Homework #2 is due Sunday, Oct 3rd @ 11:59pm

Project #2 is due Sunday, Oct 17th @ 11:59pm
→ Q&A Session on Thursday, Sept 30th from 5-6pm
→ See the Piazza post for details

Mid-Term Exam is Wednesday, Oct 13th
→ During regular class time from 3:05-4:25pm
→ More details next week…
We want to allow multiple threads to read and update a B+Tree at the same time.

We need to protect against two types of problems:
→ Threads trying to modify the contents of a node at the same time.
→ One thread traversing the tree while another thread splits/merges nodes.
What was the first step that all the update examples did on the B+Tree?

Taking a write latch on the root every time becomes a bottleneck with higher concurrency.
Most modifications to a B+Tree will not require a split or merge.

Instead of assuming that there will be a split/merge, optimistically traverse the tree using read latches.

If you guess wrong, repeat traversal with the pessimistic algorithm.
BETTER LATCHING ALGORITHM

**Search**: Same as before.

**Insert/Delete**:

→ Set latches as if for search, get to leaf, and set $W$ latch on leaf.

→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

This approach optimistically assumes that only leaf node will be modified; if not, $R$ latches set on the first pass to leaf are wasteful.
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

```
R
```

```
A
```

```
20
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

H will not need to coalesce, so we’re safe!
EXAMPLE #2 – DELETE 38

H will not need to coalesce, so we’re safe!
**EXAMPLE #2 – DELETE 38**

H will not need to coalesce, so we’re safe!
EXAMPLE #4 – INSERT 25
We need to split F, so we have to restart and re-execute like before.
The threads in all the examples so far have acquired latches in a “top-down” manner. 
→ A thread can only acquire a latch from a node that is below its current node.
→ If the desired latch is unavailable, the thread must wait until it becomes available.

But what if we want to move from one leaf node to another leaf node?
LEAF NODE SCAN EXAMPLE #1
LEAF NODE SCAN EXAMPLE #1

$T_i$: Find Keys < 4
LEAF NODE SCAN EXAMPLE #1

T₁: Find Keys < 4
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys < 4
LEAF NODE SCAN EXAMPLE #1

T₁: Find Keys < 4

Do not release latch on C until thread has latch on B
T_1: Find Keys < 4

Do not release latch on C until thread has latch on B
LEAF NODE SCAN EXAMPLE #1

$T_1$: Find Keys < 4
LEAF NODE SCAN EXAMPLE #2

\[ T_1 \]: Find Keys < 4
\[ T_2 \]: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

\( T_1 \): Find Keys < 4
\( T_2 \): Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

T₁: Find Keys < 4
T₂: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys $< 4$

$T_2$: Find Keys $> 1$
LEAF NODE SCAN EXAMPLE #2

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
Both $T_1$ and $T_2$ now hold this read latch.

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
LEAF NODE SCAN EXAMPLE #2

Both $T_1$ and $T_2$ now hold this read latch.

$T_1$: Find Keys $< 4$

$T_2$: Find Keys $> 1$
LEAF NODE SCAN EXAMPLE #2

Only $T_1$ holds this read latch.

Only $T_2$ holds this read latch.

$T_1$: Find Keys < 4

$T_2$: Find Keys > 1
Latches do not support deadlock detection or avoidance. The only way we can deal with this problem is through coding discipline.

The leaf node sibling latch acquisition protocol must support a “no-wait” mode.

The DBMS's data structures must cope with failed latch acquisitions.
CONCLUSION

Making a data structure thread-safe is notoriously difficult in practice.

We focused on B+ Trees, but the same high-level techniques are applicable to other data structures.
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
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

COURSE STATUS

- Query Planning
- Operator Execution
- Access Methods
- Buffer Pool Manager
- Disk Manager
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
```
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 are going to rely 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 I/O.
TODAY'S AGENDA

External Merge Sort
Aggregations
WHY DO WE NEED TO SORT?

Relational model/SQL is unsorted.
WHY DO WE NEED TO SORT?

Relational model/SQL is unsorted.

