Carnegie Mellon University

10 Sorting & Aggregations

Intro to Database Systems
15-445/15-645
Fall 2019

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Homework #3 is due Wed Oct 9th @ 11:59pm

Mid-Term Exam is Wed Oct 16th @ 12:00pm

Project #2 is due Sun Oct 20th @ 11:59pm

COURSE STATUS

We are now going to talk about how to execute queries using table heaps and indexes.

Next two weeks:
→ Operator Algorithms
→ Query Processing Models
→ Runtime Architectures

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager

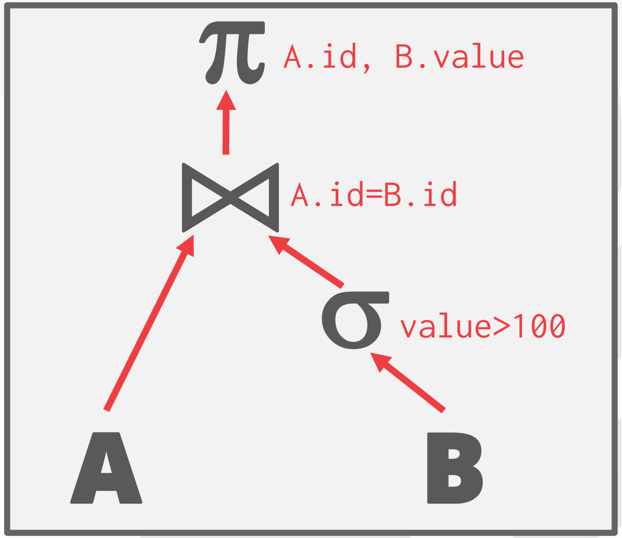
QUERY PLAN

The operators are arranged in a tree.

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

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

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



DISK-ORIENTED DBMS

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

We are going use on 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 access.

TODAY'S AGENDA

External Merge Sort
Aggregations

WHY DO WE NEED TO SORT?

Tuples in a table have no specific order.

But queries often want to retrieve tuples in a specific order.

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

SORTING ALGORITHMS

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

If data does not fit in memory, then we need to use a technique that is aware of the cost of writing data out to disk…

EXTERNAL MERGE SORT

Divide-and-conquer sorting algorithm that splits the data set into separate **runs** and then sorts them individually.

Phase #1 – Sorting

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

Phase #2 – Merging

→ Combine sorted sub-files into a single larger file.

2-WAY EXTERNAL MERGE SORT

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

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

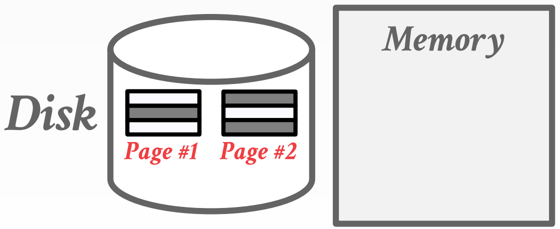
Data set is broken up into ***N*** pages.

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

2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.

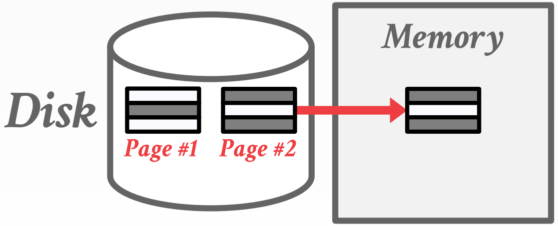


2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read every ***B*** pages of the table into memory

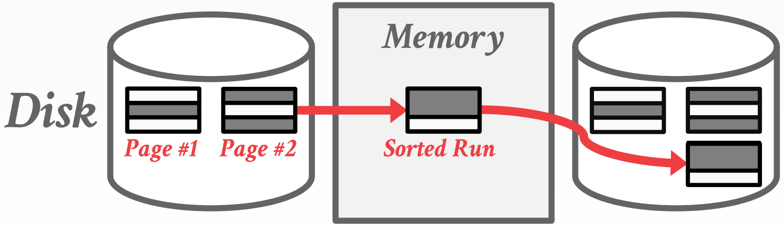
→ Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

