THE UNIVERSITY
of EDINBURGH

Advanced Database Systems

Spring 2024

Lecture #17:

External Sorting & Aggregation

R&G: Chapters 13 & 14

Query Execution Overview

SQL Query

```
SELECT S.name
  FROM Student S, Enrolled E
 WHERE S.sid = E.sid
   AND E.cid = 'INF-11199'
```

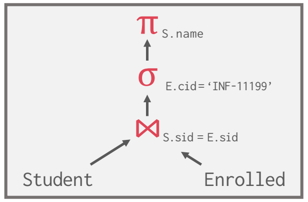
Query Parser & Optimiser

Relational Algebra

$\pi_{S.name}(\sigma_{E.cid='INF-11199'}(\text{Student} \bowtie_{S.sid=E.sid} \text{Enrolled}))$

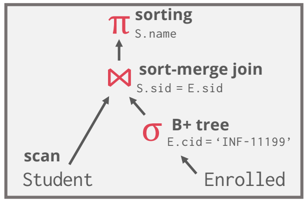
Equivalent to...

Logical Query Plan



But actually will produce plan with operator code

Optimised Physical Query Plan



Query Plans and Operators

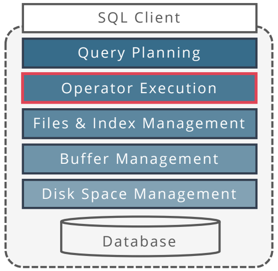
Query plan = Network of operators able to evaluate a query

One operator may have different implementations

- All semantically equivalent
- But with different performance characteristics

Focus of this lecture

- Implementation of **sort**
- Implementation of (grouped by) **aggregation**



Why Do We Need Sorting?

Explicit sorting via the SQL ORDER BY clause

```
SELECT A, B, C FROM R ORDER BY A;
```

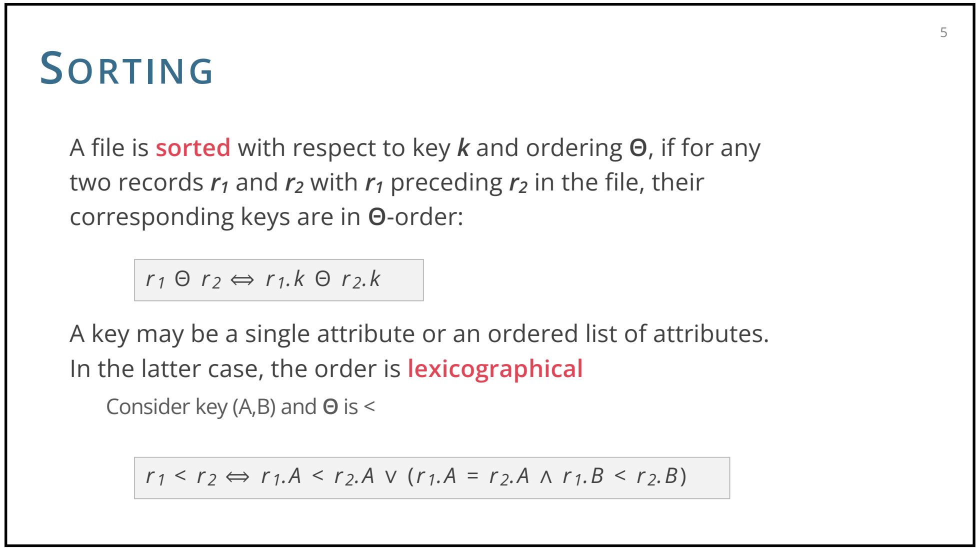
Implicit sorting, e.g., for duplicate elimination

```
SELECT DISTINCT A, B, C FROM R;
```

Implicit sorting, e.g., to prepare (sort-merge) equi-join

```
SELECT R.A, S,C FROM R JOIN S ON R.B = S.B;
```

Grouping via **group by**, first step in **bulk loading** tree indexes, **sorted** rid scans after access to unclustered indexes, etc.



