# Lecture 18: Sorting + Aggregation

# Administrivia

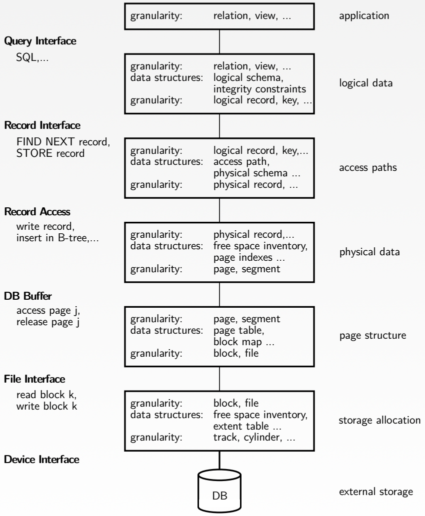
- Assignment 4 and Sheet 4 released
- Guest lecture on Nov 17
- Extra credit exam on Nov 22

# Today's Agenda

Sorting + Aggregation
1.1 Recap
1.2 External Merge Sort
1.3 Tree-based Sorting
1.4 Aggregation
1.5 Conclusion

# Recap

# A More Detailed Architecture



# Anatomy of a Database System [Monologue]

- Process Manager
  - Connection Manager + Admission Control
- Query Processor
  - Query Parser
  - Query Optimizer (*a.k.a.*, Query Planner)
  - Query Executor
- Transactional Storage Manager
  - Lock Manager
  - Access Methods (*a.k.a.*, Indexes)
  - Buffer Pool Manager
  - Log Manager
- Shared Utilities
  - Memory, Disk, and Networking Manager

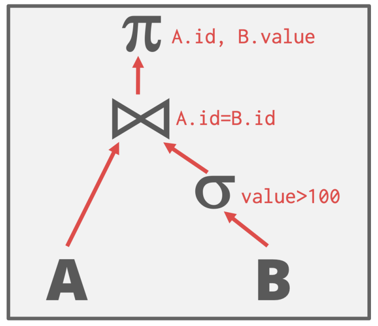
# Query Execution

- We are now going to talk about how to execute queries using table heaps and indexes.
- Coming weeks:
  - Operator Algorithms
  - Query Processing Models
  - Runtime Architectures

# Query Plan

- The operators are arranged in a tree.
- Data flows from the leaves of the tree up towards the root.
- 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

- We **cannot** assume that the results of a query fits in memory.
- We are going use the **buffer pool** to implement query execution 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
- Tree-based Sorting
- Aggregation

# External Merge Sort

# 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.
  - Aggregation (`GROUP BY`).

# Sorting Algorithms

- If data fits in memory, then we can use a standard in-memory 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.

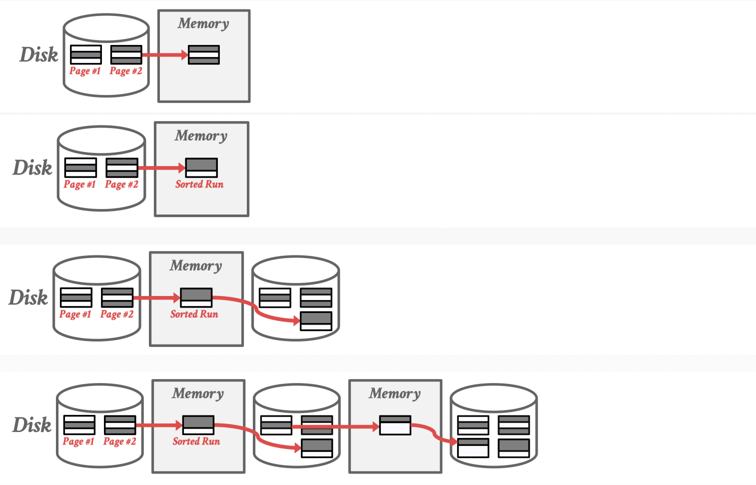
# 2-Way External Merge Sort

- 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.

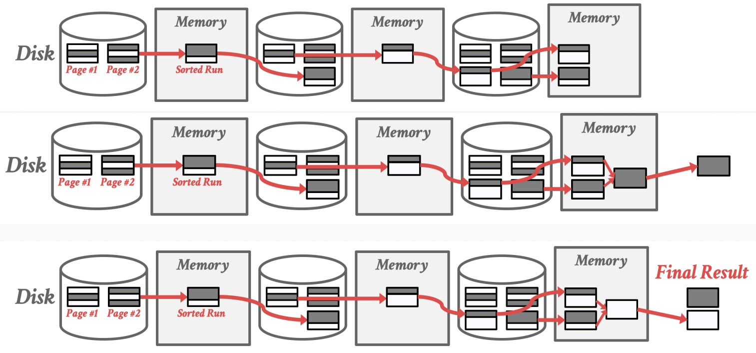
# 2-Way External Merge Sort

- **Pass 0**
  - Read every **B** pages of the table into memory
  - Sort pages into runs and write them back to disk.
- **Passes 1,2,3,. . .**
  - Recursively merge pairs of runs into runs **twice** as long.
  - Use three buffer pages (2 for input pages, 1 for output).