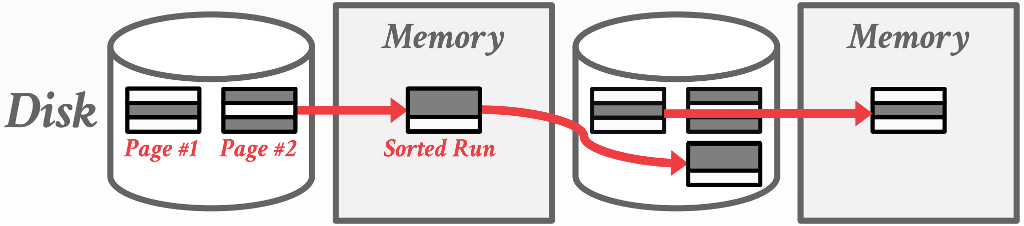
→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

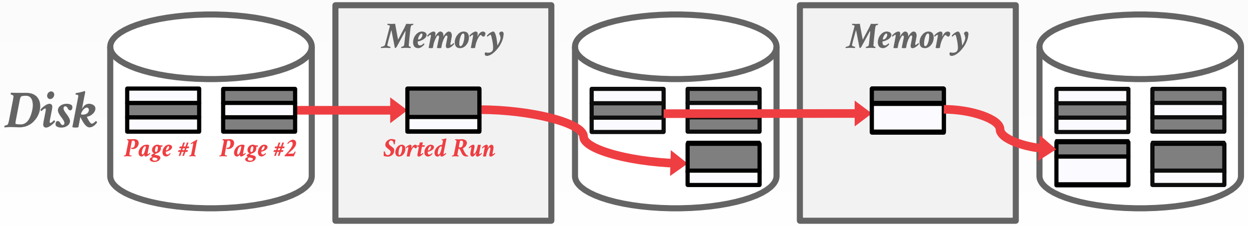
→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.



2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.



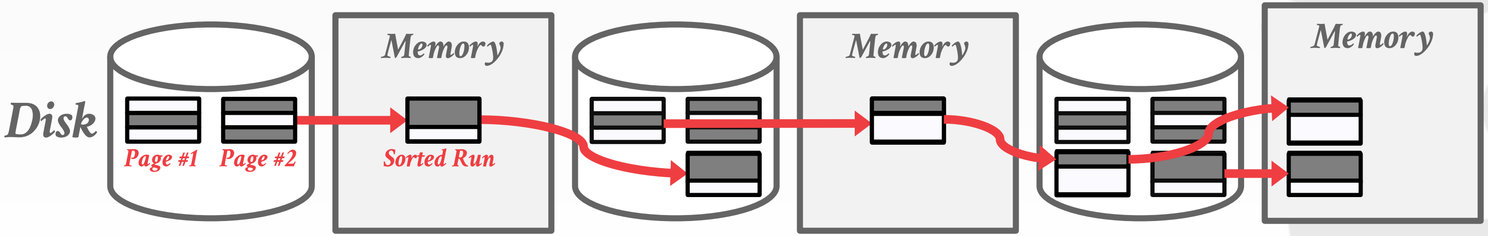
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.

Pass #1,2,3,…

→ Recursively merges pairs of runs into runs twice as long.
→ Uses three buffer pages (2 for input pages, 1 for output).



