Carnegie Mellon University

Sorting & Aggregations

Lecture #11

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

TODAY'S AGENDA

Sorting Algorithms

Aggregations

WHY DO WE NEED TO SORT?

Tuples in a table have no specific order

But users 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

Sorting Phase

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

Merge Phase

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

sort each chunk and then do k-way merge

OVERVIEW

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

Files are broken up into ***N*** pages.

The DBMS has a finite number of ***B*** fixed-size buffers.

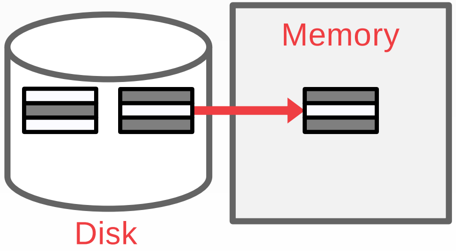
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Reads every ***B*** pages of the table into memory
→ Sorts them, and writes them back to disk.
→ Each sorted set of pages is called a ***run***.

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)



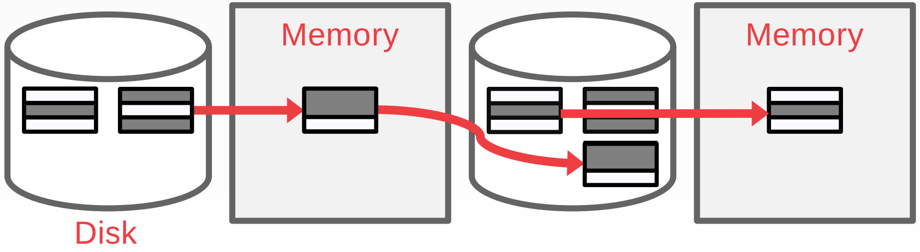
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Reads every ***B*** pages of the table into memory
→ Sorts them, and writes them back to disk.
→ Each sorted set of pages is called a ***run***.

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)



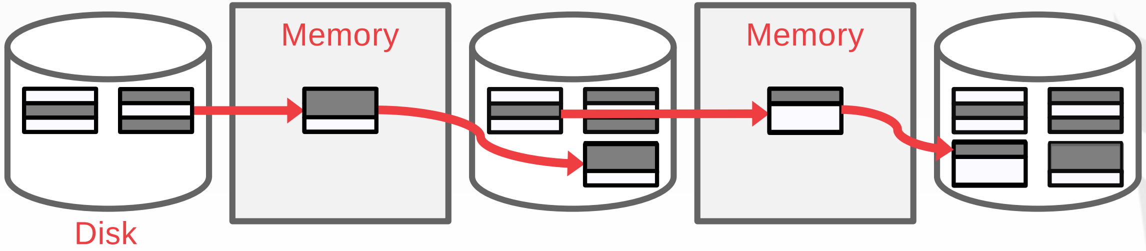
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Reads every ***B*** pages of the table into memory
→ Sorts them, and writes them back to disk.
→ Each sorted set of pages is called a ***run***.

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)



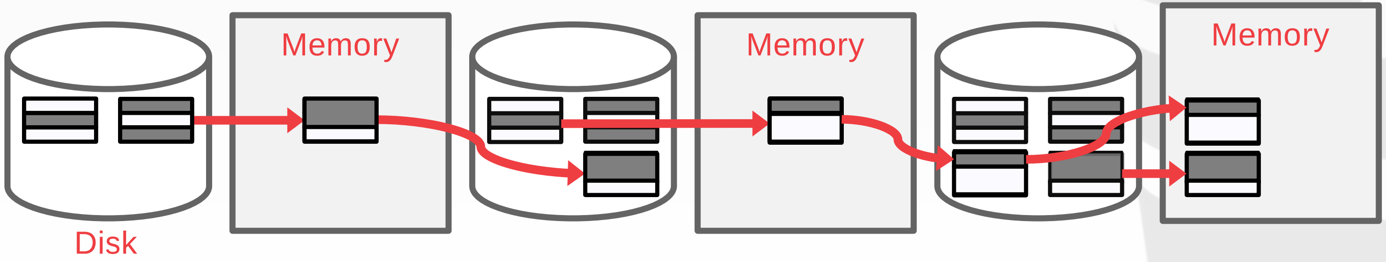
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Reads every ***B*** pages of the table into memory
→ Sorts them, and writes them back to disk.
→ Each sorted set of pages is called a ***run***.

