Query Evaluation

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
    - 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

Basic Query Processing Operations

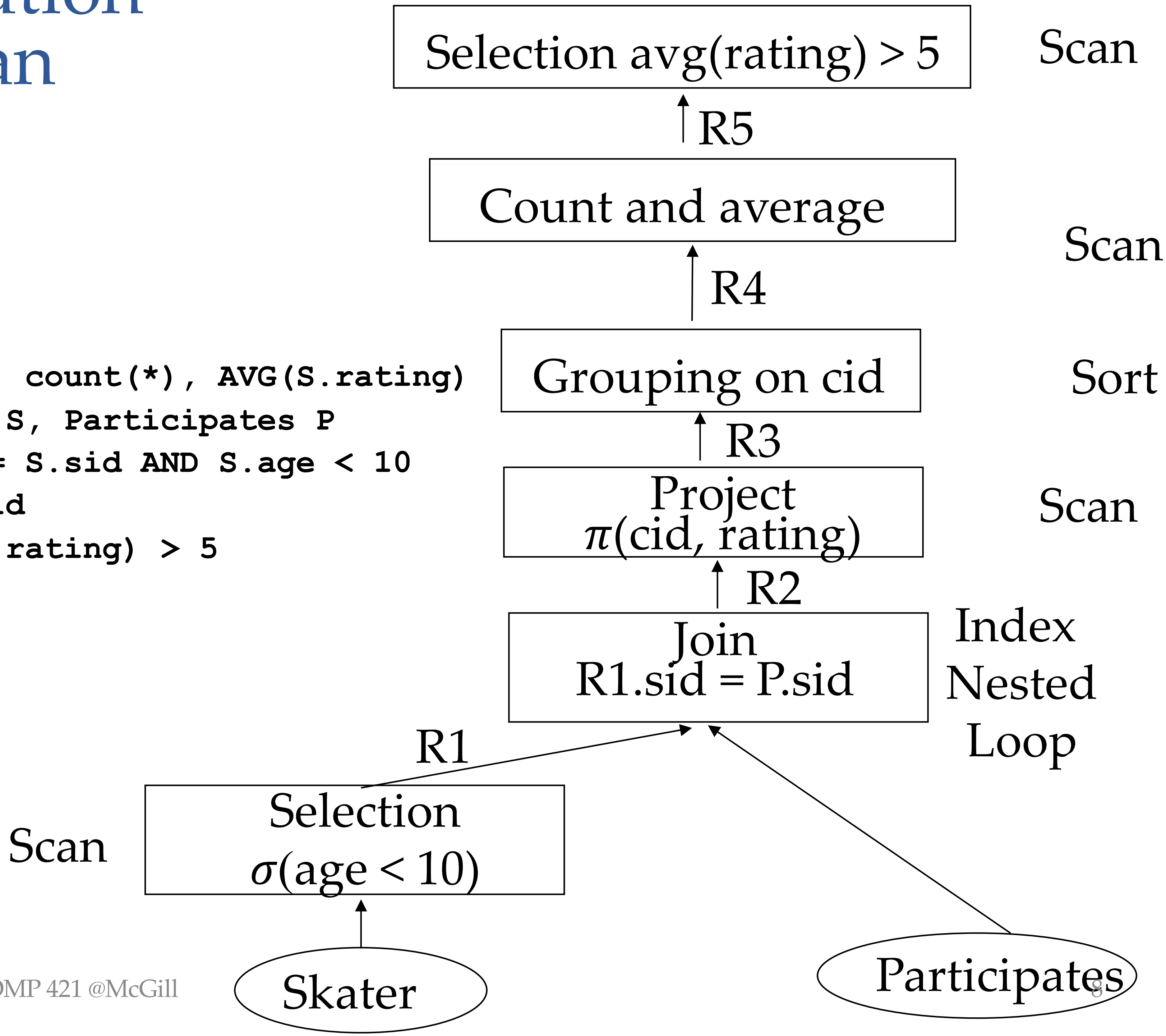
- Selection: σ
  - Scan (without an index)
  - Use an index involving the attributes of the selection
- Projection: π
- Sorting
- Joining
  - Nested loop join
  - Sort-merge join
  - Many others…
- Grouping and duplicate elimination