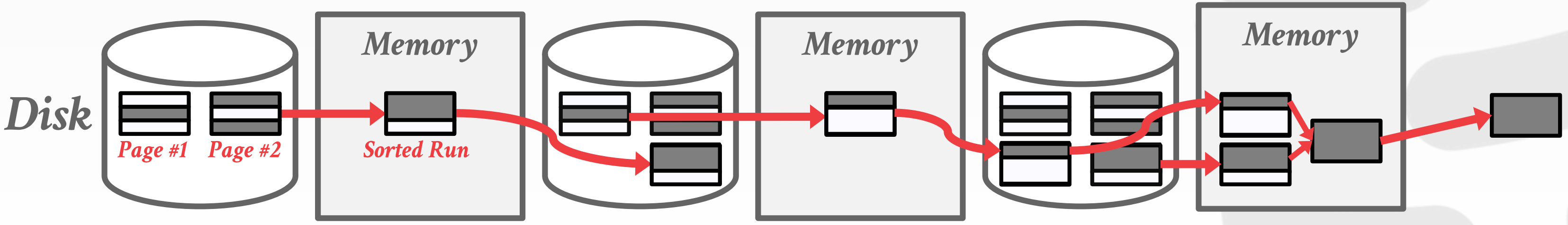
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.

Pass #1,2,3,…

→ Recursively merges pairs of runs into runs twice as long.
→ Uses three buffer pages (2 for input pages, 1 for output).



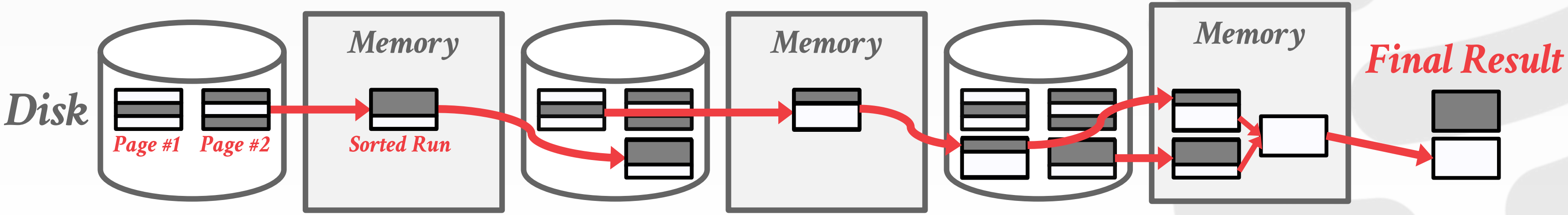
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read every ***B*** pages of the table into memory
→ Sort pages into runs and write them back to disk.

Pass #1,2,3,…

→ Recursively merges pairs of runs into runs twice as long.
→ Uses three buffer pages (2 for input pages, 1 for output).



2-WAY EXTERNAL MERGE SORT

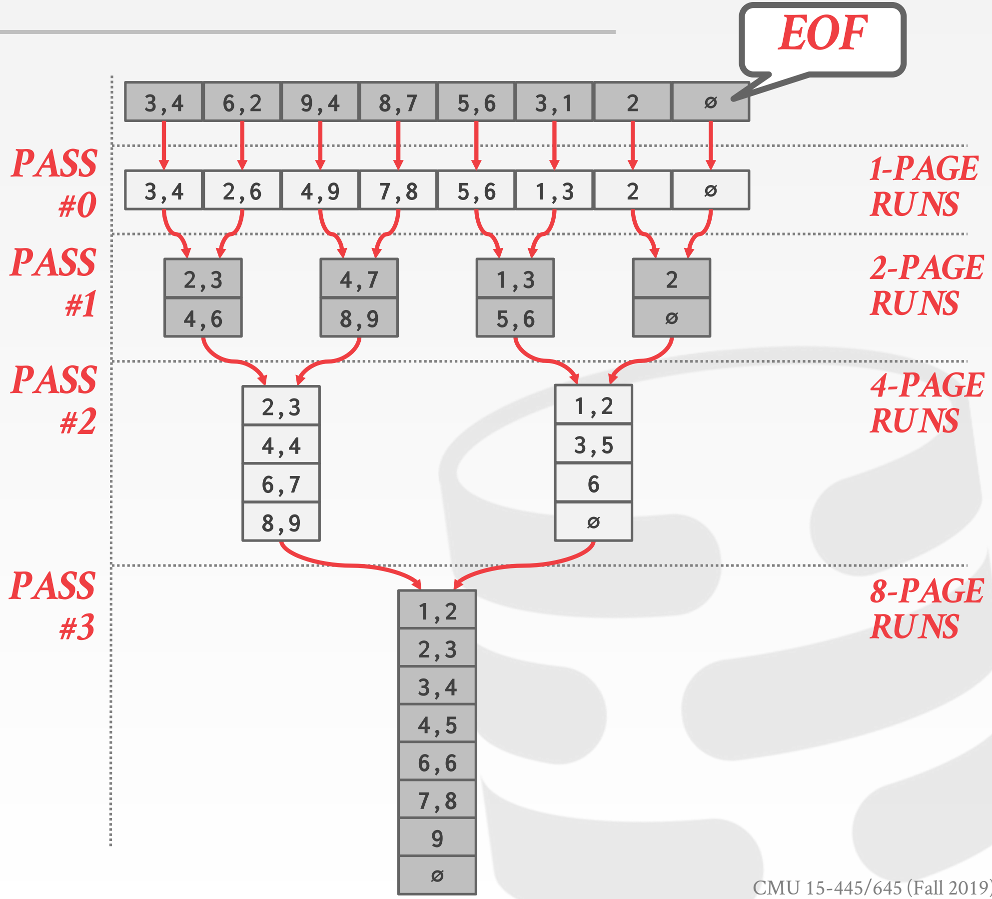
In each pass, we read and write each page in file.