Pass #1,2,3,…

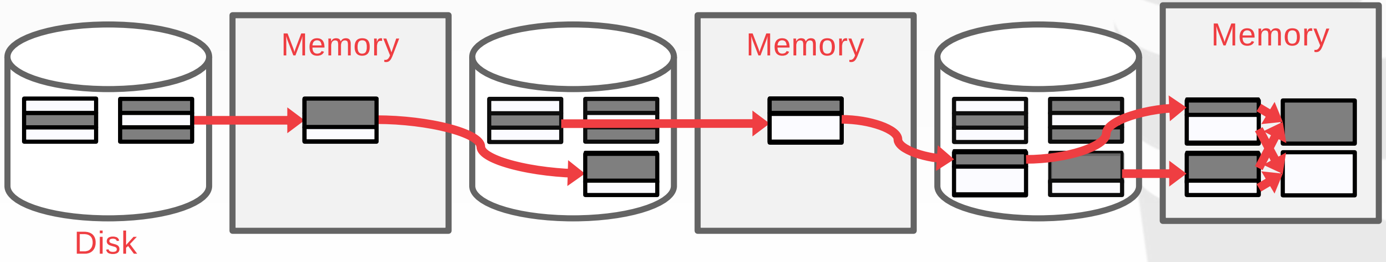
→ Recursively merges pairs of runs into runs twice as long
→ Uses three buffer pages (2 for input pages, 1 for output)



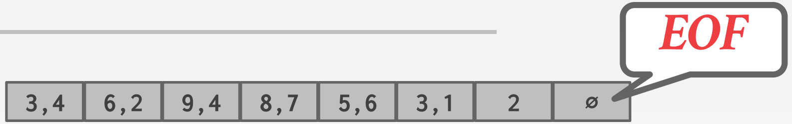
2-WAY EXTERNAL MERGE SORT

Pass #0
- → Reads every ***B*** pages of the table into memory
- → Sorts them, and writes them back to disk.
- → Each sorted set of pages is called a ***run***.

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

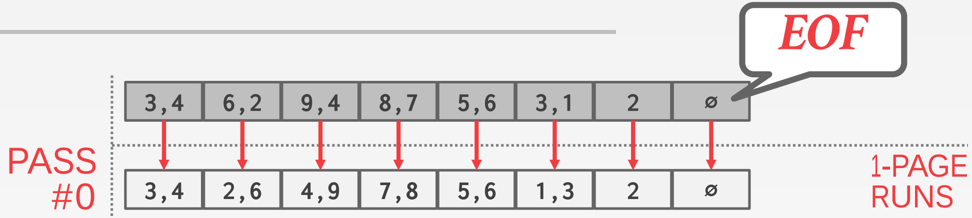
Total I/O cost
$= 2N \cdot (\text{\# of passes})$