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

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
```



Schema for Examples

Users (*<u>uid</u>*: int, *uname*: string, *experience*: int, *age:* int)
GroupMembers (*<u>uid</u>*: int, *<u>gid</u>*: int, *stars*: int)

- 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., rating values: 1…..10, uniform distribution

Selection
Selectivity / Reduction Factor

- Reduction Factor of a condition is defined as
  - $Red(\sigma_{condition}(R)) = |\sigma_{condition}(R)| / |R|$
  - $Red(\sigma_{experience=5}(Users)) = |\sigma_{experience=5}(Users)| / |Users| = ?$
  - Assume we know that 10,000 users have experience of 5
    - 10.000/40.000 = 0.25
- If not known, DBMS makes simple assumptions
  - $Red(\sigma_{experience=5}(R)) = 1/|\text{different experience levels}| = 0.1$
    - Uniform distribution assumed
  - $Red(\sigma_{age<=16}(R)) = (16 - \min(age)+1) / (\max(age) - \min(age)+1) = (16 – 12+1)/(61 – 12+1)= 5/50= 0.1$
    - 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 sizes: number of input tuples multiplied by reduction factor
- How to know number of different values, how many of a certain value, max, min…:
  - through indices, heuristics, separate statistics (histograms)

Simple Selections

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

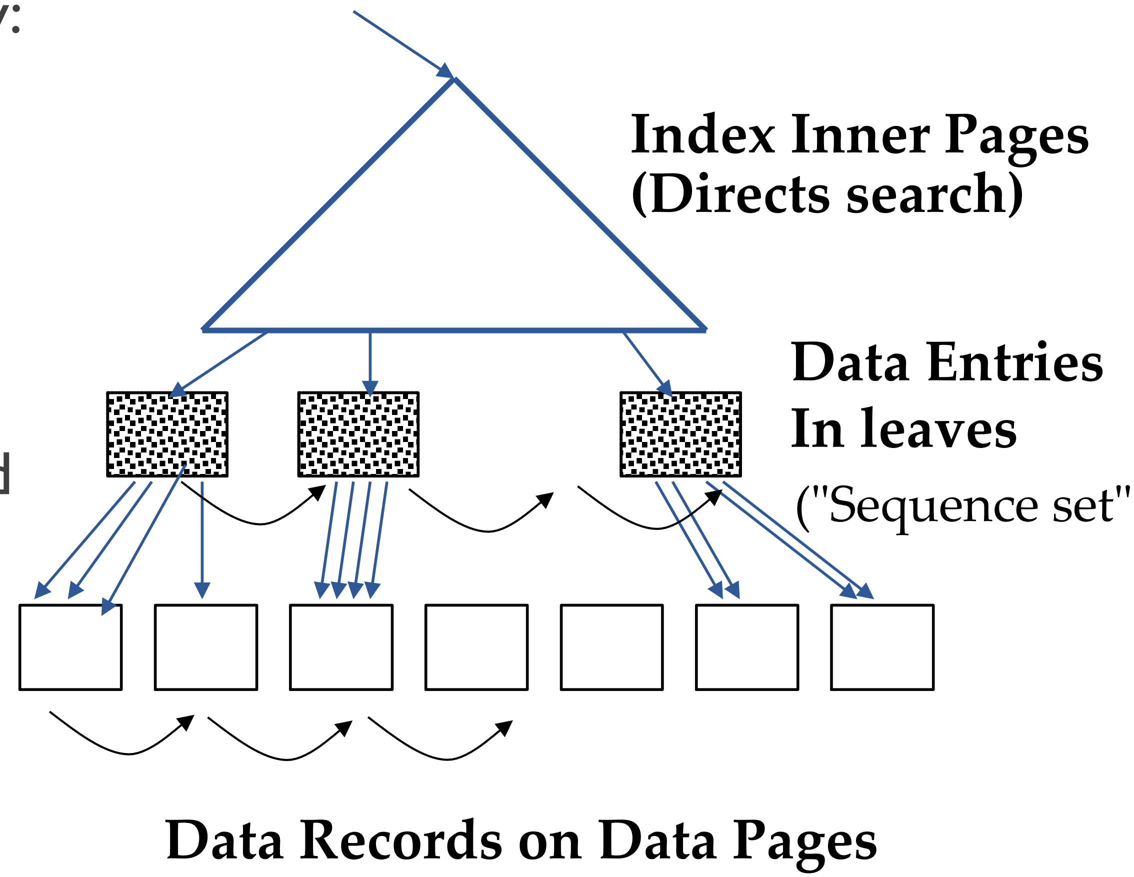
```
SELECT   *
FROM     Users
WHERE    uid = 123
```

- General form: $\sigma_{R.attr\ op\ value}(R)$
- No index:
  - Search on arbitrary attribute(s): scan the relation (cost is UserPages=500)
  - Search on primary key attribute: scan on average half of U (cost is UserPages/2=250)
- Index on selection attributes:
  - 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

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
- 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



Clustered Index for our Example

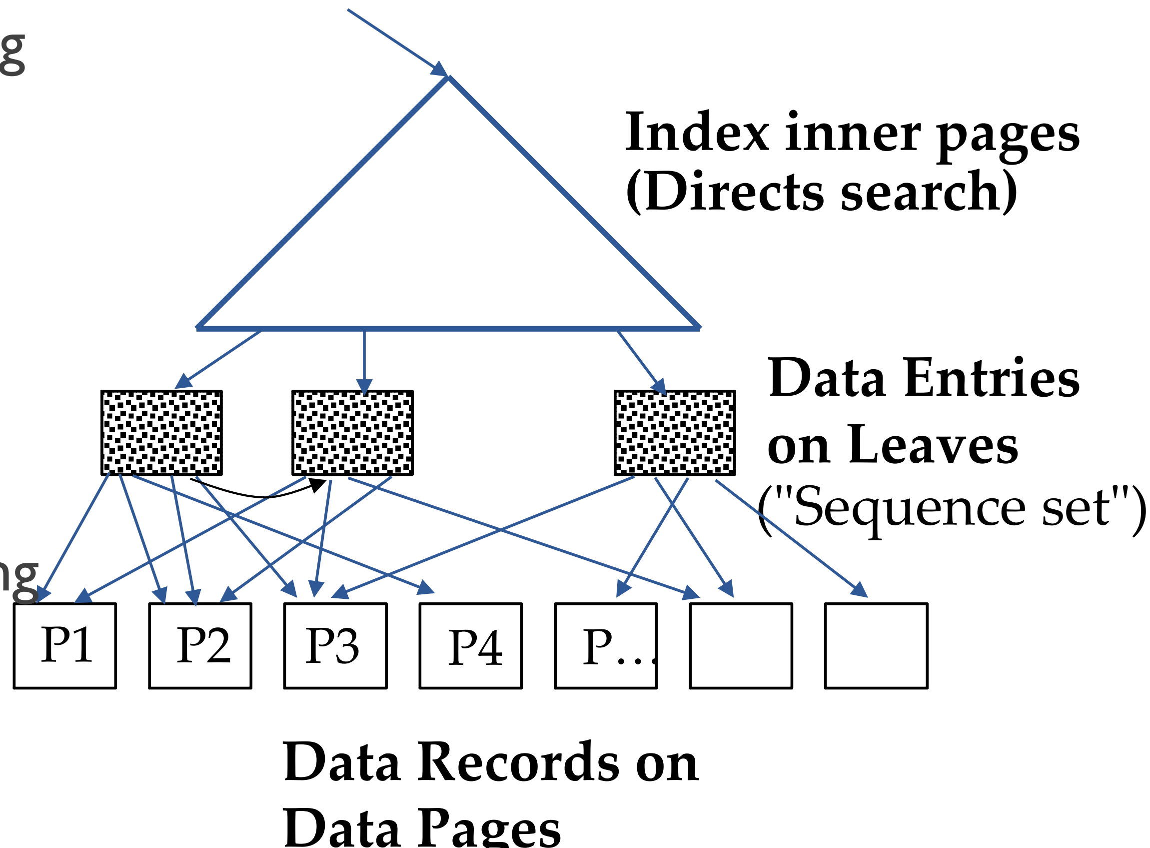
- Clustered: matching tuples are in adjacent data pages:
  - # datapages = number of matching tuples / number of tuples per page
  - Example 1: `SELECT * FROM Users WHERE uid = 123`
    - 1 tuple matches because primary key (1 data page)
    - I/O cost: 1 index leaf page + 1 data page
  - Example 2: `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



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
  - Example 2: `SELECT * FROM Users WHERE uname LIKE 'B%'`
