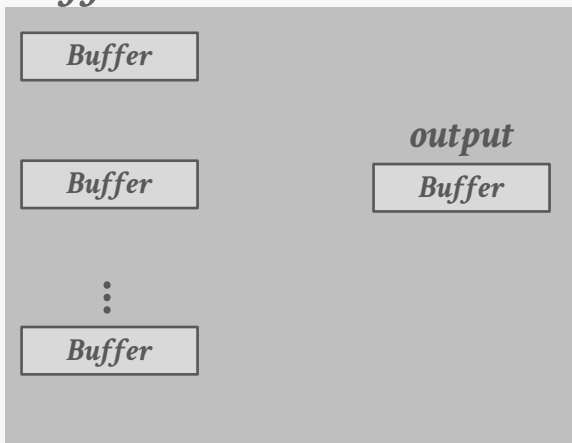
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



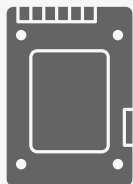
## *Buffer Pool*



# DOUBLE BUFFERING

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

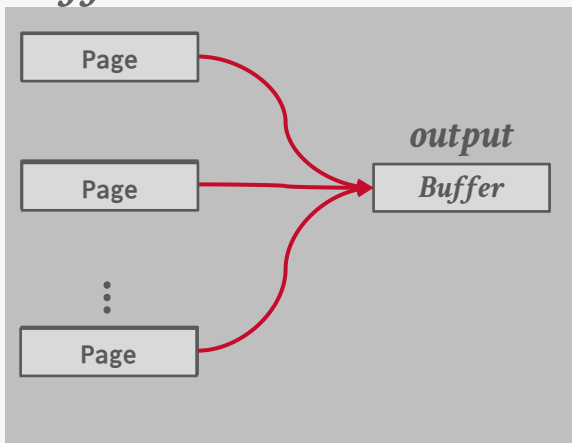
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



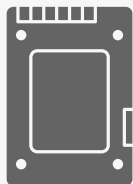
*Buffer Pool*



# DOUBLE BUFFERING

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

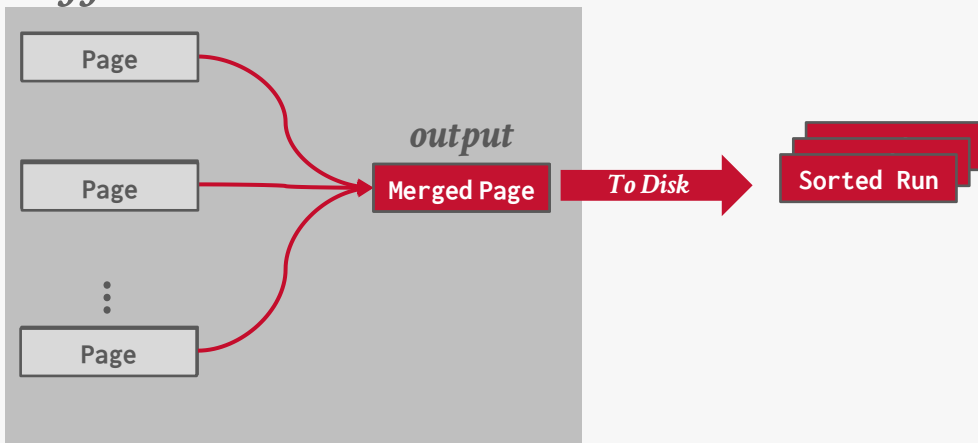
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



## *Buffer Pool*

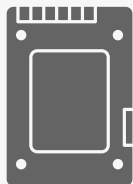


# DOUBLE BUFFERING

---

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

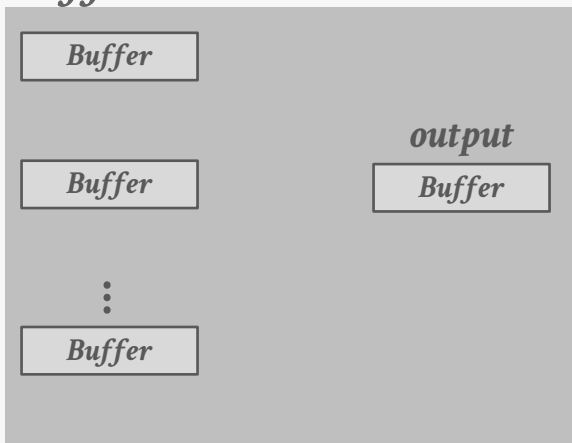
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



