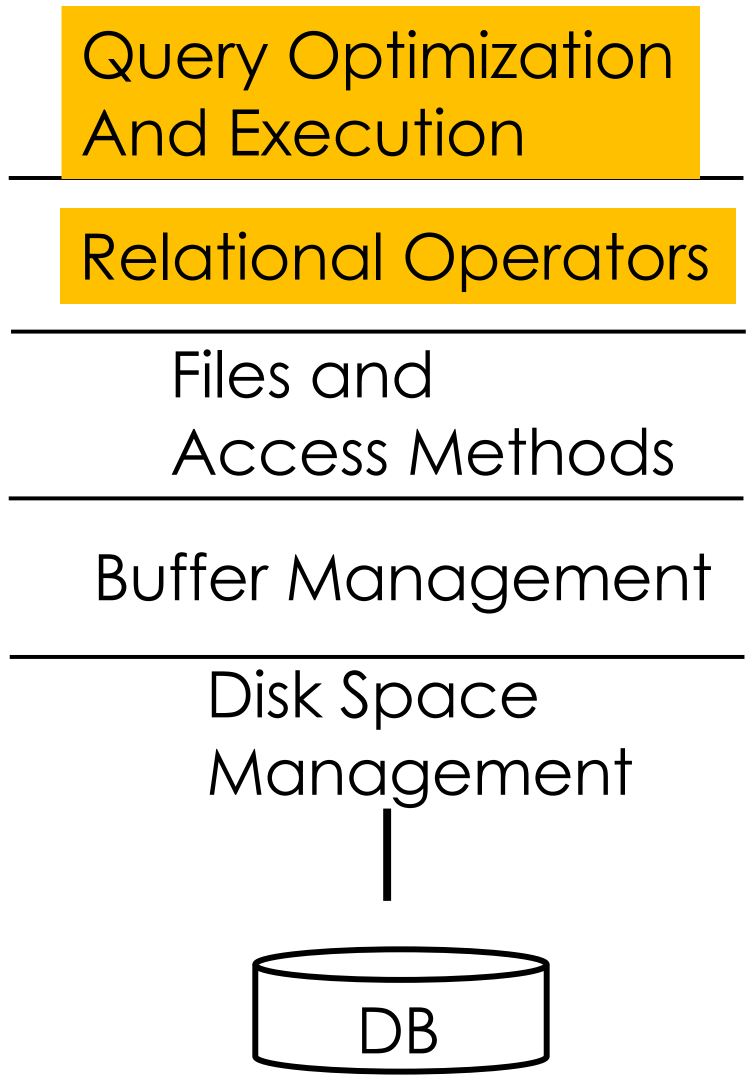
Query Evaluation

Internals of a DBS I



Query Evaluation

- Processing of SQL queries
  - *Parser* translates into internal query representation
    - The parser parses the string and checks for syntax errors
    - Checks existence of relations and attributes
    - Replaces views by their definitions
  - The *query optimizer* translates the query into an efficient execution plan
    - Many different execution plans exist;
    - Choosing an efficient execution plan is crucial
  - The *plan executor* executes the execution plan and delivers data

Query Evaluation

- Query decomposition
  - Queries are composed of few basic operators
    - Selection
    - Projection
    - Order by
    - Join
    - Group by
    - Intersection
    - …
  - Several alternative algorithms for implementing each relational operator
  - Not a single "best" algorithm; efficiency of each implementation depends on factors like size of the relation, existing indexes, size of buffer pool etc.

Basic terminology

- Access Path
  - The method used to retrieve a set of tuples from a relation
    - Basic: file scan
    - indexing → index plus matching selection condition (index can be used to retrieve only the tuples that satisfy condition)
    - Partitioning and others

Cost model

- In order to compare different alternatives we must estimate how expensive a specific execution is
- Input for cost model:
  - the query,
  - database statistics (e.g., distribution of values, etc.),
  - resource availability and costs: buffer size, I/O delay, disk bandwidth, CPU costs, network bandwidth, ….
- Our cost model
  - Number of I/O = pages retrieved from disk
    - assumption that the root and all intermediate nodes of the B+ are in main memory:
    - leaf pages and data pages may not be in main memory!!!
    - A page P might be retrieved several times if many pages are accessed between two accesses of P - might still be in memory
      might have already been evicted

Basic Query Processing Operations

- Selection: σ
  - Scan (without an index)
  - Use an index involving the attributes of the selection
- Projection: π
- Sorting
- Joining
  - Nested loop join
  - Sort-merge join
  - Many others…

big thing!