    - Estimated that around 100 tuples match
    - In worst case there 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 index-leaf-page + 100 pages (some pages are retrieved twice)

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:
  - Indices 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

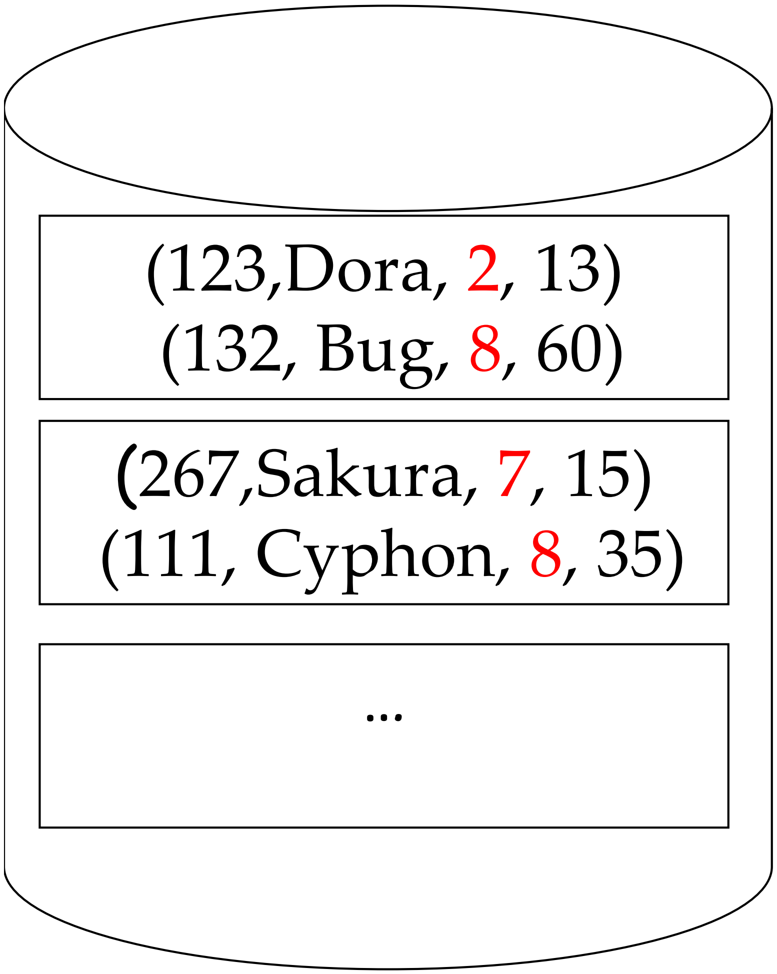
Selections on 2 or more attributes

- A=100 AND B=50 (pages = 500)
  - No index:
    - 500
  - Index on A attribute:
    - Get all tuples for A=100 through index, check value of B
    - Cost same as the query for A=100
  - 2 indexes; one for each attribute
    - 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
    - In some cases can just use A's index (e.g. when A is unique or has small reduction factor)
  - 1 index with both attributes
- A=100 and B<50 (Red(B<50) = 0.5)
    - Very low reduction factor for B<50, not much use.
- OR

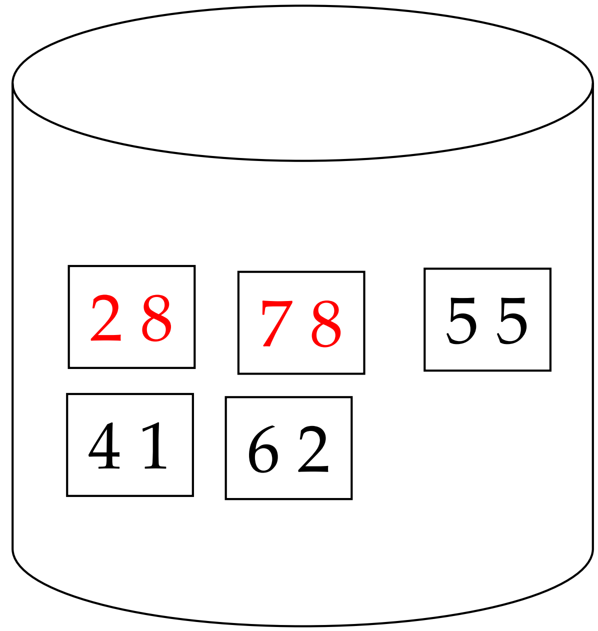
Selections on 2 or more attributes

- A=100 OR B=50 (pages = 500)
  - No index:
    - 500
  - 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)
    - Have to read all the leaf pages anyways.

External Sorting
Example Setup



Represented in the following as:



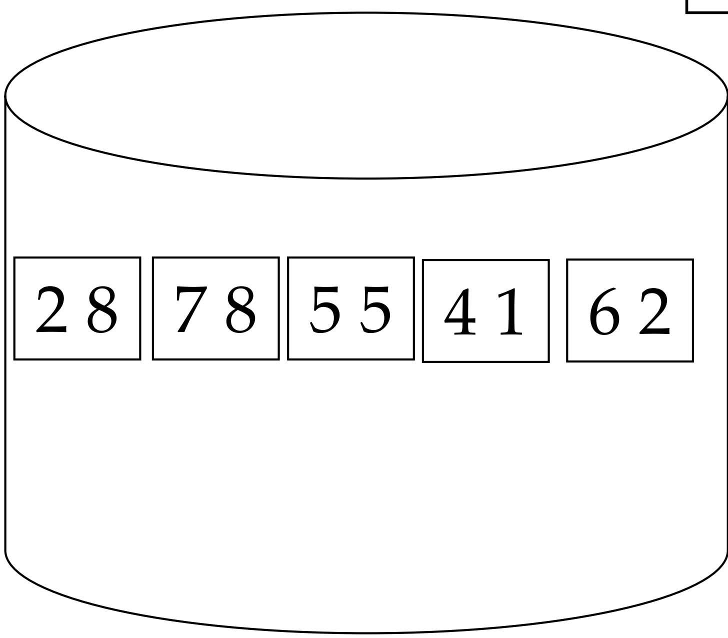
Summary of example:

- 5 pages
- Each with two tuples

External Sorting

Task:

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

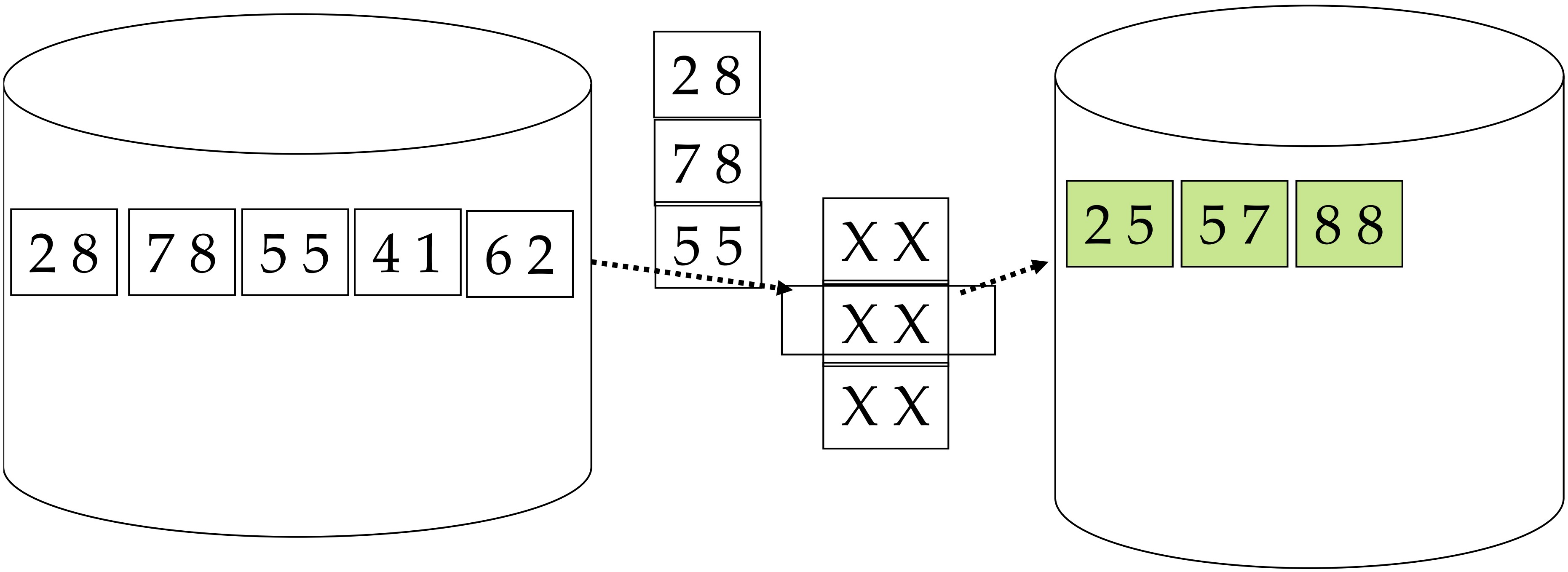


Summary of example:

- 5 pages
- Each with two tuples

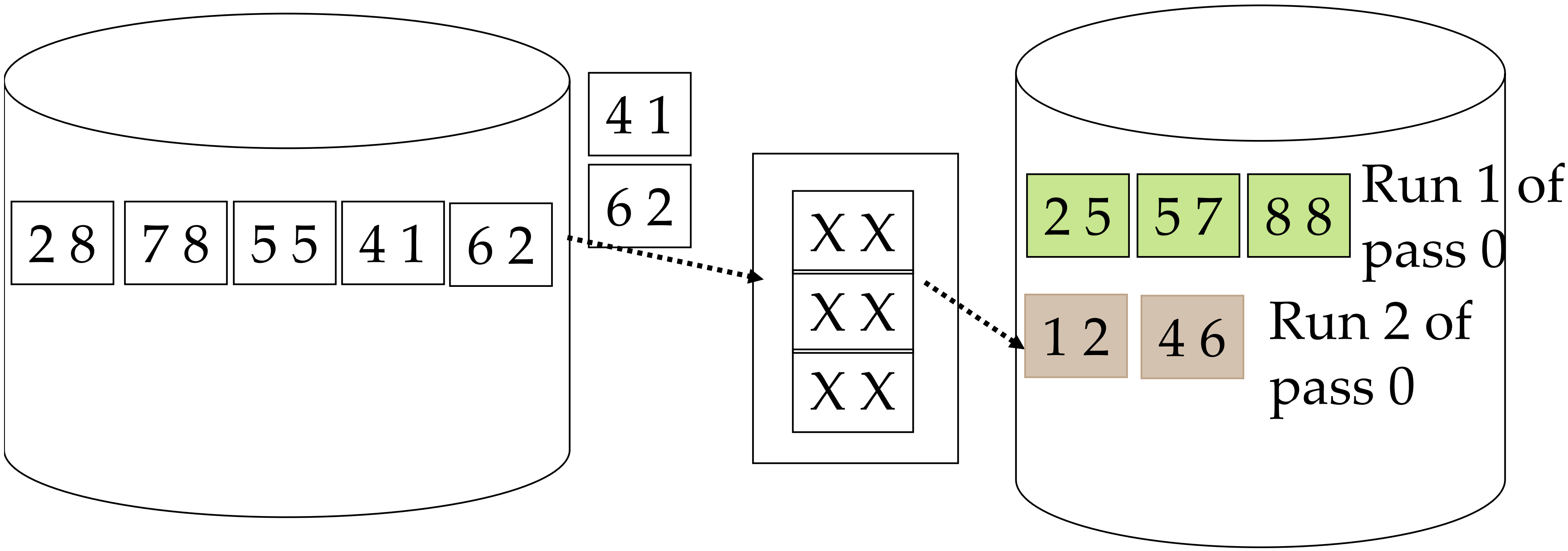
What if only 3 buffer frames available?
That is, data pages do not
fit into main memory

External Sorting - Pass 0



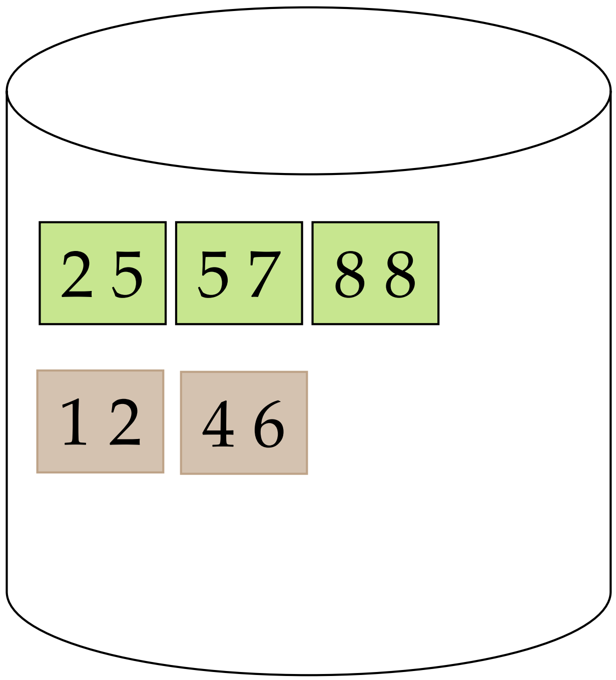
- In example: 5 pages, 3 buffer frames
- N pages, B buffer frames:
  - Bring B pages in buffer
  - Sort with any main memory sort
  - Write out to disk to a temporary file;
  - it's called a run of B pages

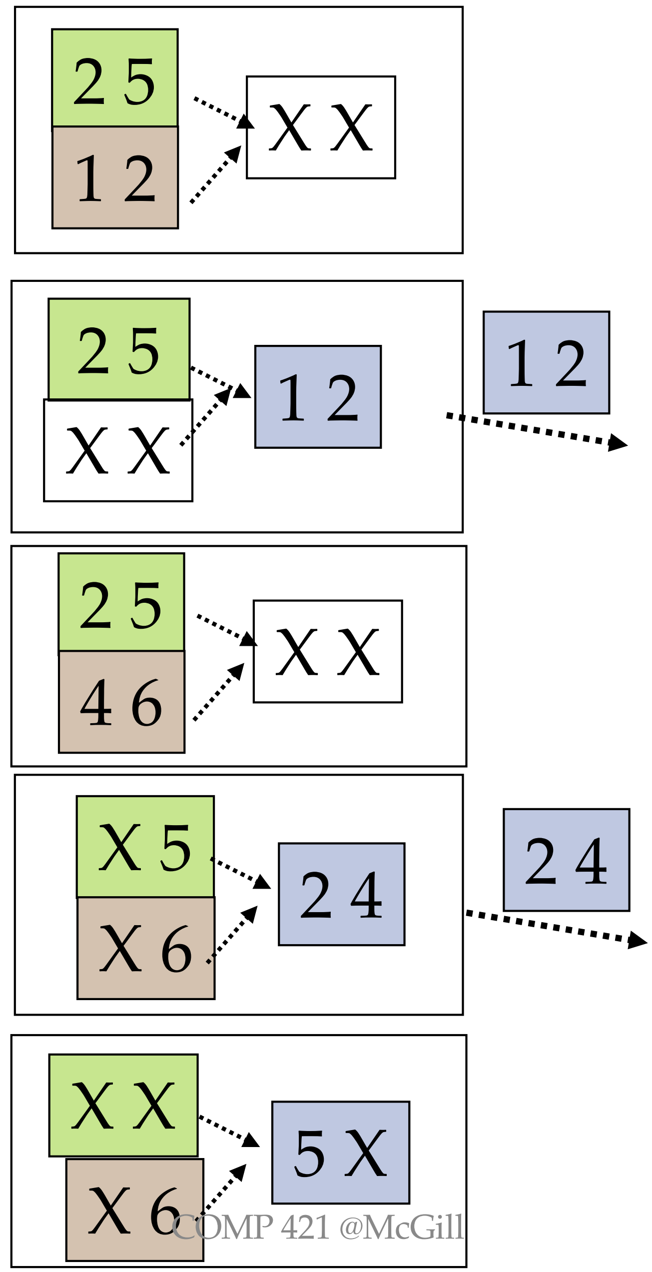
External Sorting - Pass 0

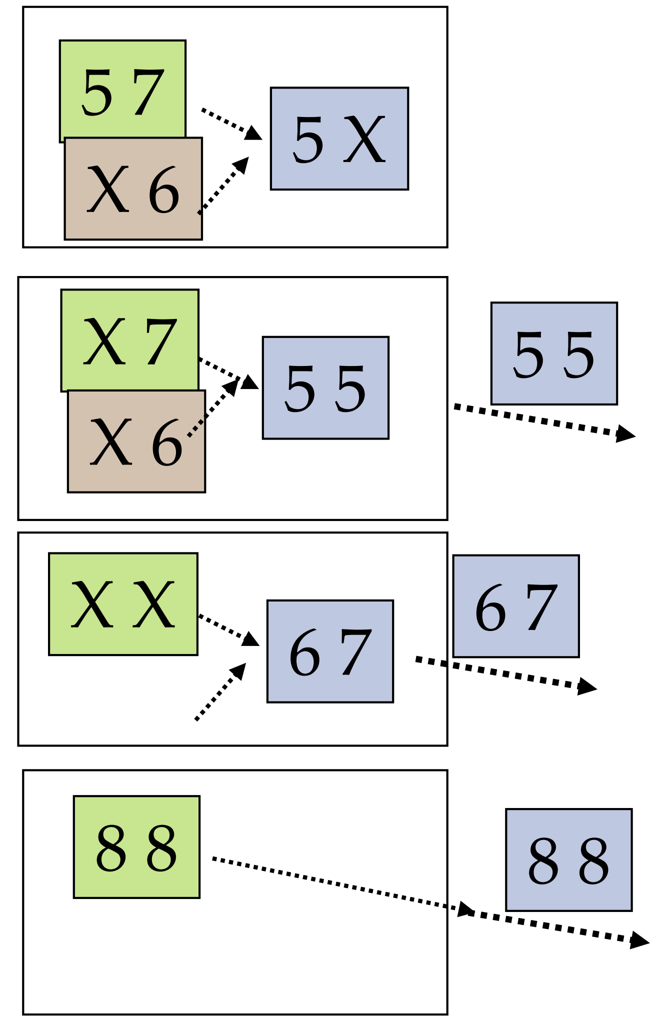


- In example: 5/3 = 2 runs (round up!)
- In total N/B runs
- Cost ?

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 ?

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 then 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: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)

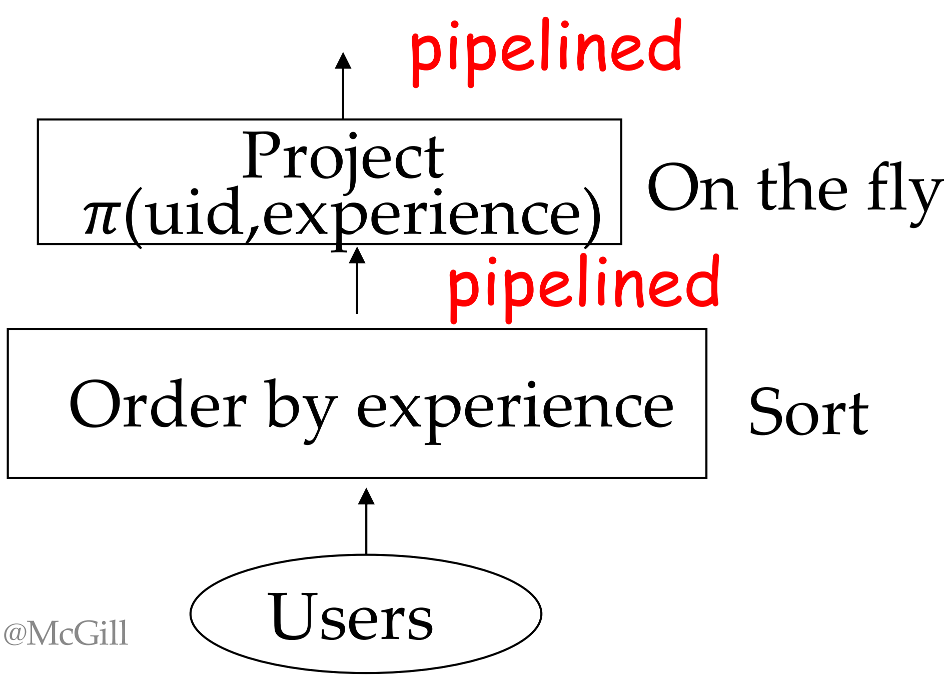
Sort Costs

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

Equality Joins