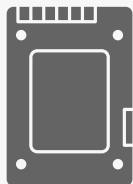
## *Buffer Pool*



# DOUBLE BUFFERING

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

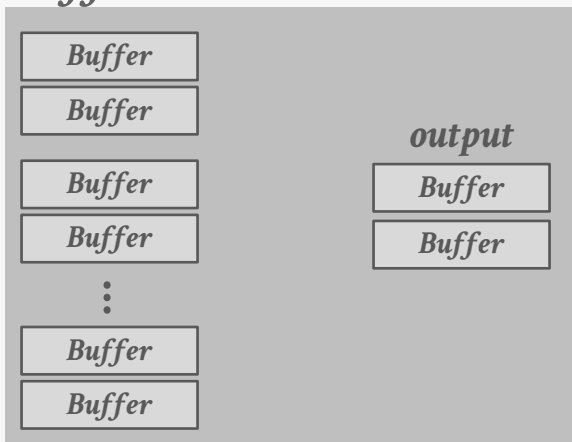
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



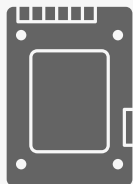
## *Buffer Pool*



# DOUBLE BUFFERING

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

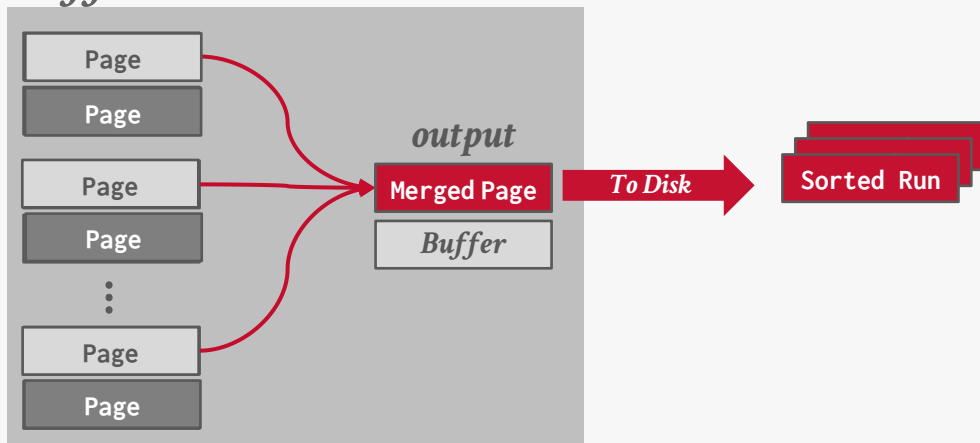
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



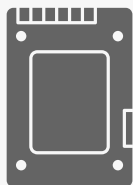
## *Buffer Pool*



# DOUBLE BUFFERING

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

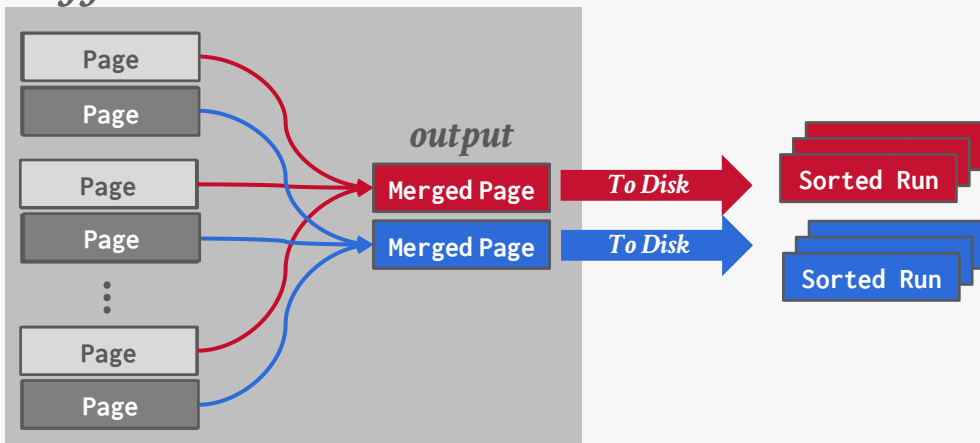
- Overlap CPU and I/O operations
- Reduces effective buffers available by half



*Disk*



## *Buffer Pool*



# B+TREES 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.

→ Some DBMSs support prefix key scans for sorting.