# 2-Way External Merge Sort



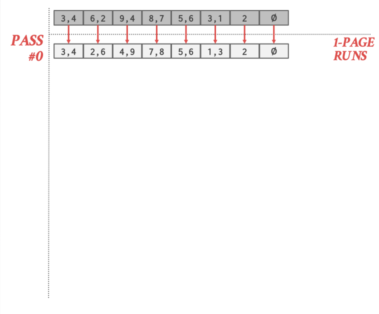
# 2-Way External Merge Sort

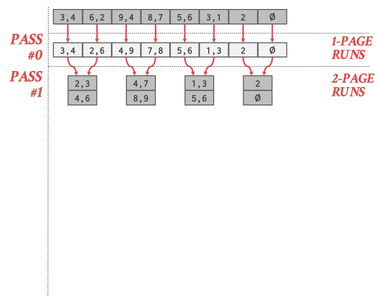


# 2-Way External Merge Sort

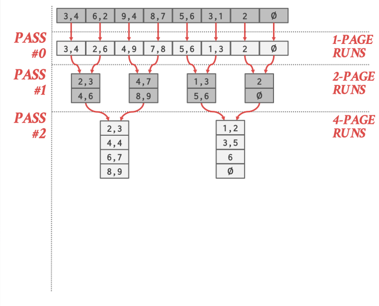
- In each pass, we read and write each page in file.
- Number of passes = $1 + \lceil \log_2 N \rceil$
- Total I/O cost = 2N x (Number of passes)

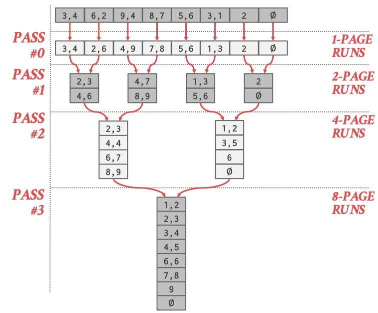
# 2-Way External Merge Sort



# 2-Way External Merge Sort



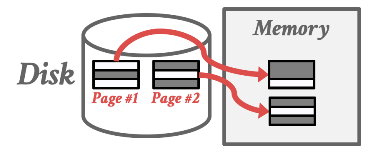

Georgia Tech

# 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.



# General External Merge Sort

- **Pass 0**
  - Use **B** buffer pages.
  - Produce **N / B** sorted runs of size **B**
- **Pass 1,2,3,. . .**
  - Merge **B-1** runs (*i.e.*, K-way merge).
- Number of passes = 1 + $\lceil log_{B-1} N/B \rceil$
- Total I/O Cost = 2N x (Number of passes)

# K-Way Merge Algorithm

- Input: **K** sorted sub-arrays
- Output: 1 sorted array
  - Efficiently compute the minimum element of all **K** sub-arrays.
  - Repeatedly transfer that element to output array
- Internally maintain a heap to efficiently compute minimum element.

# Example

- Sort 108 pages with 5 buffer 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.
- $1 + log_{B-1} N/B = 1 + \lceil log_4 22 \rceil = 1 + \lceil 2.229 \rceil = 4$ passes

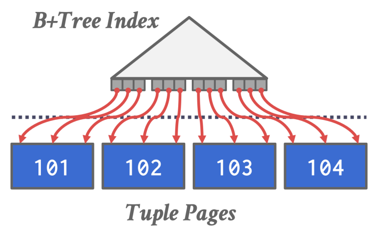
# Tree-based Sorting

# 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

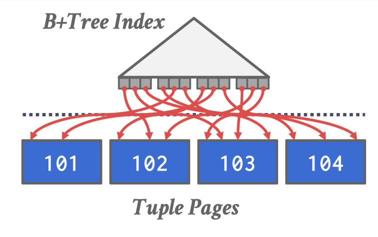
# Case 1 – Clustered B+Tree

- 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.



# Aggregation

# Aggregation

- 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 |

Georgia Tech

# 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 **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.
  - May 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:
  - `GROUP BY`: Perform aggregate computation.
  - `DISTINCT`: Discard duplicates.
- 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 |

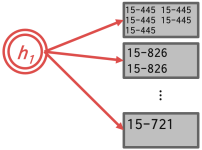
***Filter***

| 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***

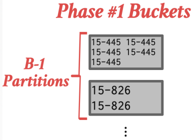


# Phase 2 – ReHash

- 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')
```




enrolled(sid,cid,grade)

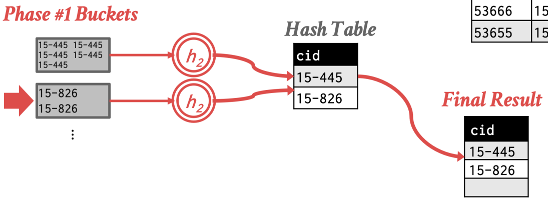
| sid | cid | grade |
|---|---|---|
| 53666 | 15-445 | C |
| 53688 | 15-721 | A |
| 53688 | 15-826 | B |
| 53666 | 15-721 | C |
| 53655 | 15-445 | C |

# 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 |

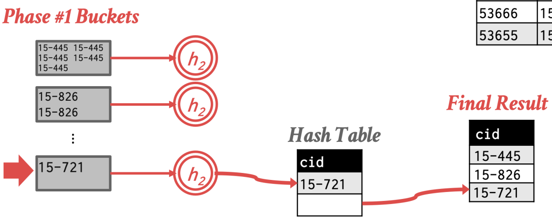


# Phase 2 – ReHash

```
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 |



# Hashing Summarization

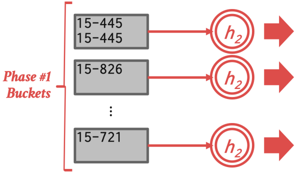
- During the `ReHash` phase, store pairs of the form (`GroupKey` $\longrightarrow$ `RunningVal`)
- When we want to insert a new tuple into the hash table:
  - If we find a matching `GroupKey`, just update the `RunningVal` appropriately
  - Else insert a new `GroupKey` $\longrightarrow$ `RunningVal`

# 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)




***Hash Table***

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

***Final Result***

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

# Conclusion

# Conclusion

- 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