```
SELECT  *
FROM    Users U, GroupMembers GM
WHERE   U.uid = GM.uid
```

	uid	uname	experience	age
$p1_u$	123	Dora	2	13
	132	Bug	8	60
$p2_u$	267	Sakura	7	15
	111	Cyphon	8	35

	uid	gid	stars
$p1_g$	123	G1	2
	132	G1	5
$p2_g$	132	G2	3
	132	G3	1
$p3_g$	123	G2	4
	111	G4	2

Join Cardinality Estimation

- | Users ⋈ GroupMembers | = ?
  - Join attribute is primary key for Users
  - Each GroupMember tuple matches exactly with one Users tuple
  - Result: |GroupMembers|
- | Users X GroupMembers | = ?
  - Result: |Users| * |GroupMembers|
  - Cross product is always the product of individual relation sizes
- For other joins more difficult to estimate

Cardinality Estimation

- | Users $\bowtie \sigma_{(stars > 3)}$(GroupMembers) | = ?
  - Result: | $\sigma_{(stars > 3)}$(GroupMembers) |
  - Assuming 1-5 stars and 100,000 members:
    - 40,000
- | $\sigma_{(experience > 5)}$ (Users)$\bowtie$ (GroupMembers) | = ?
  - Assume 1-10 experience levels and 40,000 users and uniform distribution for experience
  - Red($\sigma_{(experience > 5)}$(Users)) = 1/2
  - Result: ½ * |GroupMembers|

Simple Nested Loop Join

- For each tuple in the *outer* relation Users U we scan the entire *inner* relation GroupMembers GM.

```
foreach tuple u in U do
 foreach tuple g in GM do
    if u.uid == g.uid  then add <u, g> to result
```

	uid	uname	experience	age
$p1_u$	123	Dora	2	13
	132	Bug	8	60
$p2_u$	267	Sakura	7	15
	111	Cyphon	8	35

	uid	gid	stars
$p1_g$	123	G1	2
	132	G1	5
$p2_g$	132	G2	3
	132	G3	1
$p3_g$	123	G2	4
	111	G4	2

Simple Nested Loop Join

- For each tuple in the *outer* relation Users U we scan the entire *inner* relation GroupMembers GM.

```
foreach tuple u in U do
 foreach tuple g in GM do
     if u.uid == g.uid  then add <u, g> to result
```

- Cost: UserPages + |Users| * GroupMemberPages = 500 + 40,000*1000 !

- NOT GOOD
- We need page-oriented algorithm!

Page Nested Loop Join

- For each *page* p_u of Users U, get each *page* p_g of GroupMembers GM
  - write out matching pairs <u, g>, where u is in p_u and g is in p_g.

```
For each page pu of Users U
   for each page pg of GroupMembers GM
      for each tuple u in pu do
         for each tuple g in pg do
            if u.uid == g.uid  then add <u, g> to result
```

	uid	uname	experience	age
$p1_u$	123	Dora	2	13
	132	Bug	8	60
$p2_u$	267	Sakura	7	15
	111	Cyphon	8	35

	uid	gid	stars
$p1_g$	123	G1	2
	132	G1	5
$p2_g$	132	G2	3
	132	G3	1
$p3_g$	123	G2	4
	111	G4	2

Page Nested Loop Join

- For each *page* p_u of Users U, get each *page* p_g of GroupMembers GM
  - write out matching pairs <u, g>, where u is in p_u and g is in p_g.