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

Cases to consider:

→ Clustered B+Tree

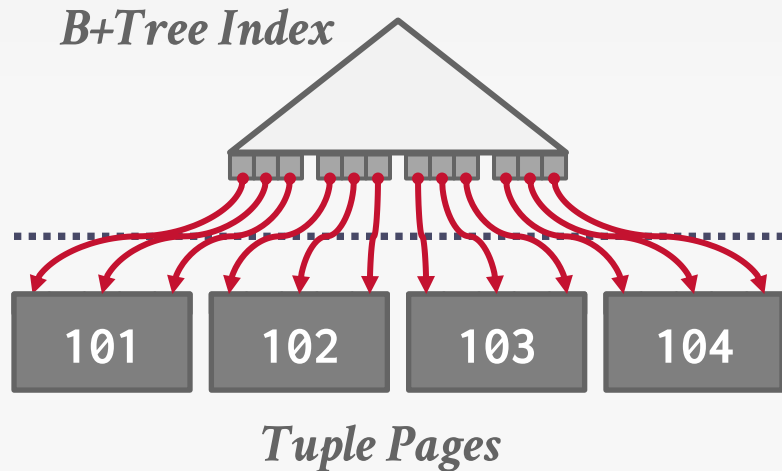
→ Unclustered B+Tree

# CASE #1: CLUSTERED B+TREE

---

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

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



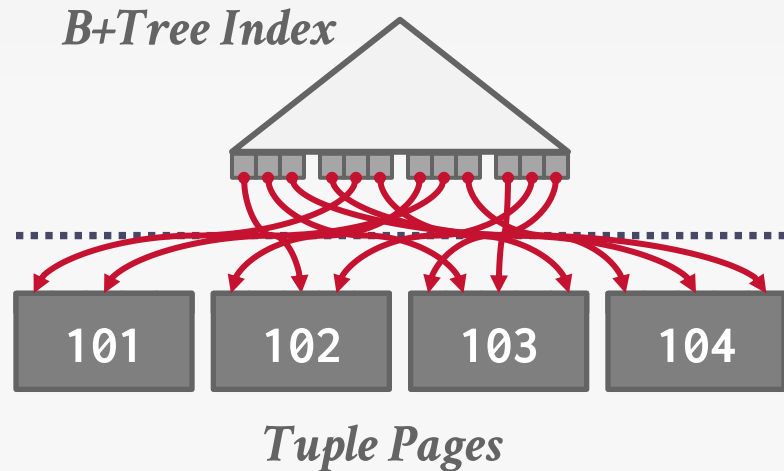
# CASE #2: UNCLUSTERED B+TREE

---

Chase each pointer to the page that contains the data.

This is almost always a bad idea except for Top-N queries where N is small enough relative to total number of tuples in table.

→ In general, one I/O per data record.



# 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 ASC
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 ASC
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 ASC
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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Skip!*