Number of passes

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

Total I/O cost

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



2-WAY EXTERNAL MERGE SORT

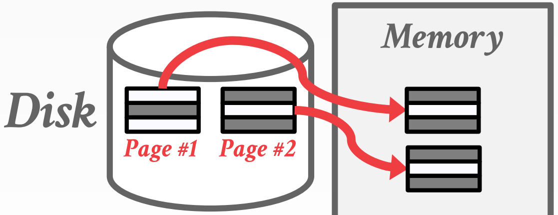
This algorithm only requires three buffer pages to perform the sorting (***B*=3**).

But even if we have more buffer space available (***B*>3**), it does not effectively utilize them…

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.

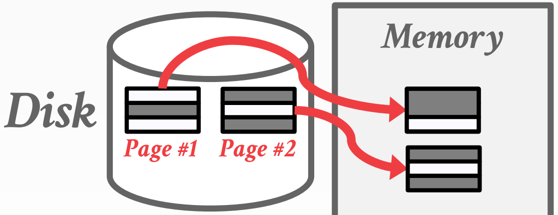
→ Reduces the wait time for I/O requests at each step by continuously utilizing the disk.



DOUBLE BUFFERING 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.



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)

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

Sort 108 pages with 5 buffer 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.

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

$$= \textbf{4 passes}$$

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.

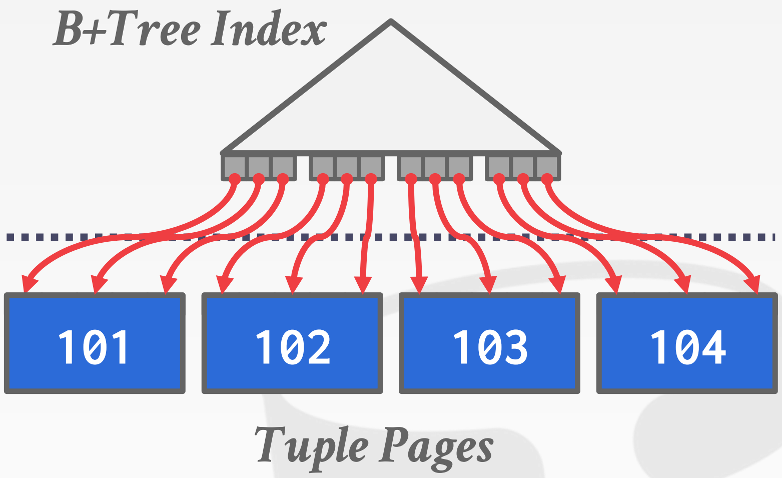
Retrieve tuples in desired sort order by simply traversing the leaf pages of the tree.

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.