```
For each page pu of Users U
   for each page pg of GroupMembers GM
      for each tuple u in pu do
          for each tuple g in pg do
            if u.uid == g.uid  then add <u, g> to result
```

- Cost: UserPages + UserPages*GroupPages = 500 + 500*1000 = 500,500

Block Nested Loop Join

- For each *block of pages* bp_u of Users U, get each *page* p_g of GroupMembers GM
  - write out matching pairs <u, g>, where u is in bp_u and g is in p_g.
- *block of pages* bp_u and one page of GM must fit in main memory
  - For each block of pages bp_u
    - Load block into main memory
    - Get first page from GM
      - Do all the matching between users in bp_u and group members in first page
    - Get second page from GM (into the same frame the first one was in before)
      - Do all the the matching between users in bp_u and group members in second page
    - …
    - Get last page from GM (into again that frame reserved for GM)
      - ...
- Cost: UserPages + UserPages / $|bp_u|$ * GroupMemberPages

Block Nested Loop

	uid	uname	experience	age
$p1_u$	123	Dora	2	13
	132	Bug	8	60
$p2_u$	267	Sakura	7	15
	111	Cyphon	8	35

	uid	gid	stars
$p1_g$	123	G1	2
	132	G1	5
$p2_g$	132	G2	3
	132	G3	1
$p3_g$	123	G2	4
	111	G4	2

Block Nested Loop Join

- Examples depending on available main memory:
- 51 Buffer Frames:
  - 500 + 500/50 * 1000 = 500 + 10,000
- 501 Buffer Frames
  - 500 + 500/500 * 1000 = 500 + 1000
  - Special case: outer relation fits into main memory!!

Index Nested Loops Join

- For each tuple in the *outer* relation Users U we find the matching tuples in GroupMembers GM through an index
  - Condition: GM must have an index on the join attribute

```
foreach tuple u in U do
 find all matching tuples g in GM through index
   then add all <u, g> to result
```

	uid	uname	experience	age
$p1_u$	123	Dora	2	13
	132	Bug	8	60
$p2_u$	267	Sakura	7	15
	111	Cyphon	8	35

	uid	gid	stars
$p1_g$	123	G1	2
	132	G1	5
$p2_g$	132	G2	3
	132	G3	1
$p3_g$	123	G2	4
	111	G4	2

Index Nested Loops Join

```
foreach tuple u in U do
 find all matching tuples g in GM through index
   then add all <u, g> to result
```

- Index MUST be on the inner relation (in this case GM).
- Cost: OuterPages + CARD(OuterRelation) * cost of finding matching tuples in inner relation
- In example of previous page:
  - Index on uid on GM is clustered:
    - 500 + 40.000 * (1 leaf page +1 data pages)
  - Index on uid on GM is not clustered:
    - 500 + 40.000 * (1 leaf page + 2.5 data pages) (on average 2.5 tuples in GM per user)

Index Nested Loops Join

- Switch inner and outer if index is on uid of Users
- Note: uid is primary key in User
  - Only one tuple matches!

```
foreach tuple g in GM do
 find the one matching tuple u in U through index
   then add <g, u> to result
```

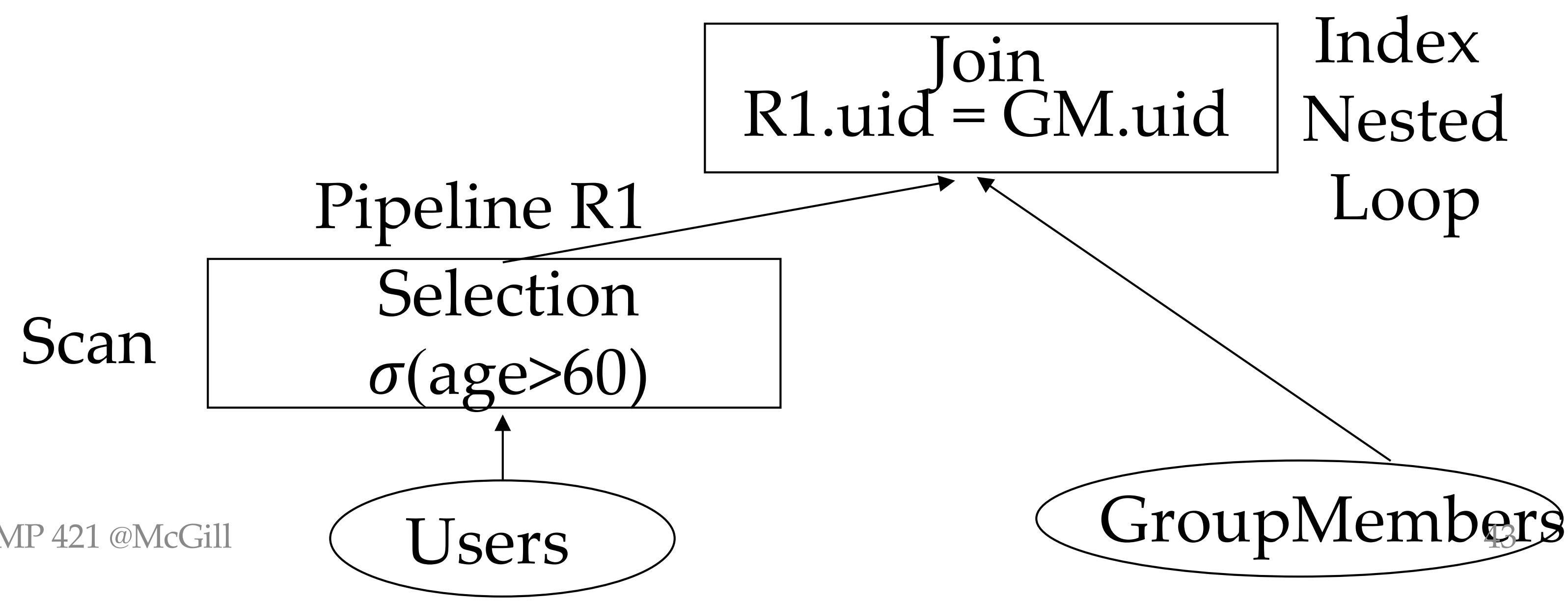
- Cost: 1000 + 100.000 * (1 leaf page + 1 data page)

	uid	gid	stars
$p1_g$	123	G1	2
	132	G1	5
$p2_g$	132	G2	3
	132	G3	1
$p3_g$	123	G2	4
	111	G4	2

	uid	uname	experience	age
$p1_u$	123	Dora	2	13
	132	Bug	8	60
$p2_u$	267	Sakura	7	15
	111	Cyphon	8	35

Block Nested Loop vs. Index

- Best Case for Block Nested Loop (if outer relation fits in main memory)
  - OuterPages + InnerPages
- Index Nested Loop:
  - OuterPages + Card(Outer) * matching tuples Inner
- Index Nested Loop wins if:
  - InnerPages > Card(Outer) * matching tuples Inner
  - E.g., if Outer is the result of a selection that only selected very few tuples
    - $\sigma_{(age > 60)}$ (Users) $\bowtie$ (GroupMembers)



Sort-Merge Join

- Sort U and GM on the join column, then scan them to do a "merge" (on join col.), and output result tuples.
  - In loop:
    - Assume the scan cursors currently points to U tuple u and GM tuple g. Advance scan cursor of U until u.uid >= g.uid and then advance scan cursor of GM so that g.uid >= u.uid. Do this until u.uid = g.uid.
    - At this point, all U tuples with same value in uid (*current U group*) and all GM tuples with same value in uid (*current GM group*) *match*; output <u, g> for all pairs of such tuples.
  - Then resume scanning U and GM.
- U is scanned once; each GM group is scanned once per matching U tuple. (Multiple scans of an GM group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

$p1_u$

$p2_u$

uid	uname	experience	age
111	Cyphon	8	35
123	Dora	2	13
132	Bug	8	60
267	Sakura	7	15

$p1_g$

$p2_g$

$p3_g$

uid	gid	stars
111	G4	2
123	G1	2
123	G2	4
132	G1	5
132	G2	3
132	G3	1

Cost of Sort-Merge Join

- Relations are already sorted:
  - UserPages + GroupMemberPages = 500 + 1000
- Relations need to be sorted – simple way:
  - Sort relations and write sorted relations to temporary stable storage
  - Read in sorted relations and merge
  - Costs: assuming 100 buffer pages
    - both Users and GroupMembers can be sorted in 2 passes (Pass 0 and 1): 4* UserPages + 4 GroupPages
    - Final merge: 500 + 1000 = (UserPages + GroupPages)
    - Total: 5 * UserPages + 5 GroupPages

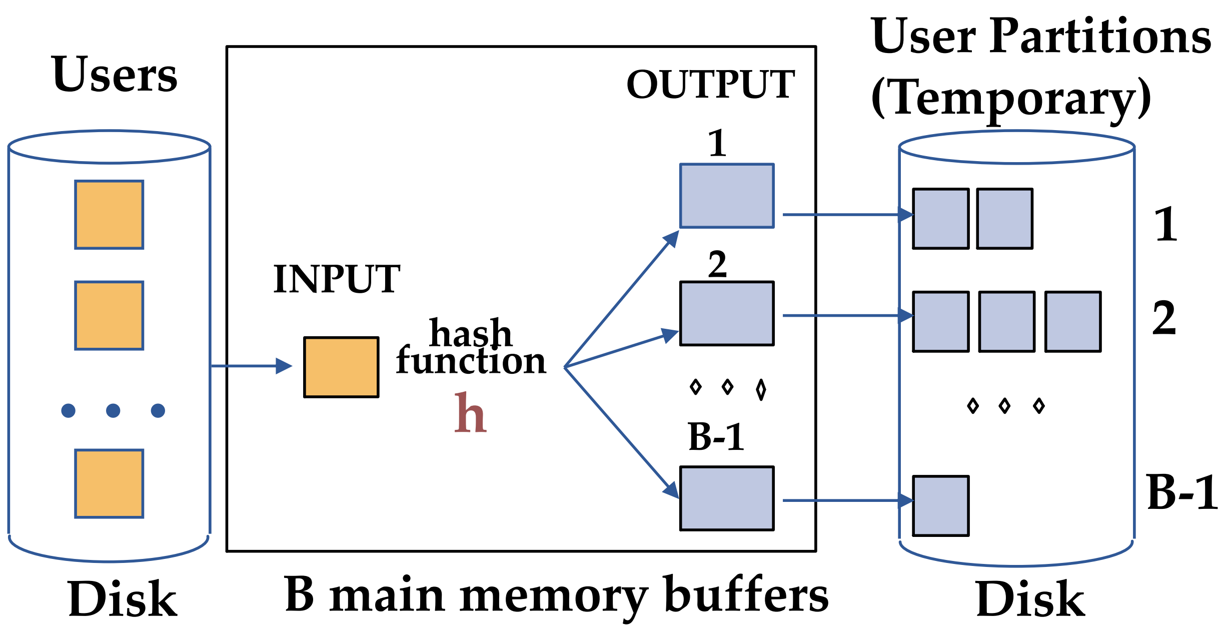
Cost of Sort-Merge Join

- Relations need to be sorted – use pipelining to combine last sort pass and join
  - Sorting performs Pass 0 reads and writes each of the relations: 2 * UserPages + 2 GroupPages
  - Pass 1 reads data, sorts and then performs merge in pipeline fashion (ignore details): UserPages + GroupPages
  - Total: 3 * UserPages + 3 * GroupPages = 4,500

Hash Join: first step

- Partition both relations using hash fn **h** (that returns a value between 1 and B-1 (if there are B buffer frames):
  - h(u.uid) = i → tuple u of User is in UserPartition i.
  - h(g.uid) = i → tuple g of GroupMember is in GroupMemberPartition i.
- Partitioning algorithm for relation U

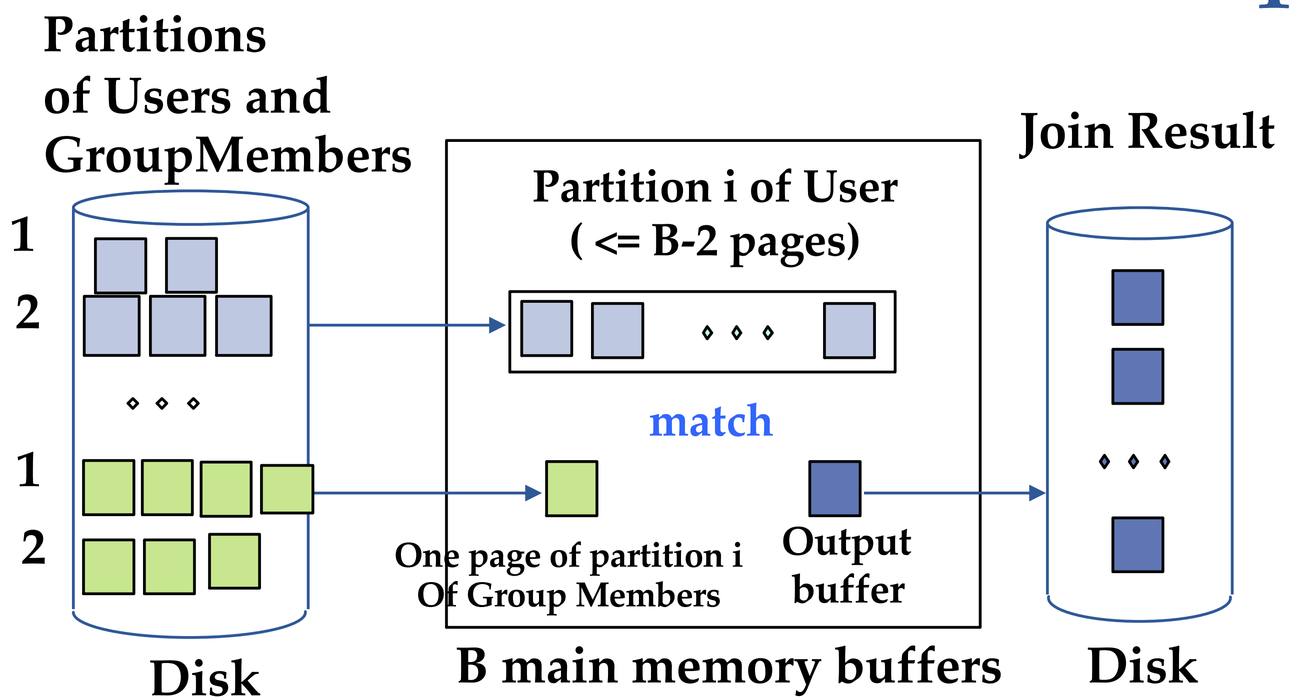
```
For each page of Users U
   for each tuple u in page do
       append u to UserPartition h(u.uid)
```



Hash Join: assumptions

- Assumption for this course:
  - Each partition of User fits into main memory and there are still 2 more buffer frames available
    - Each partition of User smaller than B-2
  - This holds if
    - Hash function creates partitions of equal size
      - Partition size for Users is UserPages / (B-1) = 500 / B-1
      - Partition size for GroupMembers is GroupMemberPages / (B-1) = 1000 / B-1
    - 500 / B-1 (rounded up) <= B -2
      - That is, buffer has at least 24 buffer frames (B=24).

Hash-Join: second step



- For u of User, g of GroupMember
  - u.uid == g.uid and u in UserPartition i ➔ g in GroupMemberPartition i
- Thus, for each i, join UserPartition i with GroupMemberPartition i only.