*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Sorted Heap*

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

# 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 ASC
FETCH FIRST 4 ROWS
WITH TIES
```

*Original Data*

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



*Skip and done!*

*Sorted Heap*

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

# COLLATIONS

A **collation** defines the rules for comparing and sorting character strings.

- Character ordering (**ORDER BY**)
- Equality comparison (**=, LIKE**)
- Case sensitivity
- Accent sensitivity
- Locale-specific rules

Binary collations are the fastest, but string comparison is not universal  
 → ASCII Order  $\neq$  Human Linguistic Order

id	name
1	banana
2	Ängel
3	zebra
4	Apple

*Sort!*

*ASCII Collation*

id	name
4	Apple
1	banana
3	zebra
2	Ängel

*Non-ASCII  
Sorted Last*

*German Collation*

id	name
2	Ängel
4	Apple
1	banana
3	zebra

*Handle "Ä"  
Similar to "Ae"*

# LOCALE-AWARE COMPARISON

---

Locale-aware comparison in ICU is based on the Unicode Collation Algorithm (UCA).

Rather than comparing raw Unicode code points, ICU converts strings into collation elements with multiple comparison levels and then compares those elements lexicographically.

→ Each character is mapped to one or more collation elements.

<u>Level</u>	<u>Meaning</u>	<u>Example</u>
<u>Primary</u>	Base letter differences	<b>a vs. b</b>
<u>Secondary</u>	Accent differences	<b>e vs. é</b>
<u>Tertiary</u>	Case differences	<b>a vs. A</b>
<u>Quaternary</u>	Punctuation / tie-breaking	<i>Optional</i>

# LOCALE-AWARE COMPARISON

Comparison proceeds

lexicographically by level:

- Compare **primary weights** (base letters).
- If equal, compare **secondary weights** (accents).
- If equal, compare **tertiary weights** (case).
- Continue if needed.

Since doing this multi-step comparison is expensive, the DBMS will precompute sort keys.

Level	Meaning	Example
<u>P</u> Primary	Base letter differences	<b>a</b> vs. <b>b</b>
<u>S</u> Secondary	Accent differences	<b>e</b> vs. <b>é</b>
<u>T</u> Tertiary	Case differences	<b>a</b> vs. <b>A</b>
<u>Q</u> Quaternary	Punctuation / tie-breaking	<i>Optional</i>

**a** → (P1, S0, T1)

**A** → (P1, S0, T2)

**á** → (P1, S1, T1)

# 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.

## Approach #3: Key Normalization

→ Transform variable-length attribute(s) into a single encoded/padded string that preserves sort order.

# 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

  
*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

```
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*

  
*Eliminate  
Duplicates*

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

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

  
*Eliminate  
Duplicates*

# AGGREGATION 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)

  
*Join*

cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7

  
*Sort*

cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: null

  
(1, 3.62)

# AGGREGATION 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)

  
*Join*

cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7

  
*Sort*

cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: 15-445

  
(1, 3.62)

# AGGREGATION 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)

  
*Join*

cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7

  
*Sort*

cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: 15-445

  
(2, 7.32)

# AGGREGATION 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)



cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7



cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: 15-445

(2, 7.32)

# AGGREGATION 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)



cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7



cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: 15-445

(2, 7.32)

## Final Result

cid	AVG(gpa)

# AGGREGATION 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)



cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7



cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: 15-721

(1, 3.33)

## Final Result

cid	AVG(gpa)
15-445	3.66

# AGGREGATION 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)



cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7



cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89

Previous: 15-721

(1, 3.33)

## Final Result

cid	AVG(gpa)
15-445	3.66

# AGGREGATION 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)



cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7



cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89



Previous: 15-826

(1, 2.89)

## Final Result

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

# AGGREGATION 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)

  
*Join*

cid	s.gpa
15-445	3.62
15-826	2.89
15-721	3.33
15-445	3.7

  
*Sort*

cid	s.gpa
15-445	3.62
15-445	3.7
15-721	3.33
15-826	2.89



## Final Result

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

# 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

---

Divide-and-conquer approach to computing an aggregation when data does not fit in memory.

## **Phase #1 – Partition**

- Split 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

  
*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**

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

0

15-826
15-210

1

⋮

15-721
--------

**B-1**

# 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

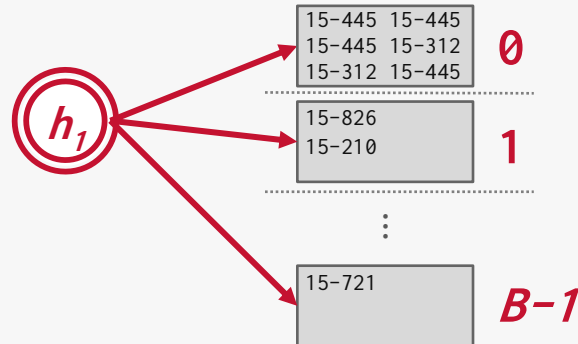
**Filter**

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

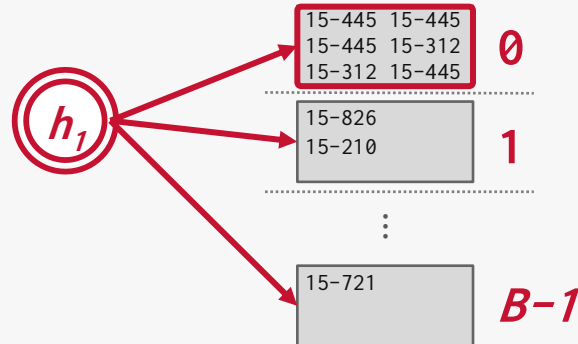
**Filter**

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 #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