Queries may request that tuples are sorted in a specific way (ORDER BY).
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)
→ ...
If data fits in memory, then we can use a standard sorting algorithm like quicksort.
SORTING ALGORITHMS

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...
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.
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*).
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*).
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*).
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*).
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.
2-WAY EXTERNAL MERGE SORT
Pass #0

→ Read all \( B \) pages of the table into memory
→ Sort pages into runs and write them back to disk
2-WAY EXTERNAL MERGE SORT

Pass #0

→ Read all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk
Pass #0

→ Read all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk
Pass #0

→ Read all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk
Pass #0

→ Read all \( B \) pages of the table into memory
→ Sort pages into runs and write them back to disk
Pass #0

→ Read all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk
Pass #0
→ Read all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk

Pass #1,2,3,...
→ Recursively merge pairs of runs into runs twice as long
2-WAY EXTERNAL MERGE SORT

Pass #0
→ Read all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk

Pass #1,2,3,…
→ Recursively merge 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 all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk

**Pass #1,2,3,...**
→ Recursively merge 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 all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk

Pass #1, 2, 3, ...
→ Recursively merge 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 all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk

**Pass #1,2,3,…**
→ Recursively merge 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 all $B$ pages of the table into memory
→ Sort pages into runs and write them back to disk

Pass #1,2,3,…
→ Recursively merge pairs of runs into runs twice as long
→ Uses three buffer pages (2 for input pages, 1 for output)
In each pass, we read and write every page in the file.

Number of passes
= 1 + ⌈log₂N⌉
Total I/O cost
= 2N · (# of passes)
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

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

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

\[= 2N \cdot (\# \text{ of passes})\]
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
= $2N \cdot (# \text{ of } \text{passes})$
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...
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.
DOUBBLE 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.
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.
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.
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 = $1 + \lceil \log_{B-1} \left\lceil N / B \right\rceil \rceil$
Total I/O Cost = $2N \cdot (\# \text{ of passes}$)
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 $= 2N \cdot (\# \text{ of passes})$
GENERAL EXTERNAL MERGE SORT

Pass #0
→ Use $B$ buffer pages
→ Produce $\lceil \frac{N}{B} \rceil$ sorted runs of size $B$

Pass #1,2,3,...
→ Merge $B-1$ runs (i.e., K-way merge)

Number of passes $= 1 + \lceil \log_{B-1} \left( \frac{N}{B} \right) \rceil$

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

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: \( N=108, B=5 \)
EXAMPLE

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

\[ \text{Pass #0:} \frac{N}{B} = \left\lfloor \frac{108}{5} \right\rfloor = 22 \text{ sorted runs of 5 pages each} \]
(last run is only 3 pages).
Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: \( N=108, B=5 \)

→ **Pass #0:** \([N / B] = [108 / 5] = 22\) sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** \([N' / B-1] = [22 / 4] = 6\) sorted runs of 20 pages each (last run is only 8 pages).
Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: \( N=108, \ B=5 \)

→ **Pass #0:** \( \lceil N \div B \rceil = \lceil 108 \div 5 \rceil = 22 \) sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** \( \lceil N' \div B-1 \rceil = \lceil 22 \div 4 \rceil = 6 \) sorted runs of 20 pages each (last run is only 8 pages).

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

→ **Pass #1:** $[N' / B-1] = [22 / 4] = 6$ sorted runs of 20 pages each (last run is only 8 pages).

→ **Pass #2:** $[N'' / B-1] = [6 / 4] = 2$ 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: \( N=108, \ B=5 \)

\[
\Rightarrow \text{Pass #0: } \lceil \frac{N}{B} \rceil = \lceil \frac{108}{5} \rceil = 22 \text{ sorted runs of 5 pages each (last run is only 3 pages).}
\]

\[
\Rightarrow \text{Pass #1: } \lceil \frac{N'}{B-1} \rceil = \lceil \frac{22}{4} \rceil = 6 \text{ sorted runs of 20 pages each (last run is only 8 pages).}
\]

\[
\Rightarrow \text{Pass #2: } \lceil \frac{N''}{B-1} \rceil = \lceil \frac{6}{4} \rceil = 2 \text{ sorted runs, first one has 80 pages and second one has 28 pages.}
\]

\[
\Rightarrow \text{Pass #3: Sorted file of 108 pages.}
\]

\[
1+\lceil \log_{B-1}\left[ \frac{N}{B} \right] \rceil = 1+\lceil \log_4 22 \rceil = 1+\lceil 2.229... \rceil = 4 \text{ passes}
\]
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
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.
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.
Collapse values for a single attribute from multiple tuples into a single scalar value.