Sorting

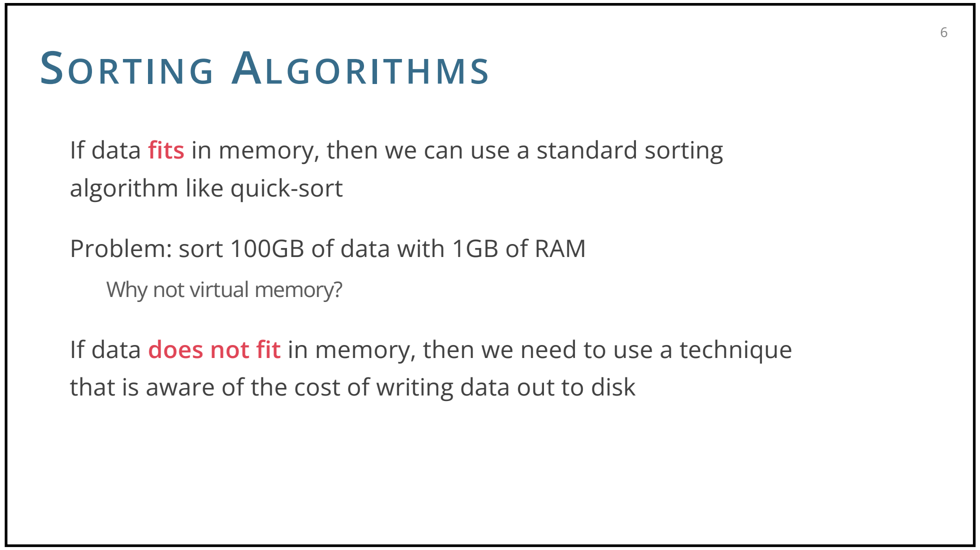
A file is **sorted** with respect to key ***k*** and ordering **Θ**, if for any two records r_1 and r_2 with r_1 preceding r_2 in the file, their corresponding keys are in **Θ**-order:

$$r_1 \; \Theta \; r_2 \Leftrightarrow r_1.k \; \Theta \; r_2.k$$

A key may be a single attribute or an ordered list of attributes. In the latter case, the order is **lexicographical**

Consider key (A,B) and **Θ** is <

$$r_1 < r_2 \Leftrightarrow r_1.A < r_2.A \lor (r_1.A = r_2.A \land r_1.B < r_2.B)$$



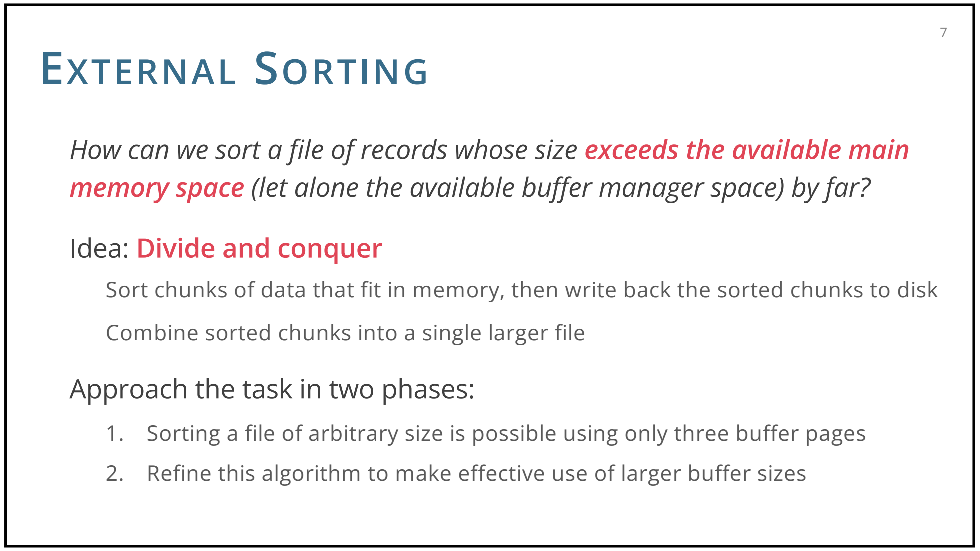
Sorting Algorithms

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

Problem: sort 100GB of data with 1GB of RAM

Why not virtual memory?