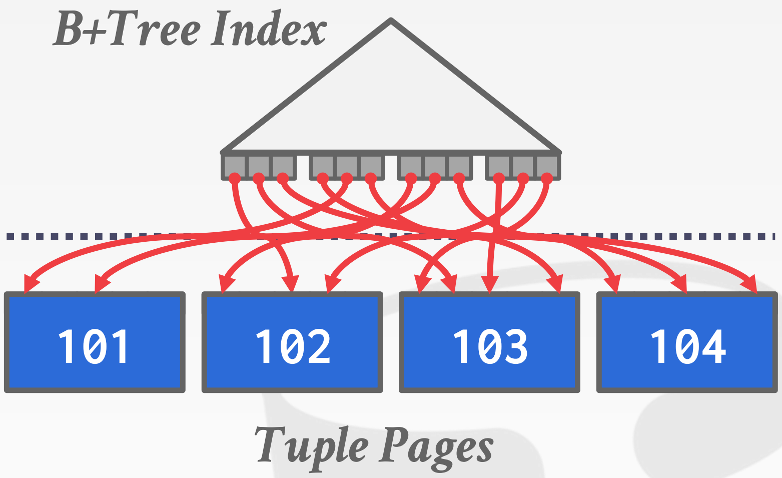
This is always better than external sorting because there is no computational cost and all disk access is sequential.



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 multiple tuples into a single scalar value.

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

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

Eliminate Dupes

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 Dupes

ALTERNATIVES TO SORTING

What if we don't 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 it 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.

→ We know that all matches live in the same partition.