2-WAY EXTERNAL MERGE SORT

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

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

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

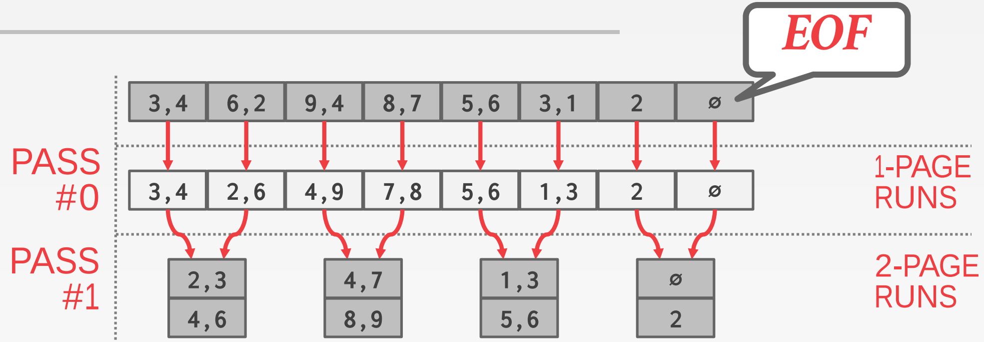


2-WAY EXTERNAL MERGE SORT

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

Number of passes
$= \mathbf{1 + \lceil \log_2 N \rceil}$

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



2-WAY EXTERNAL MERGE SORT

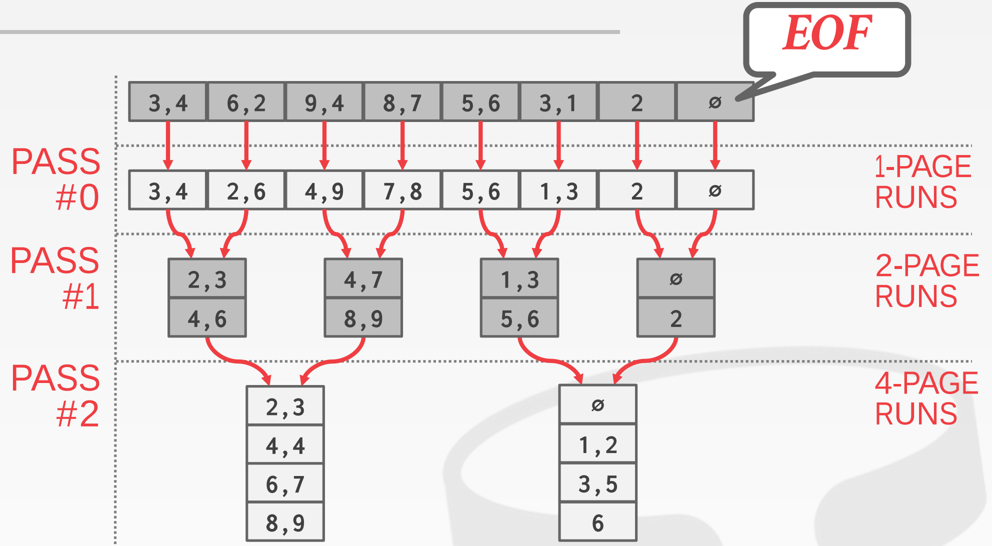
In each pass, we read and write each page in file.