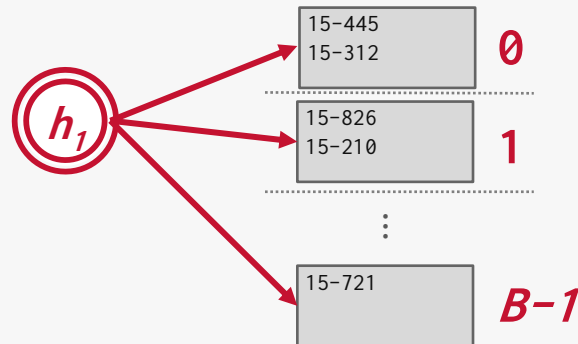
**Filter**

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.

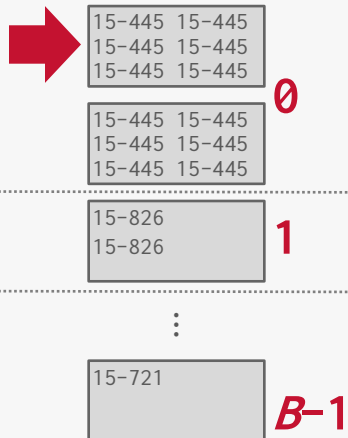
# PHASE #2: REHASH

```
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 #1 Buckets



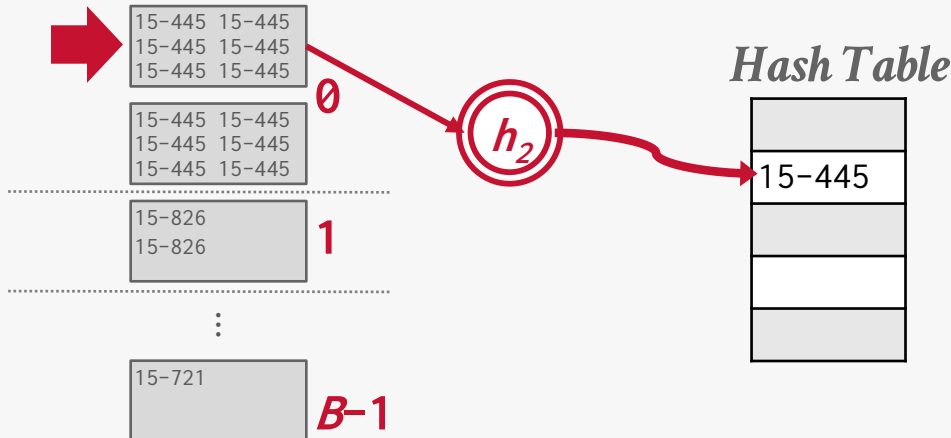
# PHASE #2: REHASH