Two implementation choices:
→ Sorting
→ Hashing
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
ORDER BY cid

enrolled(sid, cid, grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
SORTING AGGREGATION

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

Filter

enrolled(sid, cid, grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
### SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
ORDER BY cid

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
</tr>
<tr>
<td>15-826</td>
</tr>
<tr>
<td>15-721</td>
</tr>
<tr>
<td>15-445</td>
</tr>
</tbody>
</table>

**Filter**

**Remove Columns**
### SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B','C')
ORDER BY cid

#### Filter

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

#### Remove Columns

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

#### Sort

<table>
<thead>
<tr>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
</tr>
<tr>
<td>15-826</td>
</tr>
<tr>
<td>15-721</td>
</tr>
<tr>
<td>15-445</td>
</tr>
<tr>
<td>15-445</td>
</tr>
<tr>
<td>15-826</td>
</tr>
</tbody>
</table>

#### Example Table

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
SORTING AGGREGATION

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

**Filter**

**Remove Columns**

**Sort**

**Eliminate Dupes**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

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

enrolled(sid, cid, grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

Filter

Remove Columns

Sort

Eliminate Dupes
What if we do not need the data to be ordered?
→ Forming groups in **GROUP BY** (no ordering)
→ Removing duplicates in **DISTINCT** (no ordering)
What if we do **not** need the data to be ordered?
→ Forming groups in **GROUP BY** (no ordering)
→ Removing duplicates in **DISTINCT** (no ordering)

Hashing is a better alternative in this scenario.
→ Only need to remove duplicates, no need for ordering.
→ Can be computationally cheaper than sorting.
HASHING AGGREGATE

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

→ **DISTINCT**: Discard duplicate

→ **GROUP BY**: Perform aggregate computation

If everything fits in memory, then this is easy.

If the DBMS must spill data to disk, then we need to be smarter…
EXTERNAL HASHING AGGREGATE

**Phase #1 – Partition**
- Divide tuples into buckets based on hash key
- Write them out to disk when they get full

**Phase #2 – ReHash**
- Build in-memory hash table for each partition and compute the aggregation
PHASE #1 – PARTITION

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

$\rightarrow$ A partition is one or more pages that contain the set of keys with the same hash value.

$\rightarrow$ 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.
SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')

enrolled(sid, cid, grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
PHASE #1 – PARTITION

\[
\text{SELECT DISTINCT cid}
\]
\[
\text{FROM enrolled}
\]
\[
\text{WHERE grade IN ('B', 'C')}
\]

```
<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
```

enrolled(sid, cid, grade)

```
<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
```
PHASE #1 – PARTITION

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

enrolled(sid,cid,grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

Filter

Remove Columns
**PHASE #1 – PARTITION**

Select distinct cid from enrolled where grade in ('B','C')

**Filter**

**Remove Columns**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**B-1 partitions**

$h_1$
PHASE #1 – PARTITION

SELECT DISTINCT cid
FROM enrolled
WHERE grade IN ('B', 'C')
**PHASE #1 – PARTITION**

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**Filter**

**Remove Columns**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>5366</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>5366</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>5366</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>5366</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**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')

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

enrolled(sid,cid,grade)
PHASE #2 – REHASH

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

Phase #1 Buckets

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15-826</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15-826</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

enrolled(sid,cid,grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
**PHASE #2 – REHASH**

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

---

**Phase #1 Buckets**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>15-445</td>
<td></td>
</tr>
</tbody>
</table>

**B-1 Partitions**

<table>
<thead>
<tr>
<th>15-826</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-826</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**enrolled(sid, cid, grade)**
PHASE #2 – REHASH

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

Phase #1 Buckets

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

enrolled(sid,cid,grade)
PHASE #2 – REHASH

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

**Phase #1 Buckets**

<table>
<thead>
<tr>
<th>15-445</th>
<th>15-445</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>15-445</td>
</tr>
</tbody>
</table>

**Hash Table**

```
<table>
<thead>
<tr>
<th>15-445</th>
</tr>
</thead>
</table>
```

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>
**Phase #1 Buckets**

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

**Hash Table**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**Phase #2 – Rehash**
**Phase #2 – Rehash**

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**Phase #1 Buckets**

- 15-826 15-826

**Hash Table**

- 15-445
- 15-826

**Final Result**

- 15-445
- 15-826
**PHASE #2 – REHASH**

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

**Phase #1 Buckets**

<table>
<thead>
<tr>
<th>cid</th>
<th>15-445</th>
<th>15-445</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15-445</td>
<td>15-445</td>
</tr>
<tr>
<td></td>
<td>15-826</td>
<td>15-826</td>
</tr>
<tr>
<td></td>
<td>15-826</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>15-445</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15-721</td>
</tr>
</tbody>
</table>