- Grouping and duplicate elimination

related to sorting

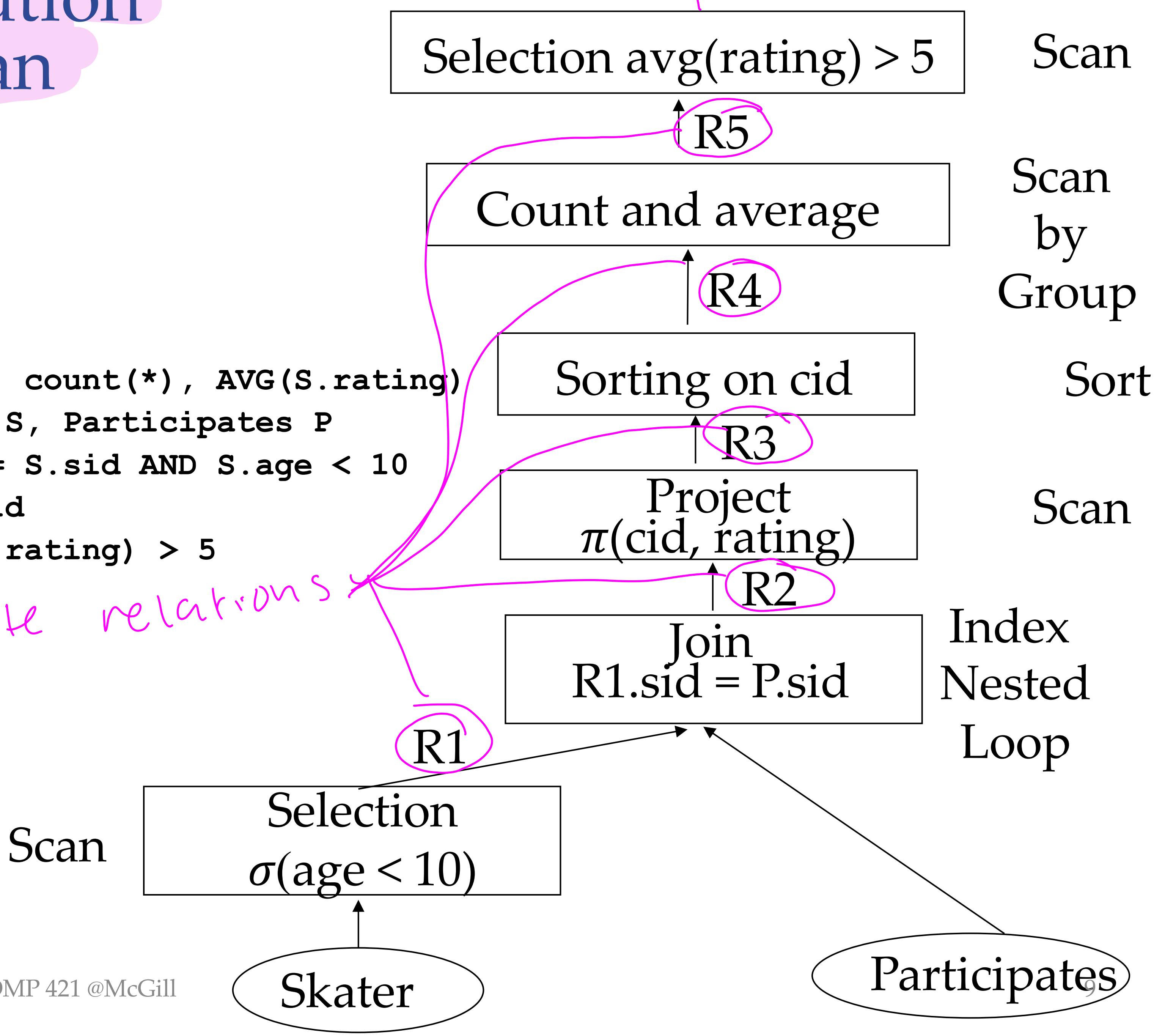
Concatenation of Operators

- Execution tree:
  - Leaf nodes are the base relations
  - Inner nodes are operators
  - Tuples from base relations (leaves) flow into the parent operator node(s)
  - Output of an operator node flows into the parent node
  - Output of the root flows as result to the client

final result to client

Execution Plan

```
SELECT P.cid, count(*), AVG(S.rating)
FROM Skaters S, Participates P
WHERE P.sid = S.sid AND S.age < 10
GROUP BY P.cid
HAVING AVG(S.rating) > 5
```



intermediate relations

could have write access cost if need to write large intermediate relation to disk

Schema for Examples

Users (*uid*: int, *uname*: string, *experience*: int, *age:* int)
GroupMembers (*uid*: int, *gid*: int, *stars*: int)

cardinality

Statistics

- Users (U):
  - 40,000 tuples (denoted as CARD(U))
  - Around 80 tuples per data page
  - 500 data pages (denoted as UserPages)
  - An index on uid has around 170 leaf pages
  - An index on uname has around 300 leaf pages
- GroupMembers (GM):
  - 100,000 tuples (denoted as CARD(GM))
  - Around 100 tuples per data page
  - Total of 1000 data pages (denoted as GroupMemberPages)
- Database statistics tables
  - Keep track of cardinality of tables, number of pages, what indexes, how big an index, etc.
  - keep track of domain of values and their rough distribution
    - E.g., experience values: 1…..10, uniform distribution

Selection

Selectivity / Reduction Factor

how many tuples will be in the result?