```
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



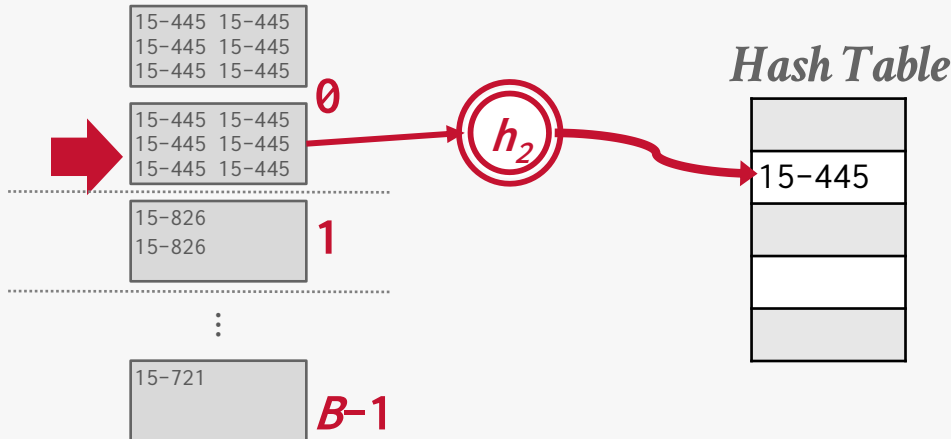
# PHASE #2: REHASH

```
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



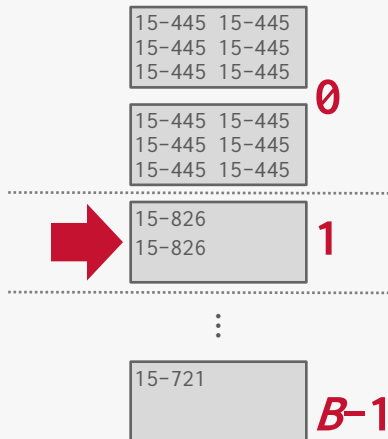
# PHASE #2: REHASH

```
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



$h_2$

## Hash Table

15-445

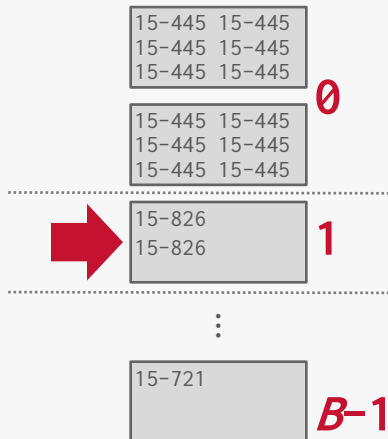
# PHASE #2: REHASH

```
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



$h_2$

## Hash Table

15-445

## Final Result

cid
15-445

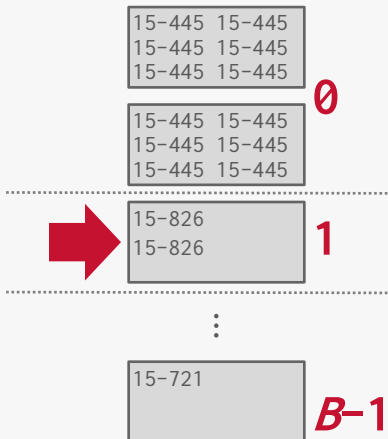
# PHASE #2: REHASH

```
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


## Final Result

cid
15-445

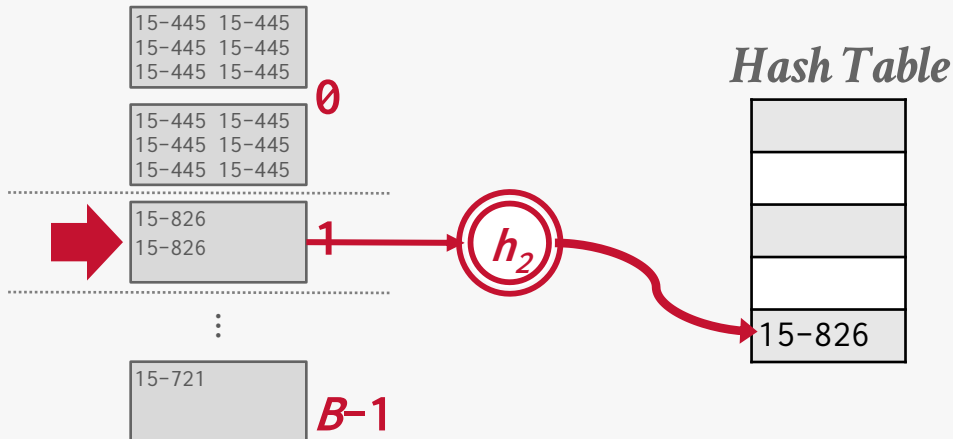
# PHASE #2: REHASH

```
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

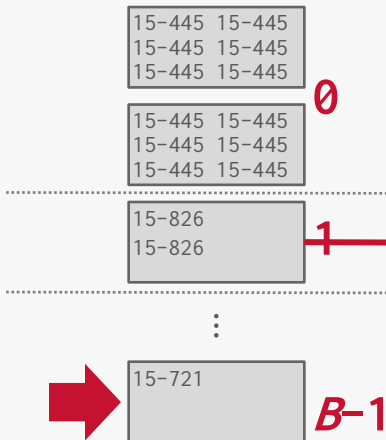
# PHASE #2: REHASH