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 Sorting

*How can we sort a file of records whose size **exceeds the available main memory space** (let alone the available buffer manager space) by far?*

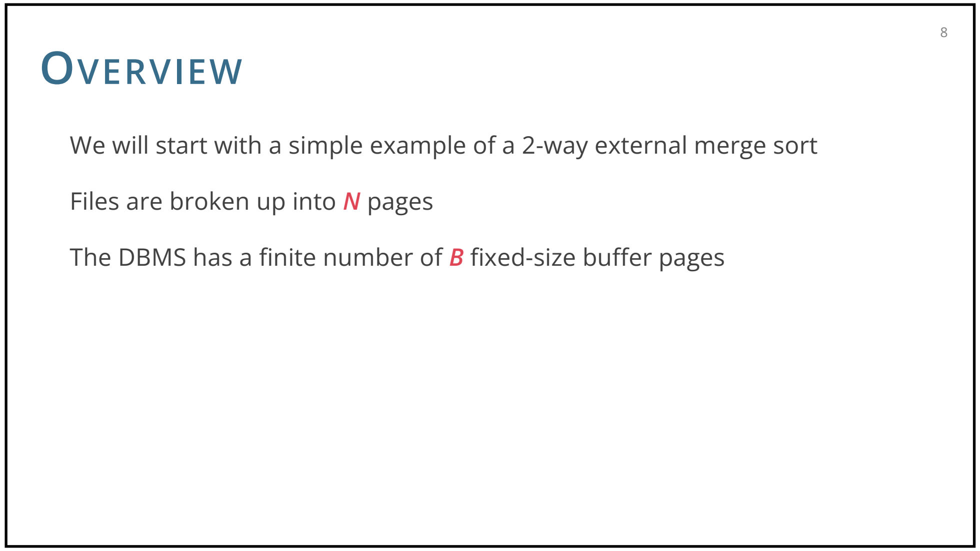
Idea: **Divide and conquer**

Sort chunks of data that fit in memory, then write back the sorted chunks to disk

Combine sorted chunks into a single larger file

Approach the task in two phases:

1. Sorting a file of arbitrary size is possible using only three buffer pages
2. Refine this algorithm to make effective use of larger buffer sizes



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 buffer pages

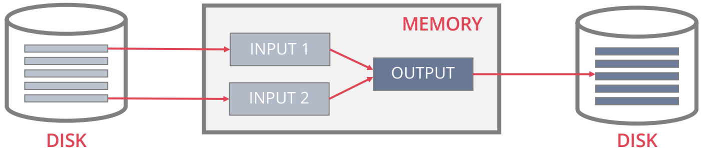
2-Way External Merge Sort

Pass #0

- Read a page into memory, sort it, and write it back to disk (***uses 1 buffer page***)
- Each sorted set of pages is called a **run**

Pass #1, #2, #3, …

- Recursively merge pairs of runs into runs twice as long (***uses 3 buffer pages***)
- When input is consumed read next page from disk. When output is full, write to disk

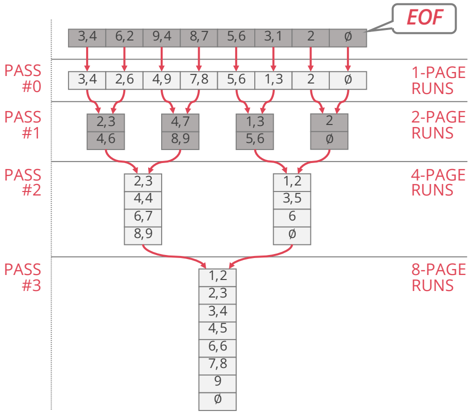


2-Way External Merge Sort

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

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

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

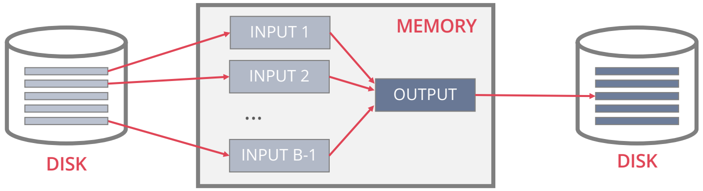


External Merge Sort

Previous algorithm uses only three buffer pages (*B* = *3*)

How can we make effective use of a larger buffer pool (*B* > *3*)?