- Reduction Factor of a condition is defined as — # tuples in result / total # tuples = reduction factor
  - $Red(\sigma_{condition}(R)) = |\sigma_{condition}(R)| \ / \ |R|$
  - $Red(\sigma_{experience=5}(Users)) = |\sigma_{experience=5}(Users)| \ / \ |Users| = ?$
  - Assume we know that 10,000 users have experience of 5
    - 10.000/40.000 = 0.25
- If not known, DBMS makes simple assumptions
  - $Red(\sigma_{experience=5}(R)) = 1/|\text{different experience levels}| = 0.1$ = reduction factor
    - Uniform distribution assumed
  - range queries $Red(\sigma_{age<=16}(R)) = (16 - \min(age)+1) \ / \ (\max(age) - \min(age)+1) = (16 – 12+1)/(61 – 12+1) = 5/50 = 0.1$
    - Size of selected range / total size of domain
  - $Red(\sigma_{experience=5 \text{ and } age <= 16}(R)) = ?$
    - Assume uniform and independent distribution
    - $Red(\sigma_{experience=5}(R)) * Red(\sigma_{age <= 16}(R))$

Selection
Selectivity / Reduction Factor

#result tuples

- Result sizes: number of input tuples multiplied by reduction factor
  - Assume reduction factor of a condition is 0.1
  - Assume 40000 tuples
  - Result size = 40000*0.1 = 4000 tuples
- How to know number of different values, how many of a certain value, max, min…:
  - through indices, heuristics, separate statistics (histograms)

string starts with B

Simple Selections

```
SELECT   *
FROM     Users
WHERE    uname LIKE 'B%'
```

```
SELECT   *
FROM     Users
WHERE    uid = 123
```

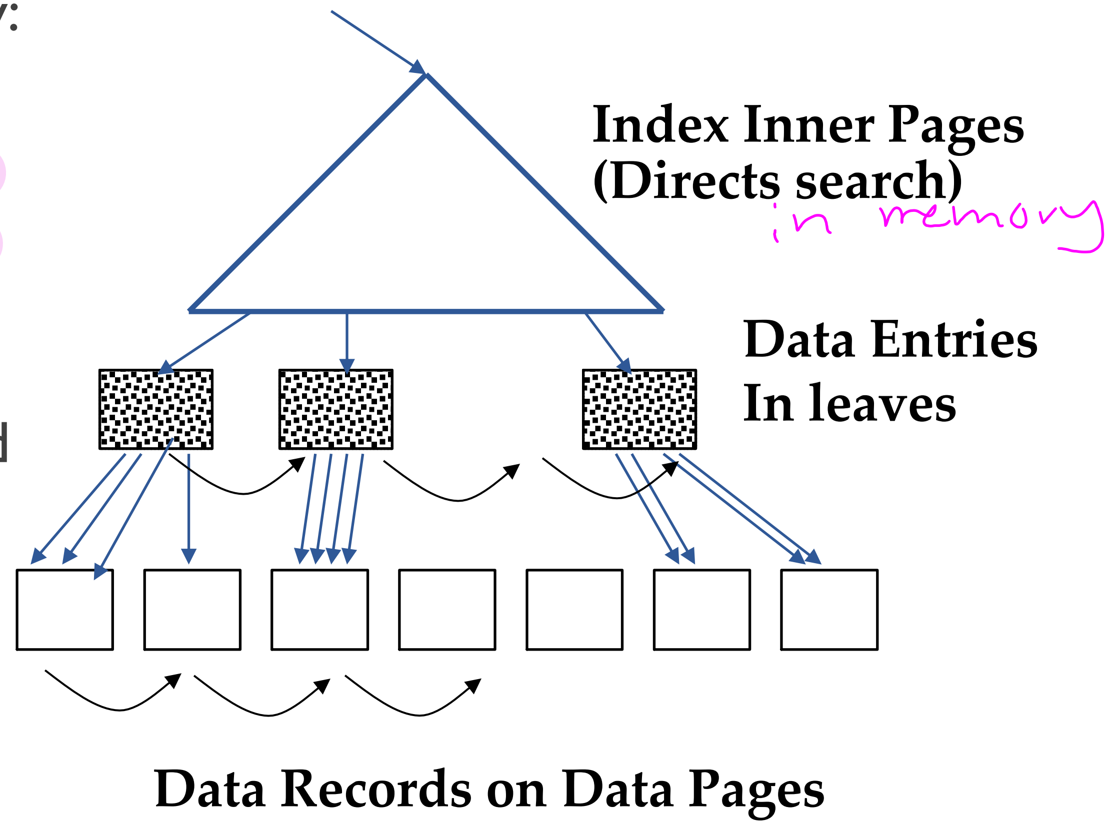
- General form: $\sigma_{R.attr\ op\ value}(R)$
- No index:
  - Search on arbitrary attribute(s): scan the relation (cost is UserPages=500) all of U
  - Search on primary key attribute: scan on average half of U (cost is UserPages/2=250) stop as soon as you find it
- Index on selection attributes: indexing
  - Use index to find qualifying data entries, then retrieve corresponding data records.
  - I/O: number of leaf pages with maches + certain number of data pages that have matching tuples
  - Now how much is that??
    - Depends on number of qualifying tuples
    - Depends on type of index

inclustering

In case of Clustered B+ Tree

Cost:

- Path from root to the left-most leaf lq with qualifying data entry:
  - Inner pages in memory
  - One I/O to retrieve lq page
- Retrieve page of first qualifying tuple: 1 I/O 1
- Retrieve all following pages as long as search criteria is fulfilled
- Each data page only retrieved once
- # data pages
  - #matching tuples / tuples per page
  - E.g., if 20% of tuples qualify then roughly 20% of data pages are retrieved