```
For each i, 1 < … i < ... Number of partitions
   Load partition i of Users into main memory
   For each page of partition i of GroupMembers
      Load page into main memory
      write all matching tuples (u,g) with u.uid = g.uid to output
```

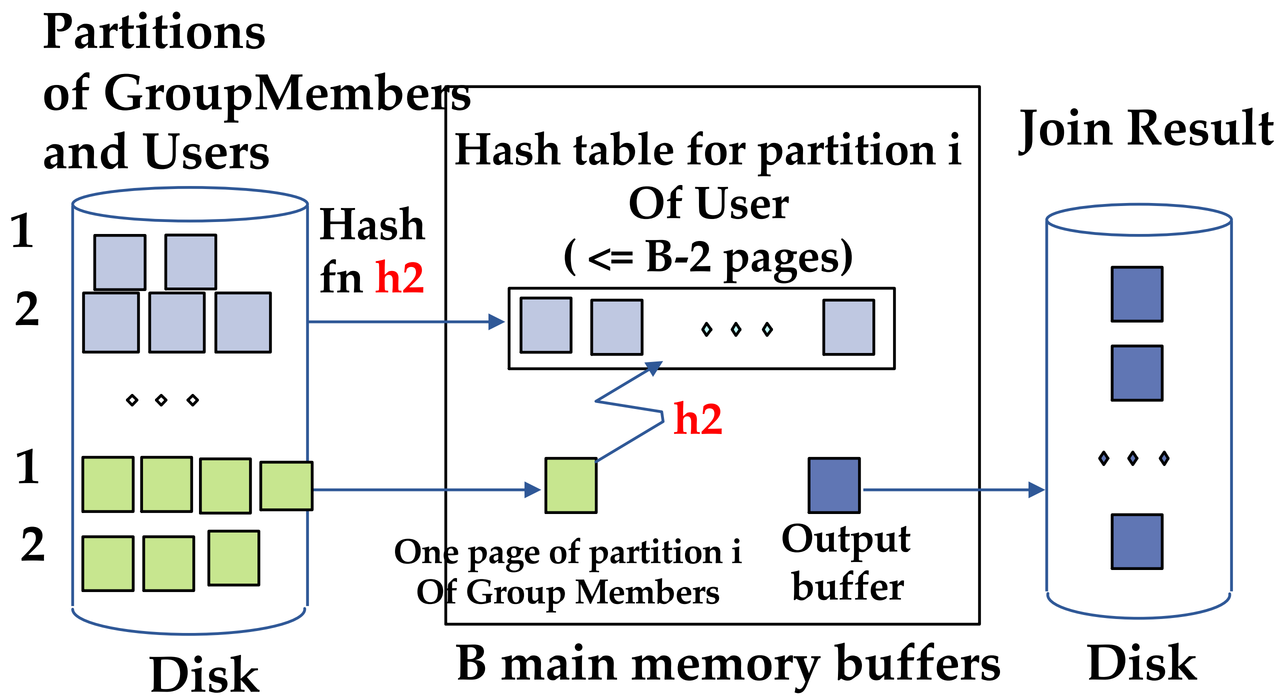
Hash Join: Costs

- Step I:
  - Read and write Users (partition U)
  - Read and write GroupMembers (partition GM)
  - 2 * UserPages + 2 * GroupMemberPages = 1000 + 2000
- Step II:
  - Read Users (partition by partition)
  - Read GroupMembers (page by page in partition order)
  - UserPages + GroupPages = 500 + 1000
- 3 * Userpages + 3 * GroupPages

Merge-Join vs. Hash-Join

- In our example:
  - Both 3*UserPages + 3*GroupPages
- Hash-Join better
  - if one relation really large
  - (sort in merge-join might need another pass)
- Merge Join better
  - if hash partitions of smaller don't fit into main memory
- Merge Join result is sorted
- Hash join good for parallelization
  - Let each pair of partitions be matched on a different node

Hash-join: CPU optimized



- Reorganize partition i of User
  - Partition according to a second hash-function
  - h2(u.uid) = j -> u is put on page j
- Match tuples of GroupMembers
  - h2(g.uid) =j -> matching u tuple must be on page j
- Avoids scanning the entire partition of U for each tuple of Group Members

FINAL

The Projection Operation