- Reduce # of initial runs by using the full buffer space during in-memory sort
- Reduce # of passes by merging *B-1* runs at a time



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., multi-way merge)

Number of passes $= 1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

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

Example

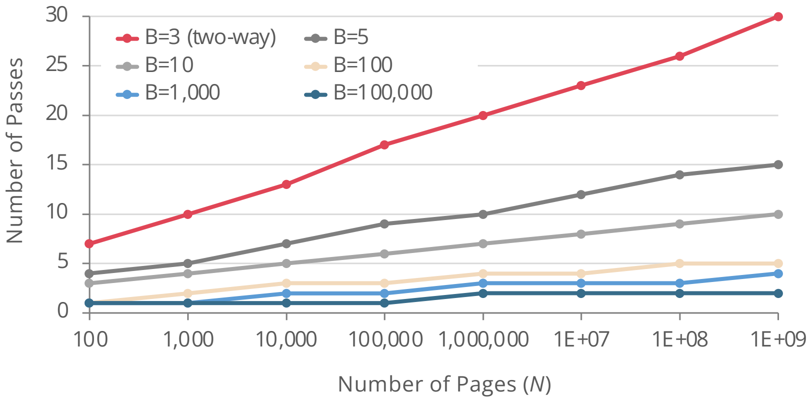
Sort N = 108 page file with B = 5 buffer pages

- **Pass #0**: $\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
- **Pass #1**: $\lceil 22/4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
- **Pass #2**: $\lceil 6/4 \rceil$ = 2 sorted runs of 80 pages and 28 pages
- **Pass #3**: Sorted file of 108 pages

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

Total I/O cost = $2N \cdot$(# of passes) = $2 \cdot 108 \cdot 4$ = 864

Number of Passes of External Sort



Using B+ Trees for Sorting

If the table to be sorted has a B+ tree index on the sort attribute(s), we may be better off by accessing the index and avoid external sorting

Retrieve sorted records by simply traversing the leaf pages of the tree

Cases to consider

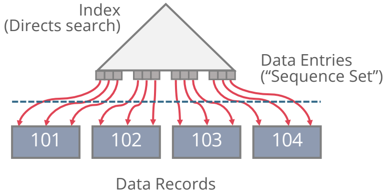
- Clustered B+ tree
- Unclustered B+ tree

Case 1: Clustered B+ Tree

Traverse to the left-most leaf page, then retrieve all leaf pages (variant A)

If variant B is used?
Additional cost of retrieving data records: each page fetched just once

Always better than external sorting!



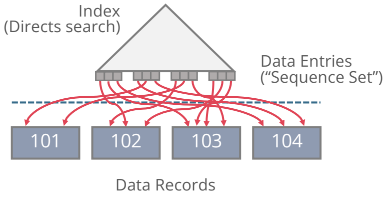
Case 2: Unclustered B+ Tree

Variant *B* for index entries
(each contains *rid* of a data record)

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

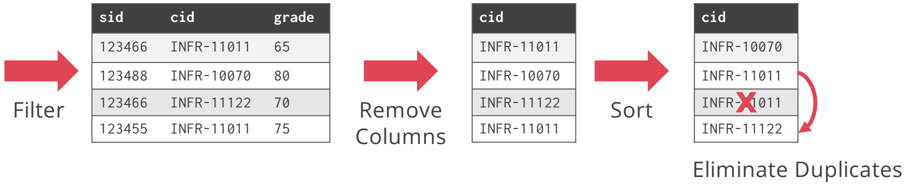


Duplicate Elimination using Sorting

```
SELECT DISTINCT cid
  FROM Enrolled
 WHERE grade < 90
```