1 leaf page access + reduction factor * num pages

1 leaf page access + (reduction factor) num pages

Clustered Index for our Example

- Clustered: matching tuples are in adjacent data pages:
  - # datapages = number of matching tuples / number of tuples per page
  - Example 1: `SELECT * FROM Users WHERE uid = 123`
    - 1 tuple matches because primary key (1 data page)
    - I/O cost: 1 index leaf page + 1 data page
  - Example 2: `SELECT * FROM Users WHERE uname LIKE 'B%'`
    - System estimates the number of matching tuples
      - e.g., around 100 tuples match
    - Since clustered there are all on few pages (say 2 data pages)
    - I/O cost: 1 index leaf page + two data pages
  - Example 3: `SELECT * FROM Users WHERE uname < 'F'`
    - Estimate is that around 10000 tuples match
    - i.e., if 25% of the data, then clustered on approx. 25% of the pages (125 data pages)
    - Cost: 1 leaf pages + 125 data pages

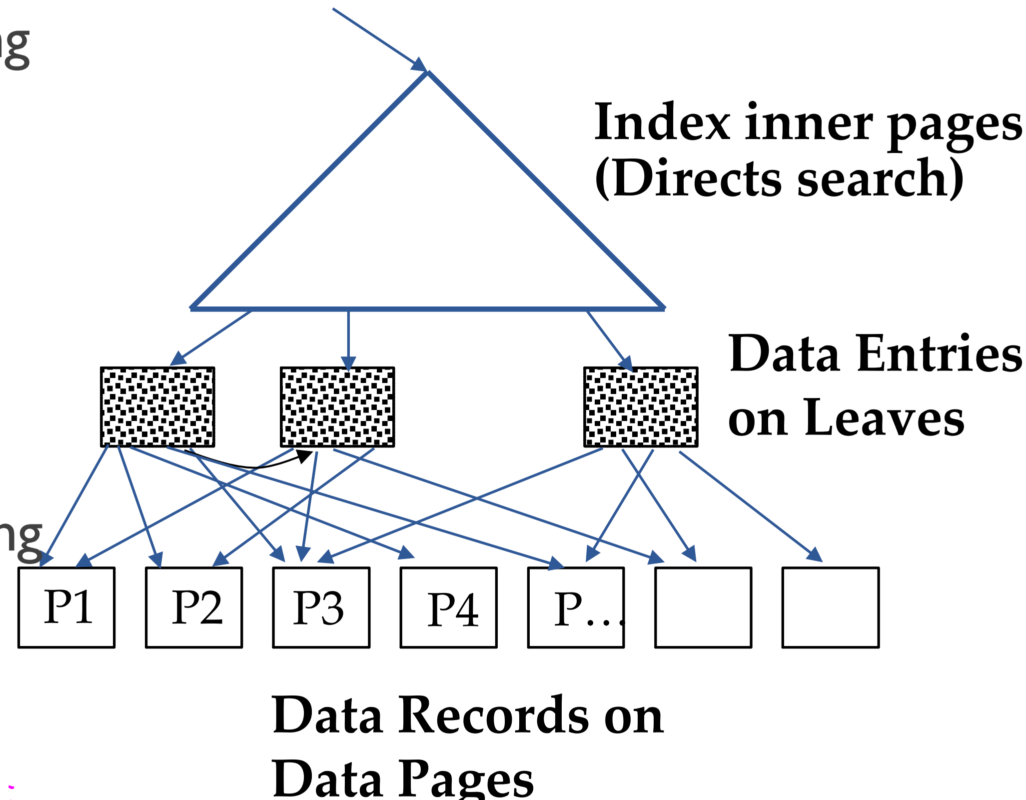
  - Note: some systems might retrieve all rids through the index (not efficient).
  - In this case, example 3 will read approx. 25% of the leaf pages

In case of non-clustered B+ Tree

Cost:

- Path to first qualifying leaf:
  - One I/O to retrieve lq page
- Retrieve page of first qualifying tuple: 1 I/O
- Retrieve page of second qualifying tuple
- …
- Retrieve next leaf page with qualifying tuple
- Retrieve page of next qualifying tuple ….
- Sometimes page might have been retrieved previously
  - Might still be in main memory :)
  - Might have been replaced again :(extra IO

each leaf page accesses multiple data pages



→ Might result in one I/O per data record!

Using an non-clustered Index for Selections

- Non clustered, simple strategy:
  - get one page after the other:
  - Worst case #data pages = #matching tuples
  - Example 1: `SELECT * FROM Users WHERE uid = 123`
    - Same as before point query
  - Example 2: `SELECT * FROM Users WHERE uname LIKE 'B%'`
    - Estimated that around 100 tuples match
    - The data entries for that will be on 1-2 leaf page
    - In worst case they are on 100 different data pages
    - But even if two are on the same page, when the second record is retrieved the page might already be no more in main memory..
    - cost: 1-2 index-leaf-page + 100 pages (some pages might be retrieved several times) still better than entire relation

Using an non-clustered Index for Selections

- Example 3: `SELECT * FROM Users WHERE uname < 'F'`
  - Estimated that 10000 tuples match = 25%
  - Likely that nearly every data page has a matching tuple! (maybe 10 don't???)
  - But as we retrieve tuple by tuple, every retrieval might lead to I/O as page might have been purged from main memory in between
  - cost: 75 leaf pages + 10000 pages (most pages will be retrieved several times)
  - 75 leaves = 25% of all leaf pages
  - Simple scan is faster!!
- Lesson learned:
  - Non-clustered Index usually only useful with very small reduction factors