```
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

15-826

## Final Result

cid
15-445
15-826

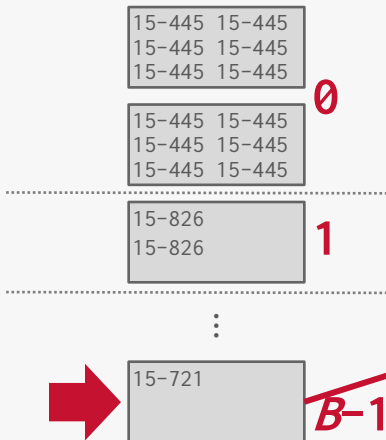
# PHASE #2: REHASH

```
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

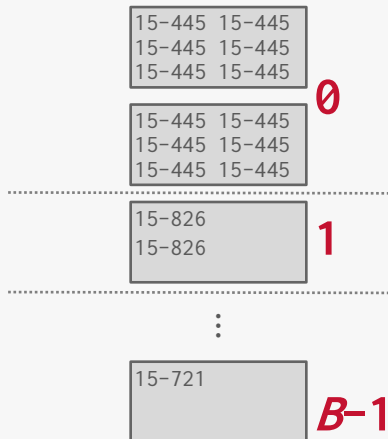
# PHASE #2: REHASH

```
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
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 as we compute the aggregate:

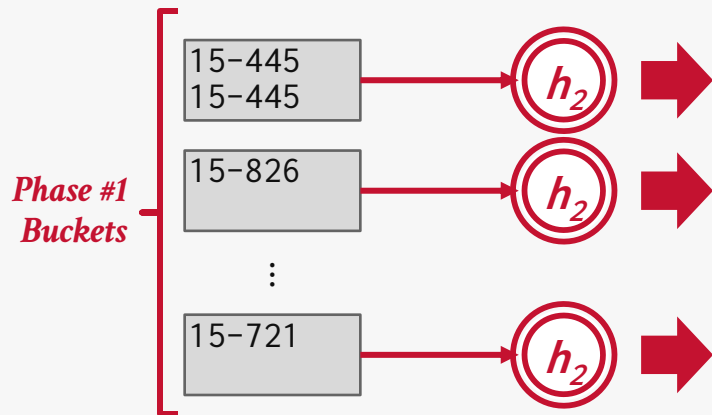
- If we find a matching **GroupKey**, just update the **RunningVal** appropriately
- Else insert a new **GroupKey**→**RunningVal**