Enrolled(sid, cid, grade)

sid	cid	grade
123466	INFR-11011	65
123488	INFR-11122	95
123488	INFR-10070	80
123466	INFR-11122	70
123455	INFR-11011	75



sid	cid	grade
123466	INFR-11011	65
123488	INFR-10070	80
123466	INFR-11122	70
123455	INFR-11011	75

cid
INFR-11011
INFR-10070
INFR-11122
INFR-11011

cid
INFR-10070
INFR-11011
INFR-11011
INFR-11122

Alternative 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

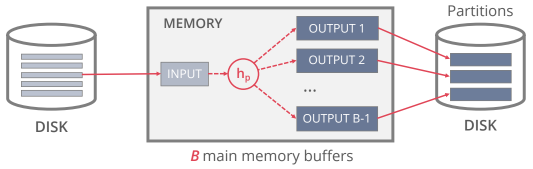
- Only need to remove duplicates, no need for ordering
- Can be computationally cheaper than sorting

External Hashing

We cannot build an in-memory hash table if there is too much data!

Start by splitting up data into smaller pieces!

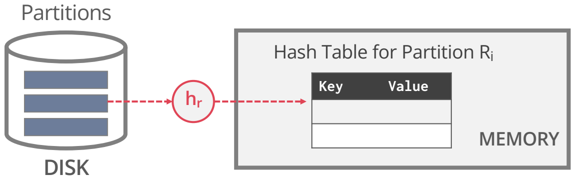
- Use a hash function h_p to partition the data
  - Stream partitions to disk
- If we have *B* pages of buffer, we can split the data into *B-1* partitions
  - 1 buffer page reserved for streaming data in



External Hashing

If partitions are small enough to fit in memory, we can load them in and make an in-memory hash table for each one, one at a time

- Then we can apply duplicate removal, aggregation, etc. in memory
- Every tuple in a partition has the same value when h_p is applied!
- In-memory hash table must use a different hash function h_r that is independent of h_p



Aggregations

Collapse multiple tuples into a single scalar value (SUM, MIN, MAX, …)

Hashing aggregates:
Populate an ephemeral hash table as the DBMS scans the relation. 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…

```
SELECT A, MAX(B) FROM R
GROUP BY A;
```

Hashing Aggregate

Partition phase

- Divide tuples into partitions based on hash key

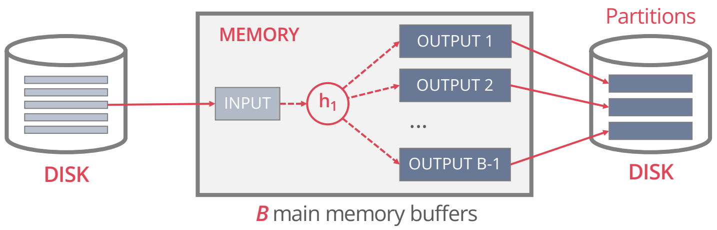
Rehash phase

- Build in-memory hash table for each partition and compute the aggregate

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

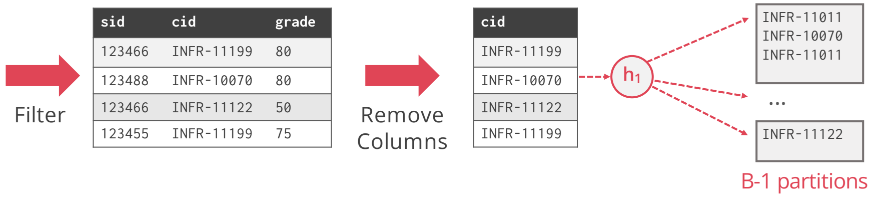


Hashing Aggregate Phase #1: Partition

```
SELECT DISTINCT cid
  FROM Enrolled
 WHERE grade < 90
```

Enrolled(sid, cid, grade)

sid	cid	grade
123466	INFR-11199	80
123488	INFR-11122	95
123488	INFR-10070	80
123466	INFR-11122	50
123455	INFR-11199	75



Filter

sid	cid	grade
123466	INFR-11199	80
123488	INFR-10070	80
123466	INFR-11122	50
123455	INFR-11199	75