Using an non-clustered Index with Sorting for Selections

- Example 3: **`SELECT * FROM Users WHERE uname < 'F'`**
  - Determine all leaf pages that have matching entries (75 leaf pages)
  - sort matching data entries (rid=pid,slot-id) in leaf-pages by page-id
  - Only fast if the 75 leaf pages with matching entries fit in main memory
  - Retrieve each page only once and get all matching tuples
  - #data pages = #data pages that have at least one matching tuple;
    - worst case is total # of data pages

  - For Example 3
    - Around 10000 tuples match
    - If they are distributed over 490
      - cost: 75 leaf pages + 490 pages (worse than a scan)
    - If they are distributed over 300 pages
      - Cost 75 leaf pages + 300 pages (better than a scan)
  - Note: sorting expensive if leaf-pages do not fit in main-memory

Selections on 2 or more attributes

- A=100 AND B=50 (pages = 500)
  - No index:
    - 500 scan all pages
  - Index on A attribute:
    - Get all tuples for A=100 through index, check value of B
    - Cost same as the query for A=100 have to check all tuples with A=100
  - 2 indexes; one for each attribute
    - Intersection based:
      - Find rids where A = 100 through A index
      - Find rids where B = 50 through B index
      - Build intersection of rids
      - Retrieve from data pages all tuples with rids in that intersection

      good if all rids fit into main mem
    - Use only one index
      - Use index on A if A is unique or has small reduction factor
  - 1 index with both attributes — good
- A=100 and B<50 (Red(B<50) = 0.5)
  - Very low reduction factor for B<50, ➔ index on B not much use

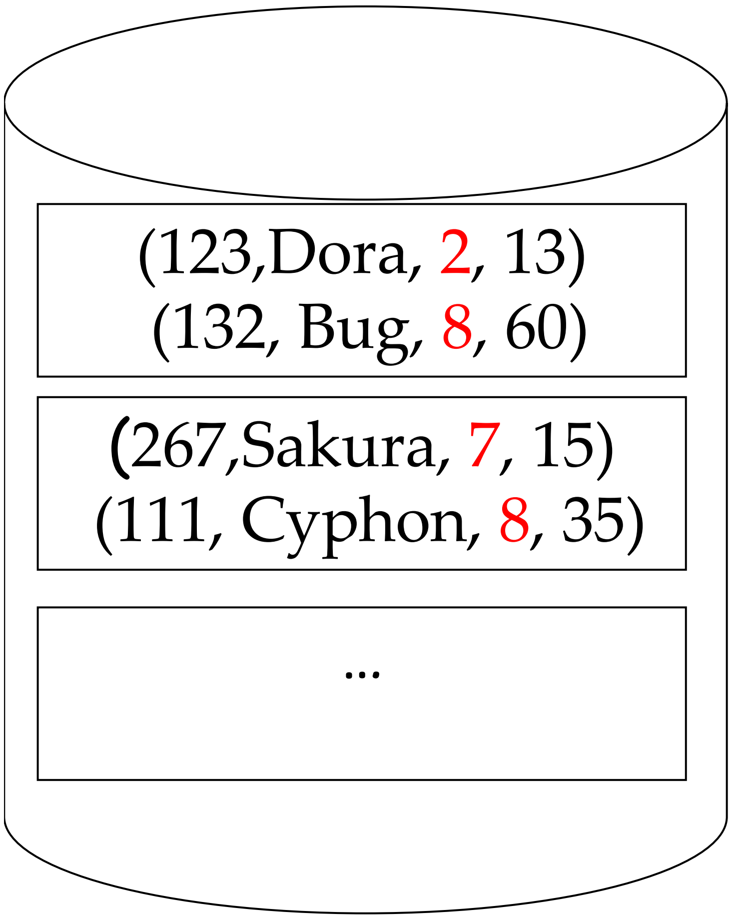
    A's reduction factor matters more

Selections on 2 or more attributes

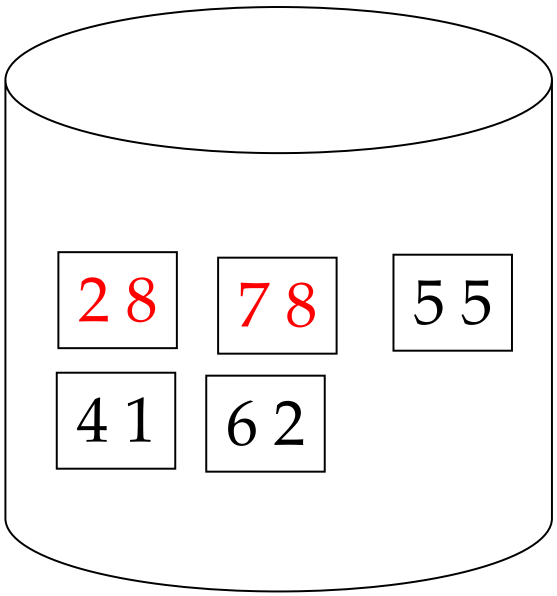
- A=100 OR B=50 (pages = 500)
  - No index:
    - 500 ← all
  - Index on A attribute:
    - Not useful
  - 2 indexes; one for each attribute
    - Find rids where A = 100 through A index
    - Find rids where B = 50 through B index
    - Build union of rids
    - Retrieve from data pages all tuples with rids in that union
  - 1 index with both attributes (A,B)
    - Could read through all data entries in all leaf pages.
    - Might or might not be faster than a basic scan depending on reduction factor

have to check all for both components

External Sorting Example Setup



Represented in the following as:

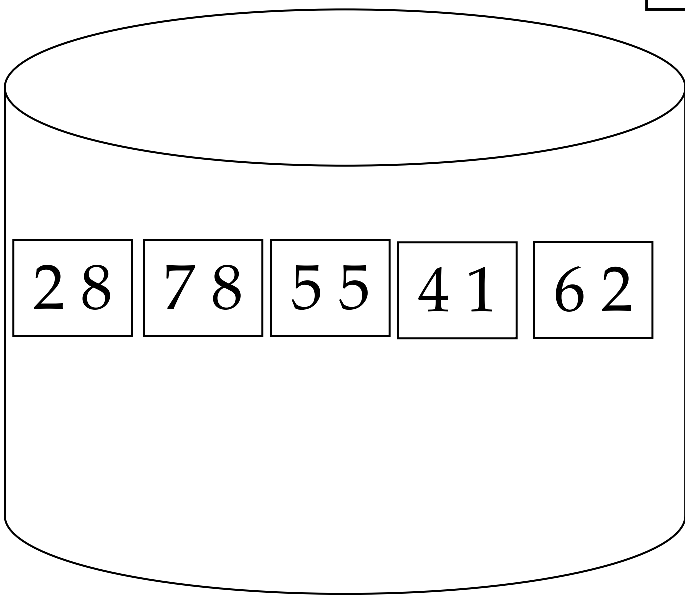


Summary of example:
- 5 pages
- Each with two tuples

External Sorting

Task:

```
SELECT   *
FROM     Users
ORDER BY experience
```

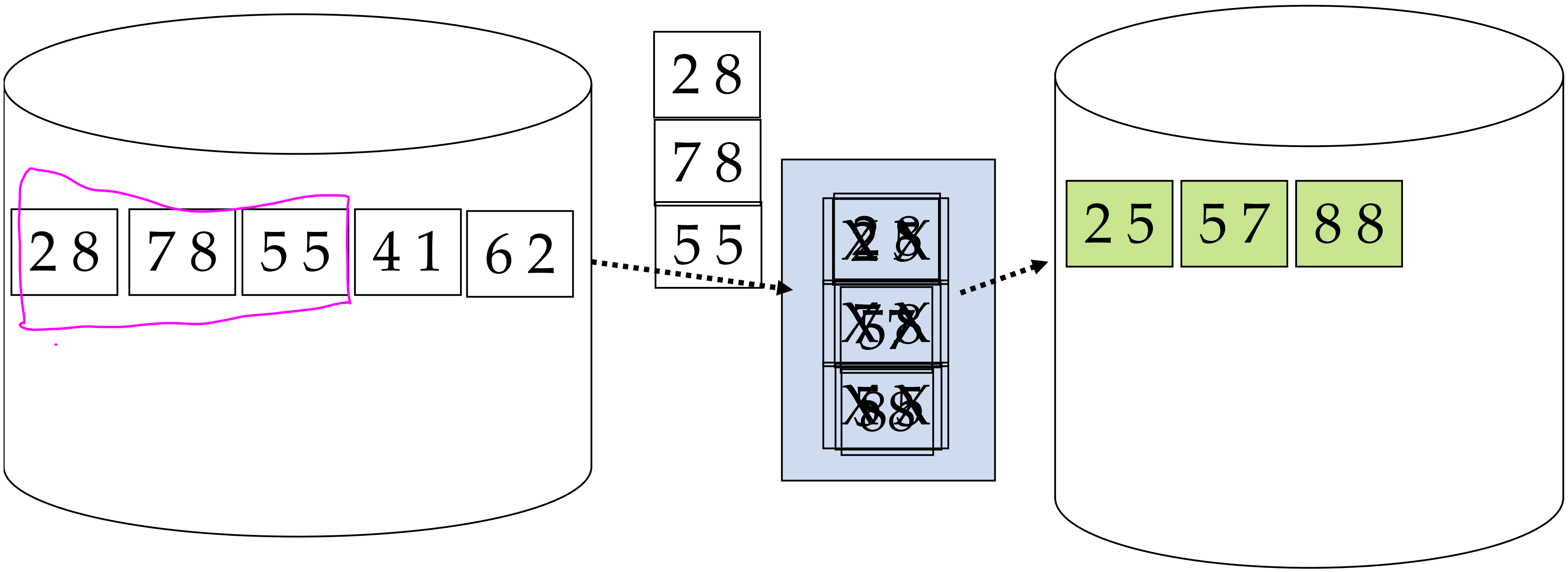


Summary of example:

- 5 pages
- Each with two tuples

What if only 3 buffer frames available?
That is, data pages do not fit into main memory