```
SELECT   GM.uid, GM.gid
FROM     GroupMembers GM
```

- Usually done on the fly together with another operation (or pipelined)
  - For instance, while reading in the transaction for a sort, or join etc.
- More complex: SELECT DISTINCT name FROM …
  - Expensive operation
  - Requires sort in order to eliminate duplicates!!
  - Often done at the very end (less tuples) or whenever the relation is sorted for some other reason
  - Database user: use DISTINCT only when really necessary

Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except similar;
- For instance: sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.

Sort → scan → merge

Aggregate Operations (AVG, MIN, etc.)

- Usually done at the very last step after all selections/joins etc.
- Without grouping:
  - In general, requires scanning the relation.
- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation.)

Execution Plan

- A plan describes how a query is executed
- A Tree (sequence) of basic operators (select, join, project, sort, etc) used to process the query
- For each operator, an indication how it will be executed (index nested loop, sort, index, simple scan....)

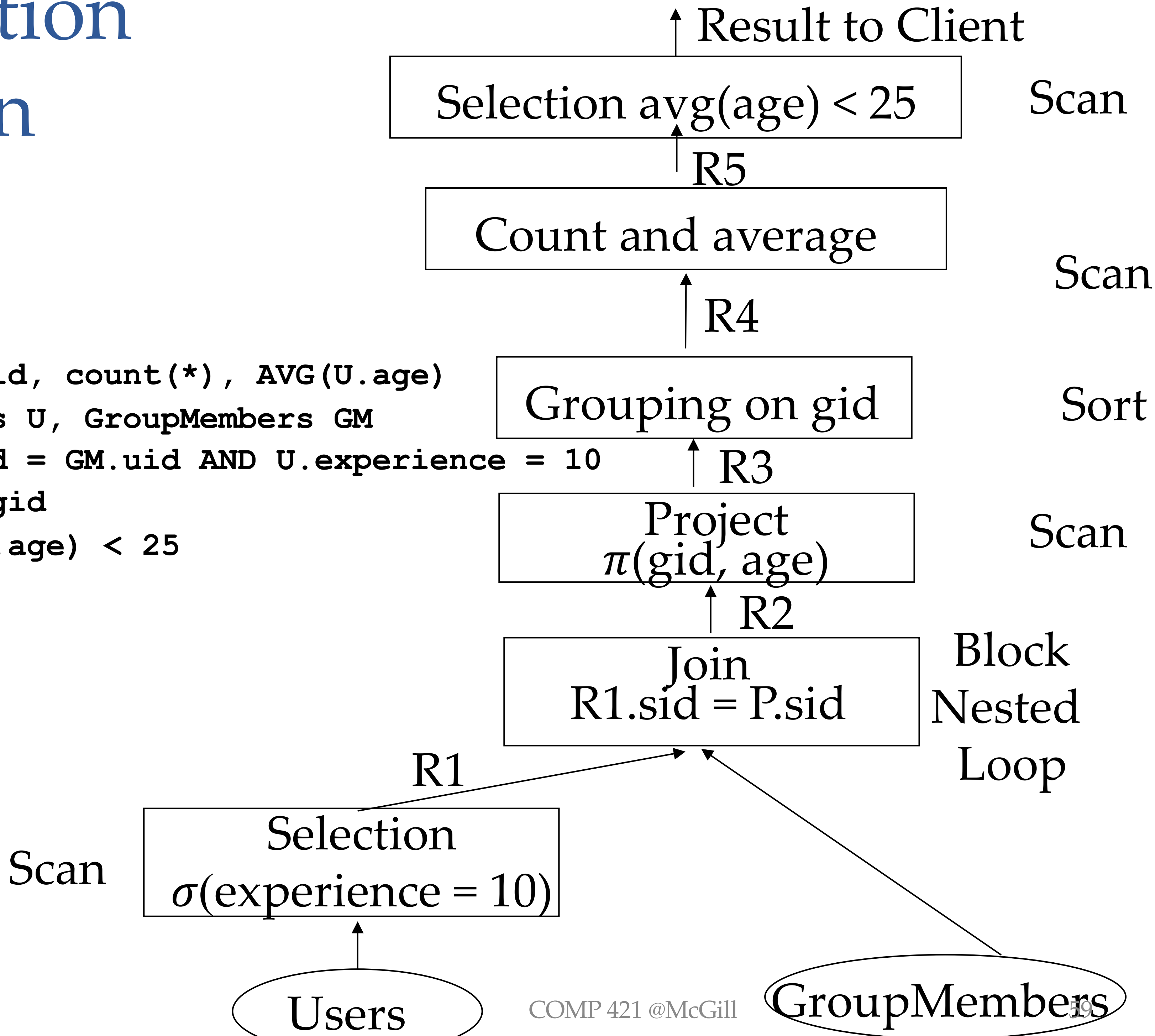
Example: Step by step

```
SELECT   GM.gid, count(*), AVG(U.age)
FROM     Users U, GroupMembers GM
WHERE    U.uid = GM.uid AND U.experience = 10
GROUP BY GM.gid
HAVING AVG(U.age) < 25
```

- Scan the Users table, determine all tuples with experience = 10 (scan or index)
  - Results in intermediate relation R1
- Join R1 and GroupMembers (several join options)
  - Results in intermediate relation R2
- Eliminate attributes other than gid and age.
  - Results in intermediate relation R3
- Group the tuples of R3 on gid (done by sort).
  - Results in intermediate relation R4
- Scan R4, count and perform average
  - Results in intermediate relation R5
- Return all tuples with avg(age) < 25.

Execution Plan

```
SELECT   GM.gid, count(*), AVG(U.age)
FROM     Users U, GroupMembers GM
WHERE    U.uid = GM.uid AND U.experience = 10
GROUP BY GM.gid
HAVING AVG(U.age) < 25
```

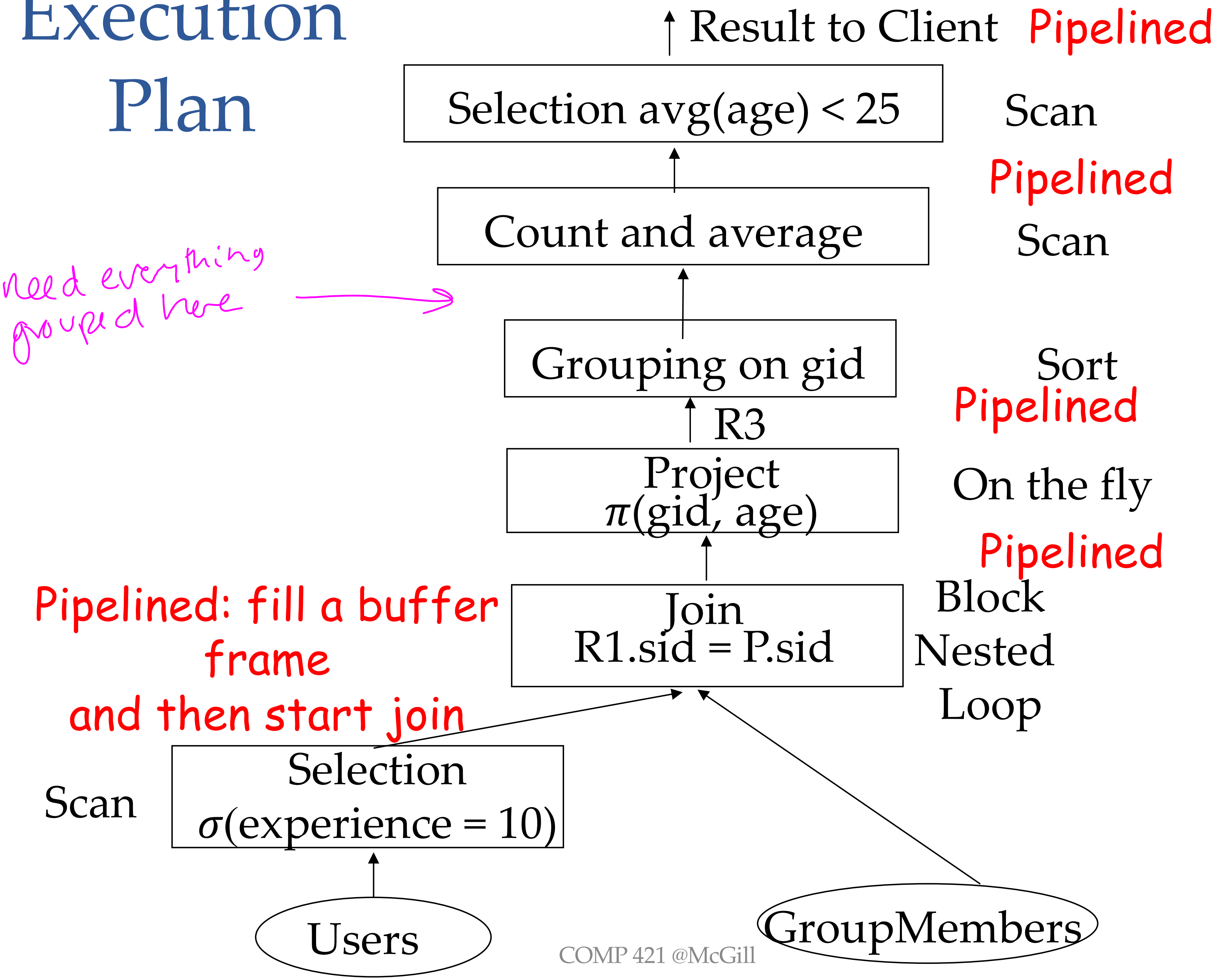


Pipelining

- Execution within one operator:
  - As soon as a tuple is determined it is forwarded to next operator
- Parallel execution of operators
- No materialization of intermediate relations if it can be avoided
- Iterator operator uses pipelining

  iterator based systems
  - Every operator is an iterator
  - Iterator provides the following interface to “parent” operators: **open, getNext, close**
  - Iterator calls interface methods of “children” operators

Execution Plan



Equality Join with further restrictions