Remove Columns

cid
INFR-11199
INFR-10070
INFR-11122
INFR-11199

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 ($\neq h_1$)
- Then go through each bucket of this hash table to bring together matching tuples

No need to load the entire partition at once in memory

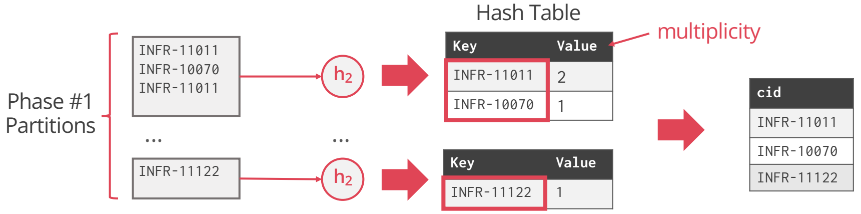
- Can load several pages at a time
- But the hash table built for each partition must fit in memory
- If not enough memory, repeat Phase #1 on each partition with a different hash function

Hashing Aggregate Phase #2: Rehash

```
SELECT DISTINCT cid
  FROM Enrolled
 WHERE grade < 90
```

Enrolled(sid, cid, grade)

sid	cid	grade
123466	INFR-11011	80
123488	INFR-11122	95
123488	INFR-10070	80
123466	INFR-11122	50
123455	INFR-11011	75



Hash Table

Key	Value
INFR-11011	2
INFR-10070	1

multiplicity

Key	Value
INFR-11122	1

cid
INFR-11011
INFR-10070
INFR-11122

Hashing Summarisation

During the Rehash phase, store pairs of the form **GroupKey** ⟶ **RunningValue**

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

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

Hashing Summarisation

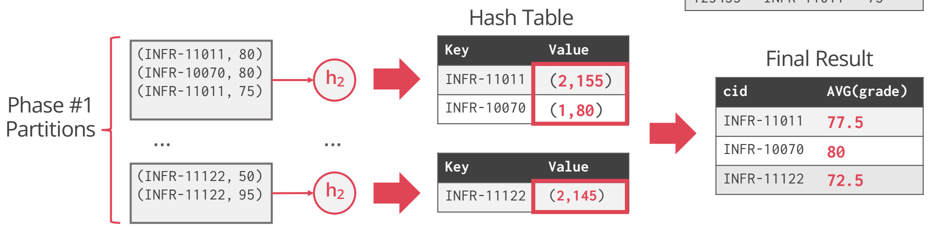
```
SELECT cid, AVG(grade)
  FROM Enrolled
 GROUP BY cid
```

Running Totals

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

Enrolled(sid, cid, grade)

sid	cid	grade
123466	INFR-11011	80
123488	INFR-11122	95
123488	INFR-10070	80
123466	INFR-11122	50
123455	INFR-11011	75



Hash Table

Key	Value
INFR-11011	(2,155)
INFR-10070	(1,80)

Key	Value
INFR-11122	(2,145)

Final Result

cid	AVG(grade)
INFR-11011	77.5
INFR-10070	80
INFR-11122	72.5

Cost Analysis

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

- *B-1* “spill partitions” in Phase #1
- Each partition (i.e., its hash table) should be no more than *B* pages big

Answer: *B · (B-1)*

- A table of *N* pages needs about sqrt(*N*) buffer pages
- Note: assumes hash distributes records evenly!
- Use a “fudge factor” *f* > 1 to capture the (small) increase in size between the partition and a hash table for that partition
- Must be *B* > *f* · *N* / (*B*-1); thus, we need approx. *B* > sqrt(*f* · *N*) buffer pages

Conclusion: Sorting vs. Hashing

External merge sort often finishes in 1-2 passes

- Great if we need output to be sorted anyway
- Not sensitive to duplicates or “bad” hash functions

Duplicate elimination

- Hashing preferred as it scales with # of distinct values
  - Delete duplicates in first pass while partitioning
  - Vs. sort which scales with # of values

Group-by aggregation

- Typically computed via hashing