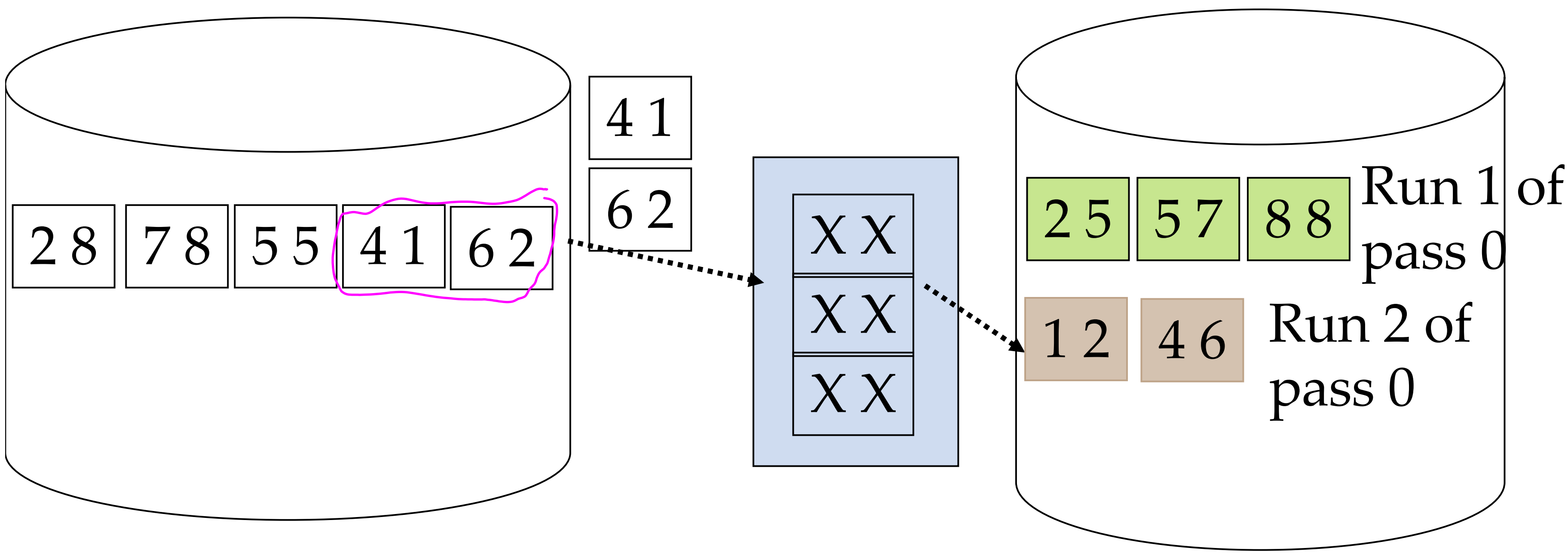
External Sorting - Pass 0



- In example: 5 pages, 3 buffer frames
- N pages, B buffer frames:
  - Bring B pages in buffer
  - Sort with any main memory sort
  - Write out to disk to a temporary file;
  - it's called a run of B pages

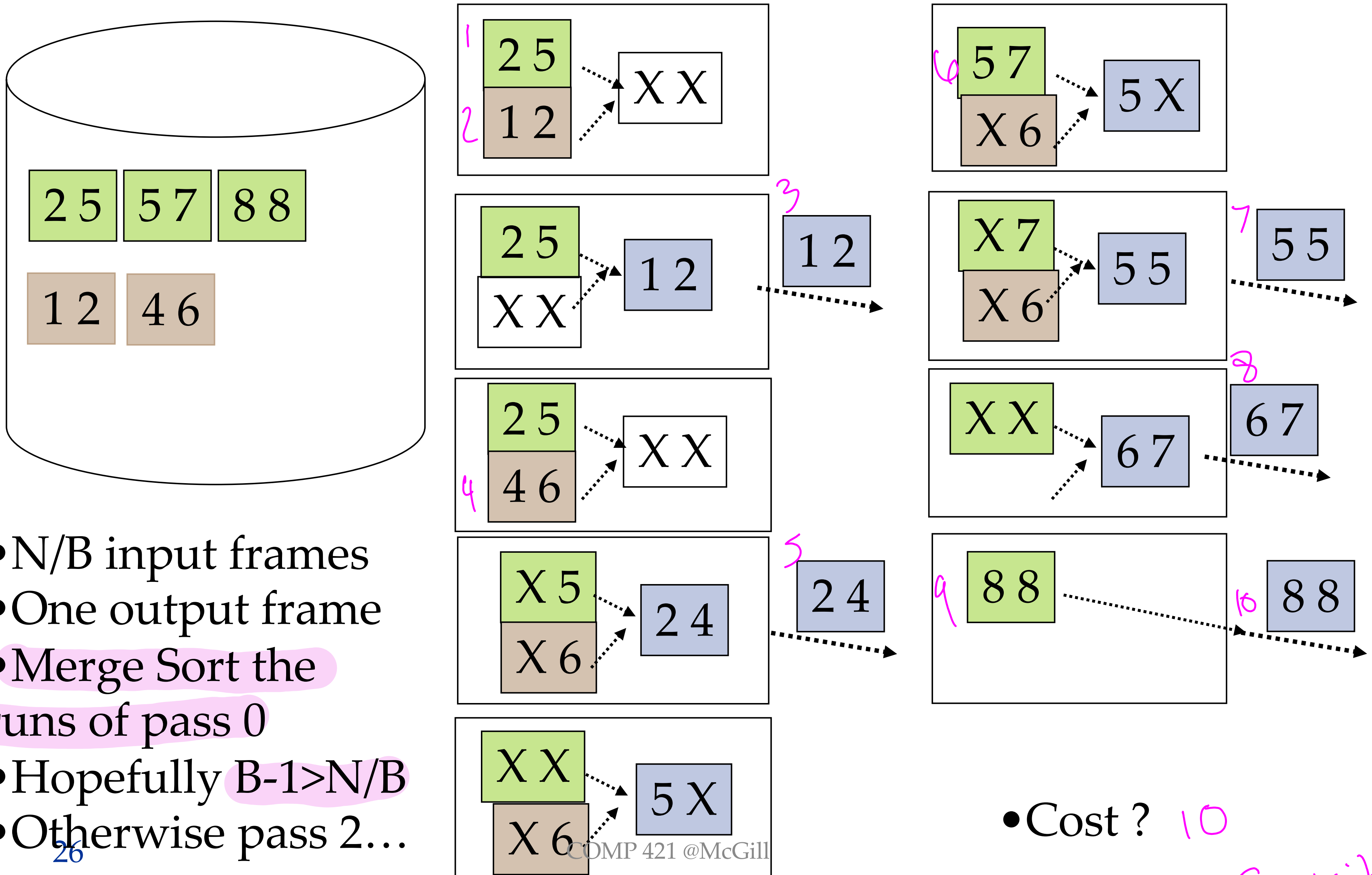
read in
↓
partial sort
↓
write intermediate out