→ 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

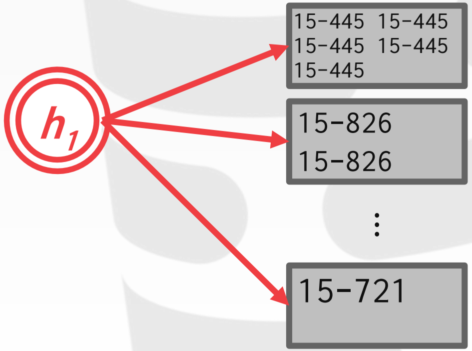
Filter

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

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')
```

Phase #1 Buckets

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 #1 Buckets

15-445 15-445 15-445 15-445 15-445
15-826 15-826

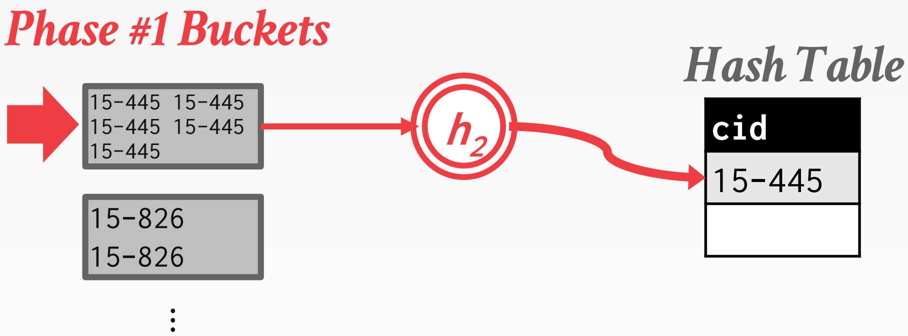
⋮

PHASE #2 – REHASH

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

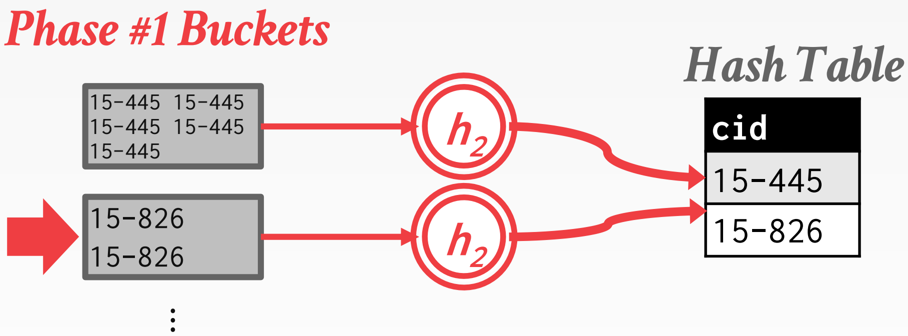
enrolled(sid,cid,grade)

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



PHASE #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

15-826
15-826

⋮

h_2

h_2

Hash Table

cid
15-445
15-826

Final Result

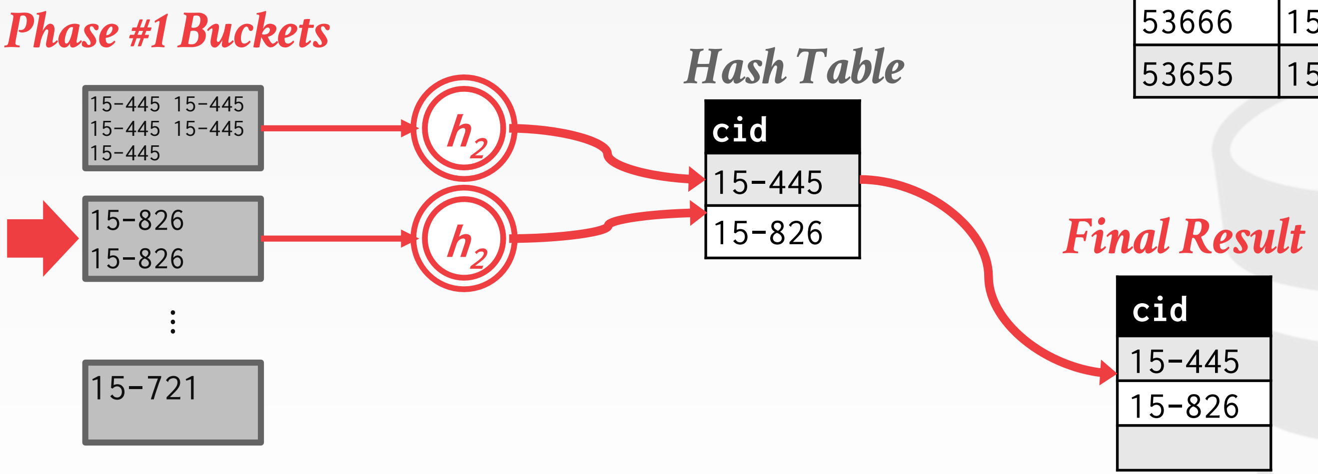
cid
15-445
15-826

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



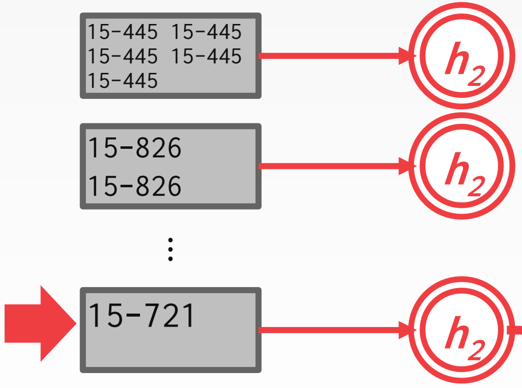
PHASE #2 – REHASH

```
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

cid
15-721

Final Result

cid
15-445
15-826
15-721

HASHING SUMMARIZATION

During the ReHash phase, store pairs of the form (`GroupKey→RunningVal`)

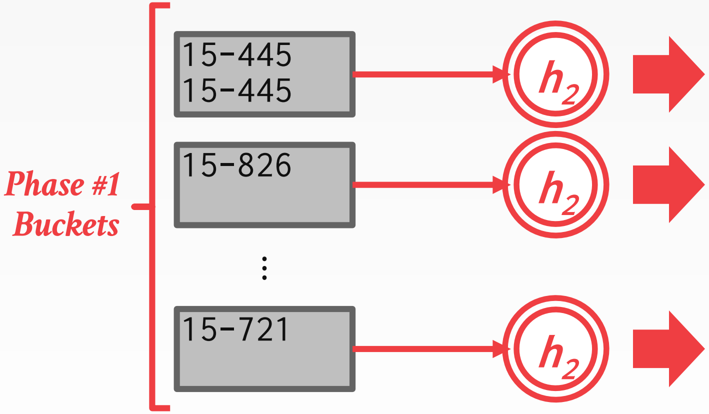
When we want to insert a new tuple into the hash table:

→ 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
```



Hash Table

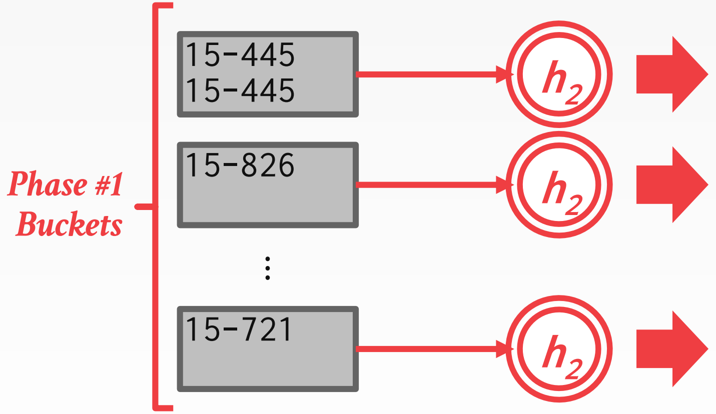
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

HASHING SUMMARIZATION

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

Running Totals

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



Hash Table

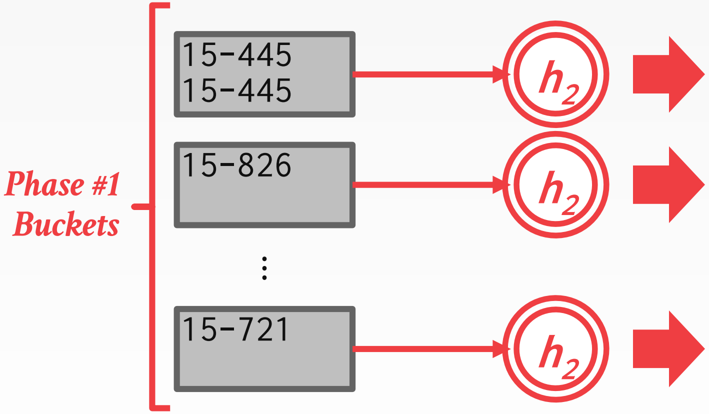
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

HASHING SUMMARIZATION

```
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

COST ANALYSIS

How big of a table can we hash using this approach?

→ B-1 "spill partitions" in Phase #1

→ Each should be no more than B blocks big

Answer: $B \cdot (B\text{-}1)$

→ A table of N pages needs about **sqrt(N)** buffers

→ Assumes hash distributes records evenly. Use a "fudge factor" $f>1$ for that: we need $B \cdot \mathrm{sqrt}(f \cdot N)$

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 seek+RD costs.
→ Double-buffering to overlap CPU and I/O.

PROJECT #2

You will build a thread-safe linear probing hash table that supports automatic resizing.

We define the API for you. You need to provide the implementation.

https://15445.courses.cs.cmu.edu/fall2019/project2/

PROJECT #2 – TASKS

Page Layouts

Hash Table Implementation

Table Resizing

Concurrency Control Protocol

DEVELOPMENT HINTS

Follow the textbook semantics and algorithms.

You should make sure your page layout are working correctly before switching to the actual hash table itself.

Then focus on the single-threaded use case first.

Avoid premature optimizations.
→ Correctness first, performance second.

THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Rebase on top of the latest BusTub master branch.

Post your questions on Piazza or come to TA office hours.

PLAGIARISM WARNING

Your project implementation must be your own work.

→ You may **not** copy source code from other groups or the web.

→ Do **not** publish your implementation on Github.

Plagiarism will **not** be tolerated.
See CMU's Policy on Academic Integrity for additional information.

NEXT CLASS

Nested Loop Join

Sort-Merge Join

Hash Join