Number of passes

$= \mathbf{1 + \lceil \log_2 N \rceil}$

Total I/O cost

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



2-WAY EXTERNAL MERGE SORT

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

Number of passes

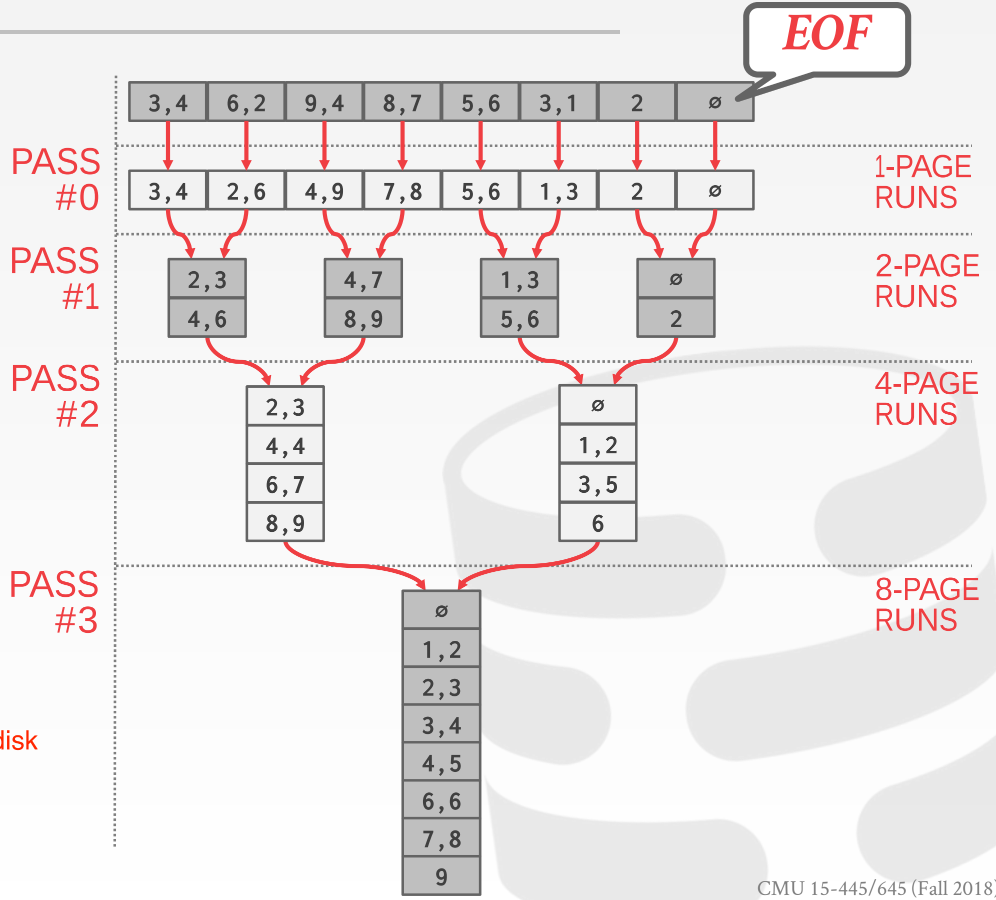
$= 1 + \lceil \log_2 N \rceil$

Total I/O cost

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

1 N for read and another for write

always two pages for input and one page for output, thinking it as a stream, when output page is full, write it out and get a fresh new one from disk



2-WAY EXTERNAL MERGE SORT

This algorithm only requires three buffer pages (*B*=3).

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

Let's next generalize the algorithm to make use of extra buffer space.

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)

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

EXAMPLE

Sort 108 page file 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, 80 pages and 28 pages

→ **Pass #3:** Sorted file of 108 pages

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

USING B+TREES

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.

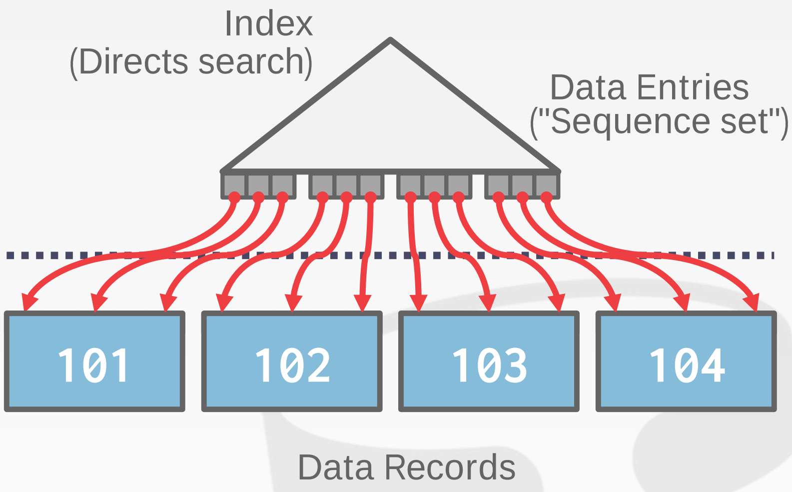
Cases to consider:

→ 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.

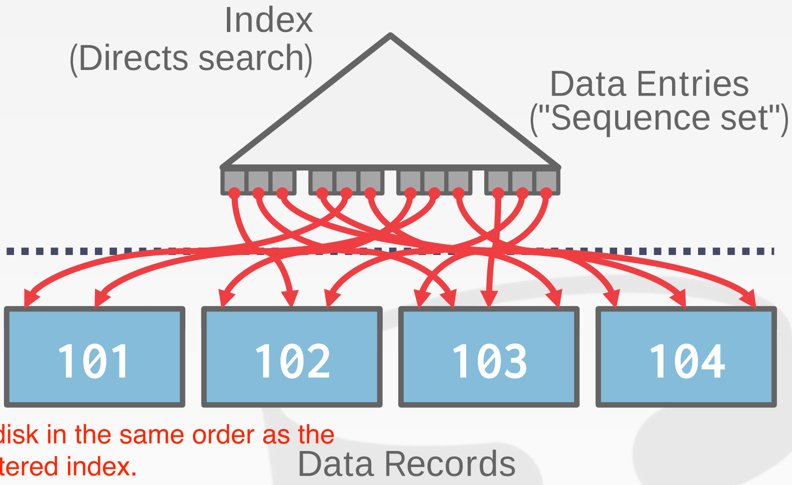
This will always better than external sorting.



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.



With a clustered index the rows are stored physically on the disk in the same order as the index. Therefore, there can be only one clustered index.

With a non clustered index there is a second list that has pointers to the physical rows. You can have many non clustered indices, although each new index will increase the time it takes to write new records.

It is generally faster to read from a clustered index if you want to get back all the columns. You do not have to go first to the index and then to the table.