External Sorting - Pass 0



- In example: 5/3 = 2 runs (round up!)
- In total N/B runs # pages / # blocks
- Cost ? → 10 (read 5 pages in, write 5 pages out)

External Sorting Pass 1:



- N/B input frames
- One output frame
- Merge Sort the runs of pass 0
- Hopefully B-1>N/B
- Otherwise pass 2…

- Cost ? 10

5 reads, 5 writes

Sort

- Sometimes a Pass 2 is needed:
  - Pass 0 created more runs than there are main memory buffers
  - Therefore Pass 1 produces more than one run
    - Take the first B-1 runs from pass 0 and merge them to one bigger Pass 1 run
    - Then take the next B-1 runs from pass 0 and merge them to one bigger Pass 1 run
    - …
  - Pass 2 takes the runs of Pass 1 and merges them
    - If there are less than B Pass 1 runs, then this is the final pass
    - Otherwise Pass 3…
- Number of passes: $1 + \lceil \log_{B-1} \underbrace{\lceil N / B \rceil}_{\text{\# runs in pass 0}} \rceil$
- Cost = 2N * (# of passes)

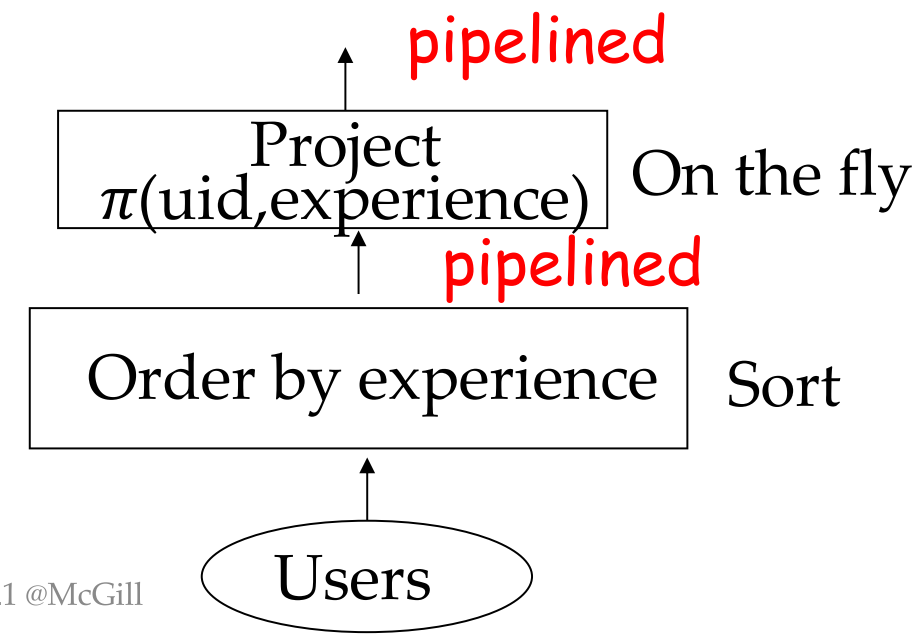
Sort Costs

Number of passes

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Sort together with other operators

- I/O Costs:
  - SELECT uname, experience FROM Users ORDER BY experience
  - If everything fits into main memory (Only pass 0 needed):
    - Read number of data pages
    - sort and pipeline result into next operator: π(uname,experience)
  - Pass 0 + pass 1 needed
    - Pass 0: read # pages, write # pages (have to write temp. results!)
    - Pass 1: read # pages, sort and pipeline result into next operator
    - 3 * #pages
  - Pass 0 + pass1 + pass2 needed
    - 5 * #pages



Sort in real life

- Blocked I/O:
  - use more output pages and flush to consecutive blocks on disk
  - Might lead to more I/O but each I/O is cheaper
- Other optimizations:
  - At write in Pass 0 to disk (if needed):
    - Do projection on necessary attributes → each tuple is smaller → less pages
    - that is, projection is pushed to the lowest level possible

example
on
mycourses
slides