**Hash Table**

<table>
<thead>
<tr>
<th>cid</th>
<th>15-445</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15-826</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

**Final Result**

<table>
<thead>
<tr>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
</tr>
<tr>
<td>15-826</td>
</tr>
</tbody>
</table>
Phase #2 – Rehash

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

Phase #1 Buckets

<table>
<thead>
<tr>
<th>cid</th>
<th>15-445</th>
<th>15-445</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15-445</td>
<td>15-445</td>
</tr>
<tr>
<td></td>
<td>15-445</td>
<td>15-445</td>
</tr>
<tr>
<td></td>
<td>15-826</td>
<td>15-826</td>
</tr>
<tr>
<td></td>
<td>15-721</td>
<td></td>
</tr>
</tbody>
</table>

enrolled(sid, cid, grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

Final Result

<table>
<thead>
<tr>
<th>cid</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td></td>
</tr>
<tr>
<td>15-826</td>
<td></td>
</tr>
</tbody>
</table>

Hash Table
Phase #1 Buckets

Phase #2 – Rehash

enrolled(sid,cid,grade)

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>C</td>
</tr>
</tbody>
</table>

Hash Table

Final Result

select distinct cid
from enrolled
where grade in ('B','C')
During the ReHash phase, store pairs of the form \((\text{GroupKey} \rightarrow \text{RunningVal})\)

When we want to insert a new tuple into the hash table:
→ If we find a matching \text{GroupKey}, just update the \text{RunningVal} appropriately
→ Else insert a new \((\text{GroupKey} \rightarrow \text{RunningVal})\)
SELECT cid, AVG(s.gpa)
FROM student AS s, enrolled AS e
WHERE s.sid = e.sid
GROUP BY cid

Phase #1 Buckets

15-445
15-445
15-826

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

**Phase #1 Buckets**

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

**Hash Table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>(2, 7.32)</td>
</tr>
<tr>
<td>15-826</td>
<td>(1, 3.33)</td>
</tr>
<tr>
<td>15-721</td>
<td>(1, 2.89)</td>
</tr>
</tbody>
</table>

**Running Totals**

- $\text{AVG(col)} \rightarrow (\text{COUNT}, \text{SUM})$
- $\text{MIN(col)} \rightarrow (\text{MIN})$
- $\text{MAX(col)} \rightarrow (\text{MAX})$
- $\text{SUM(col)} \rightarrow (\text{SUM})$
- $\text{COUNT(col)} \rightarrow (\text{COUNT})$
**HASHING SUMMARIZATION**

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

**Phase #1**

**Buckets**
- 15–445
- 15–445
- 15–826
- ...
- 15–721

**Hash Table**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–445</td>
<td>(2, 7.32)</td>
</tr>
<tr>
<td>15–826</td>
<td>(1, 3.33)</td>
</tr>
<tr>
<td>15–721</td>
<td>(1, 2.89)</td>
</tr>
</tbody>
</table>

**Running Totals**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG(col)</td>
<td>(COUNT, SUM)</td>
</tr>
<tr>
<td>MIN(col)</td>
<td>(MIN)</td>
</tr>
<tr>
<td>MAX(col)</td>
<td>(MAX)</td>
</tr>
<tr>
<td>SUM(col)</td>
<td>(SUM)</td>
</tr>
<tr>
<td>COUNT(col)</td>
<td>(COUNT)</td>
</tr>
</tbody>
</table>

**Final Result**

<table>
<thead>
<tr>
<th>cid</th>
<th>AVG(gpa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–445</td>
<td>3.66</td>
</tr>
<tr>
<td>15–826</td>
<td>3.33</td>
</tr>
<tr>
<td>15–721</td>
<td>2.89</td>
</tr>
</tbody>
</table>
CONCLUSION

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

We already discussed the optimizations for sorting:
→ Chunk I/O into large blocks to amortize costs
→ Double-buffering to overlap CPU and I/O
NEXT CLASS

Nested Loop Join
Sort-Merge Join
Hash Join