Writing to a table with a clustered index can be slower, if there is a need to rearrange the data.

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')
```

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

SORTING AGGREGATION

```
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

SORTING AGGREGATION

```
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

Sort

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

SORTING AGGREGATION

```
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

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)

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's easy.

If we have to spill to disk, then we need to be smarter…

HASHING AGGREGATE

Partition Phase

→ Divide tuples into buckets based on hash key.

ReHash Phase

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

HASHING AGGREGATE 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.

HASHING AGGREGATE 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

HASHING AGGREGATE 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

HASHING AGGREGATE PHASE #1: PARTITION

```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

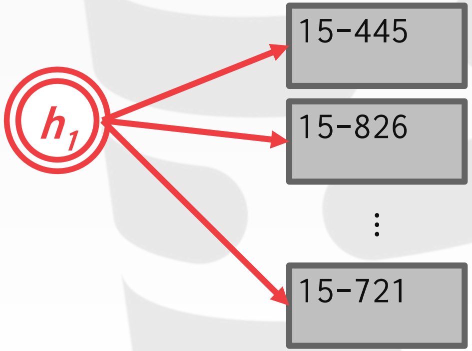
Filter

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



HASHING AGGREGATE 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.

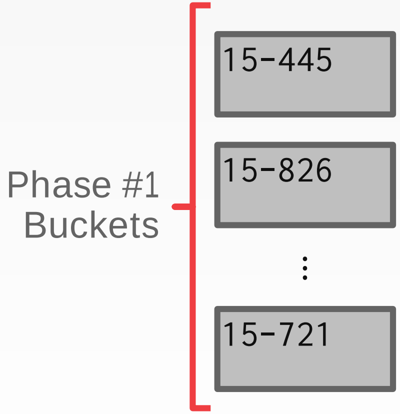
This assumes that each partition fits in memory.

HASHING AGGREGATE 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

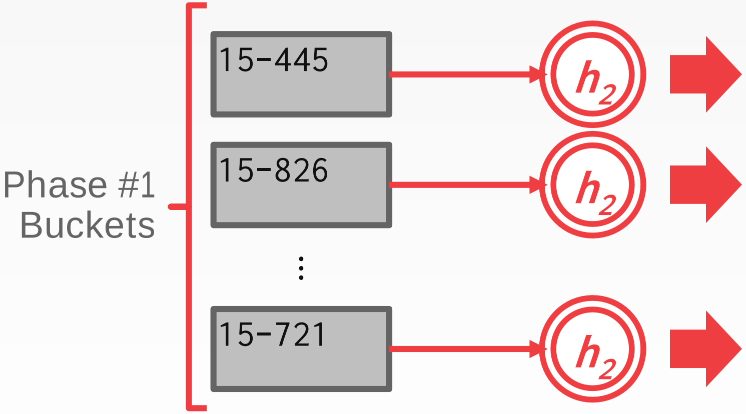


HASHING AGGREGATE PHASE #2: REHASH

```
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



Hash Table

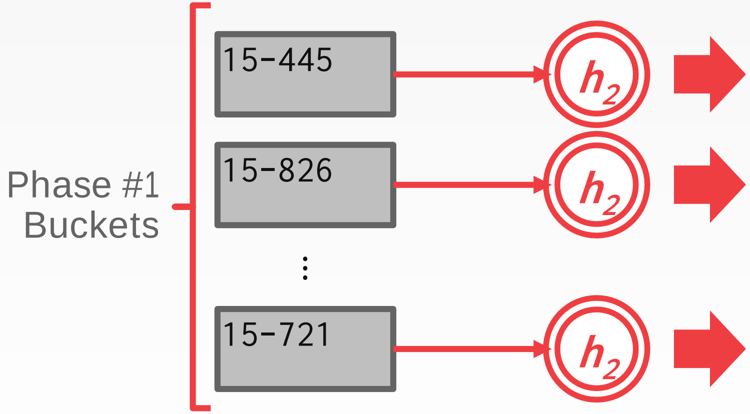
Key	Value
XXX	15-445
YYY	15-826
ZZZ	15-721