# HASHING SUMMARIZATION

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

## *Running Totals*

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

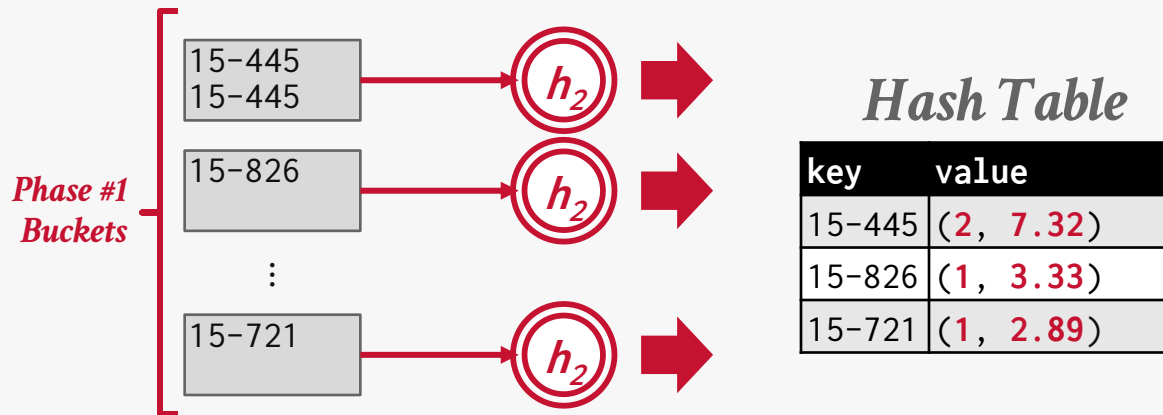


# HASHING SUMMARIZATION

```
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)

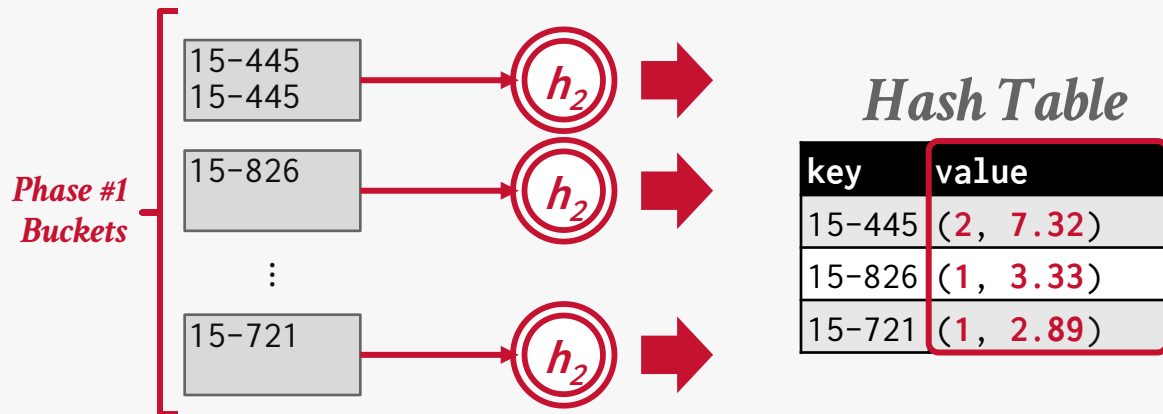


# HASHING SUMMARIZATION

```
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)

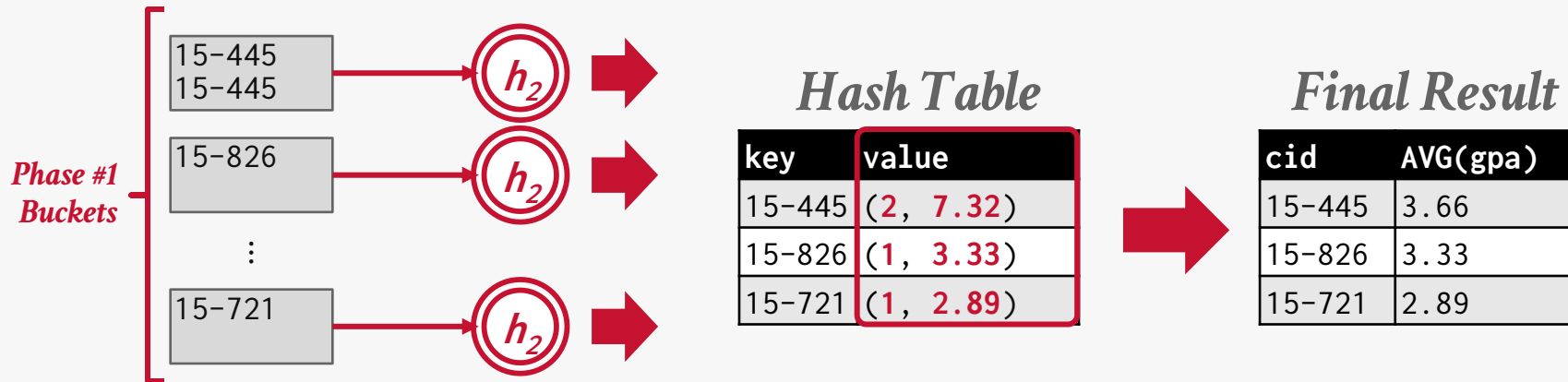


# HASHING SUMMARIZATION

```
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)



# CONCLUSION

---

External merge-sort allows the DBMS to handle data sets larger than memory.

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

# NEXT CLASS

---

Nested Loop Join

Sort-Merge Join

Hash Join