HASHING AGGREGATE 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



Hash Table

Key	Value
XXX	15-445
YYY	15-826
ZZZ	15-721

cid
15-445
15-721
15-826

HASHING SUMMARIZATION

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

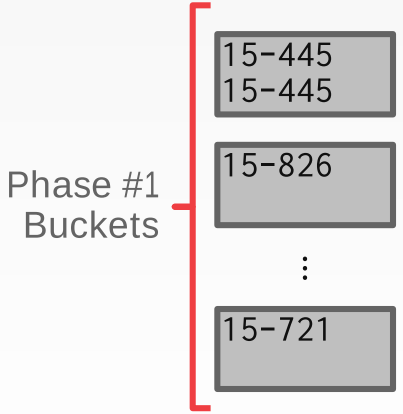
When we want to insert a new tuple into the hash table:

→ If we find a matching **GroupKey**, just update the **RunningVal** appropriately

→ Else insert a new **GroupKey→RunningVal**

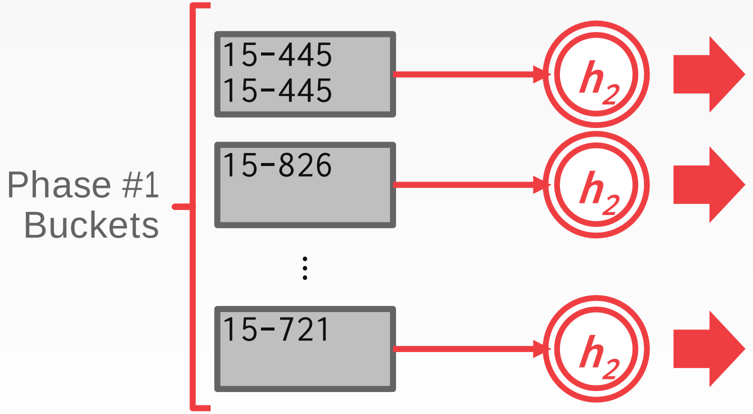
HASHING SUMMARIZATION

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



HASHING SUMMARIZATION

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



Hash Table

key	value
XXX	15-445→(2,7.32)
YYY	15-826→(1,3.33)
ZZZ	15-721→(1,2.89)

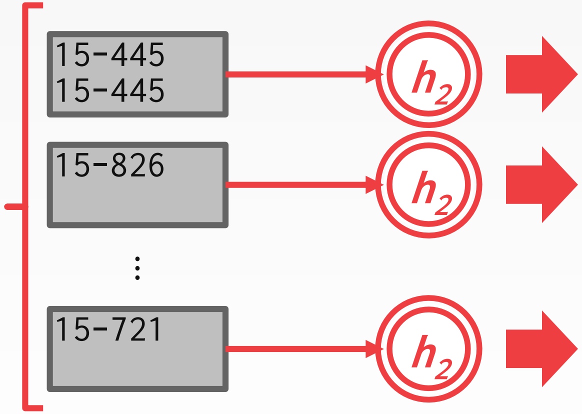
HASHING SUMMARIZATION

```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)

Phase #1 Buckets



Hash Table

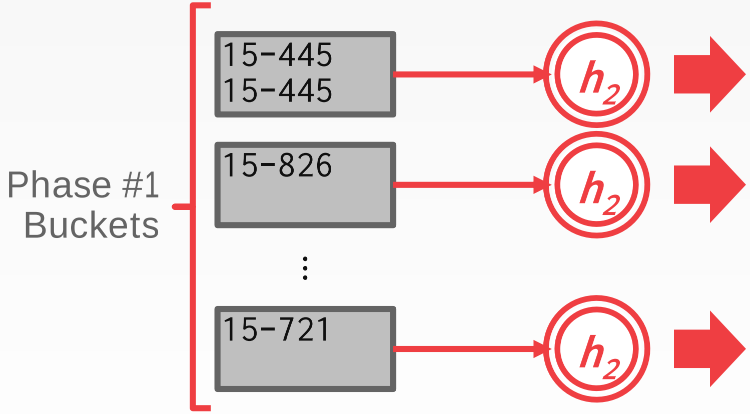
key	value
XXX	15-445→(2,7.32)
YYY	15-826→(1,3.33)
ZZZ	15-721→(1,2.89)

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
XXX	15-445→(2,7.32)
YYY	15-826→(1,3.33)
ZZZ	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 $B \cdot \text{sqrt}(f \cdot N)$

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.

NEXT CLASS

Nested Loop Join

Sort-Merge Join

Hash Join

"Exotic" Joins