```
SELECT  *
FROM    Users U, GroupMembers GM
WHERE   U.uid = GM.uid
AND     experience = 10
```

making the outer relation as small as possible is good

- (assume 1% have experience = 10)
- Do selection before join;
- Stream (pipeline) qualifying tuples into join
- Block nested loop (get qualifying tuples until block is full)
  - UserPages + UserPages*0.01/(B-2) *GroupMemberPages = 500 + 1* 1000 = 1,500
- Index nested loop
  - UserPages + Card(Users)*0.01* Cost finding GroupMembers = 500 + 400*(1+2.5) = 500 + 1400 = 1900

Equality Join with further restrictions

```
SELECT  *
FROM    Users U, GroupMembers GM
WHERE   U.uid = GM.uid
AND     experience = 10
```

- Sort-Merge Join
  - Pipelining cannot be done as sort needed; therefore intermediate relation
  - But in particular case: result of selection fits into main memory; therefore sort of user tuples in main memory
  - Cost: UserPages + 3* GroupPages = 3500
- Hash Join
  - As Users after selection fits into main memory, it becomes kind of a hash join with one partition.
  - Thus, I don't need to partition GroupMembers, and hash join becomes identifical to Block Nested Loop
  - 500 + 1000

Optimization Techniques

Two types

- Algebraic optimization:
  - Use simple rules to perform those operations first that eliminate a lot of tuples
    - Push down selections and projections
  - build a basic operator tree
  - Do not yet consider HOW to execute each operator, implementations
  - Consider the number of tuples that flow from one operator to the next
  - Key issues: statistics
- Cost-based optimizations brute force
  - Consider a set of alternative plans created by algebraic optimization
  - Consider for each operator how it could be executed
    - Key issues: available indexes, available operator implementations

execute operation in an order such that most appropriate access paths (indexes) can be used

Algebraic

Projection, Selection and Join

- Pushing down selections and projections to the relations to which selection refers
- Careful with project

```
SELECT u.uname, u.experience
FROM   Users u, GroupMembers g
WHERE  u.uid = g.uid
AND    g.gid = 'G1'
AND    u.age > 50
```

- $\pi_{uname,experience}(\sigma_{gid=G1 \wedge age>50}(Users \bowtie GroupMembers))$
  - first join, then selection, then project
- $\pi_{uname,experience}(\sigma_{age>50}(Users) \bowtie \sigma_{gid=G1}(GroupMembers))$
  - push down selections
- $\pi_{uname,experience}(\sigma_{age>50}(Users) \bowtie \pi_{uid}(\sigma_{gid=G1}(GroupMembers)))$
  - push down SOME project

Push Projections

- Pushing down projections will not reduce the number of tuples but the SIZE of the intermediate results
- Be careful not to loose attributes that you need later on
- $\pi_{uname}(\text{Users} \bowtie \text{GroupMembers}) \neq \pi_{uname}(\text{Users}) \bowtie \text{GroupMembers}$

Algebraic Optimization (contd)

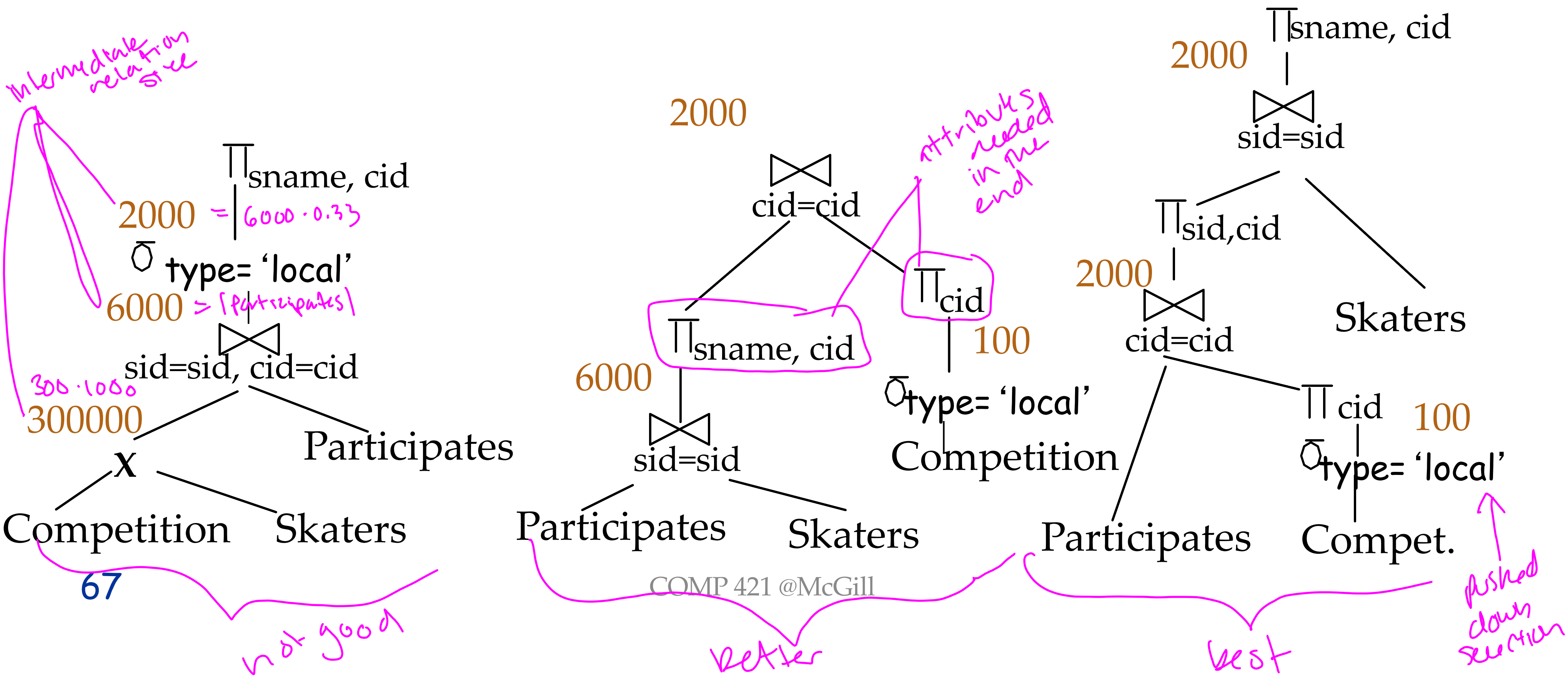
```
SELECT S.sname, P.cid
FROM Skaters S, Participates P, Competitions C
WHERE P.sid=S.sid AND P.cid=C.cid  AND C.type= 'local' ;
```

Skaters (<u>sid</u>: 4 Byte, uname (10 Byte), age (4Byte)): 1000 tuples

Participates (<u>sid</u>: 4 Byte, <u>cid</u>: 4 Byte, rank (4 Byte): 6000 tuples

Competition (<u>cid</u>: 4 Byte, location (10 Byte), type (5 Byte): 300 tuples

type attribute: 3 different values   reduction factor: 33%



Cost Based Optimization

- Find a plan with low cost
- Dynamic programming in bottom-up fashion for deep join plans :
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. *(All 2-relation plans.)* consider inner and outer
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. *(All N-relation plans.)*
  - Prune high-costs alternatives
  - Retain alternatives with interesting features
    - Cheapest plan overall, plus
    - Cheapest plan for each *interesting order* of the tuples.
  - Bottom-up calculation each possible execution plan, estimate the cost
  - In class: something between algebraic and cost-based optimization

example in my courses slides

Nested Queries

- Select S.sname
  From Skaters S
  Where S.sid IN (Select P.sid From Participates P
  Where P.cid = 103)
  - ☆ Execute subquery first; returns intermediate relation, with distinct sid
  - ☆ Join outer relation and intermediate relation with any join method
- Select S.sname
  From Skaters S
  Where exists (Select * from Participates P
  where P.cid = 103 and P.sid = S.sid)
  - ☆ Have to execute inner query for each outer tuple; little optimization possible
- Select S.sname
  From Skaters S, Participates P
  Where S.sid = P.sid AND P.cid = 103
  - ☆ all optimizations possible
- smart rewriting system able to transform queries automatically

database transforms to this

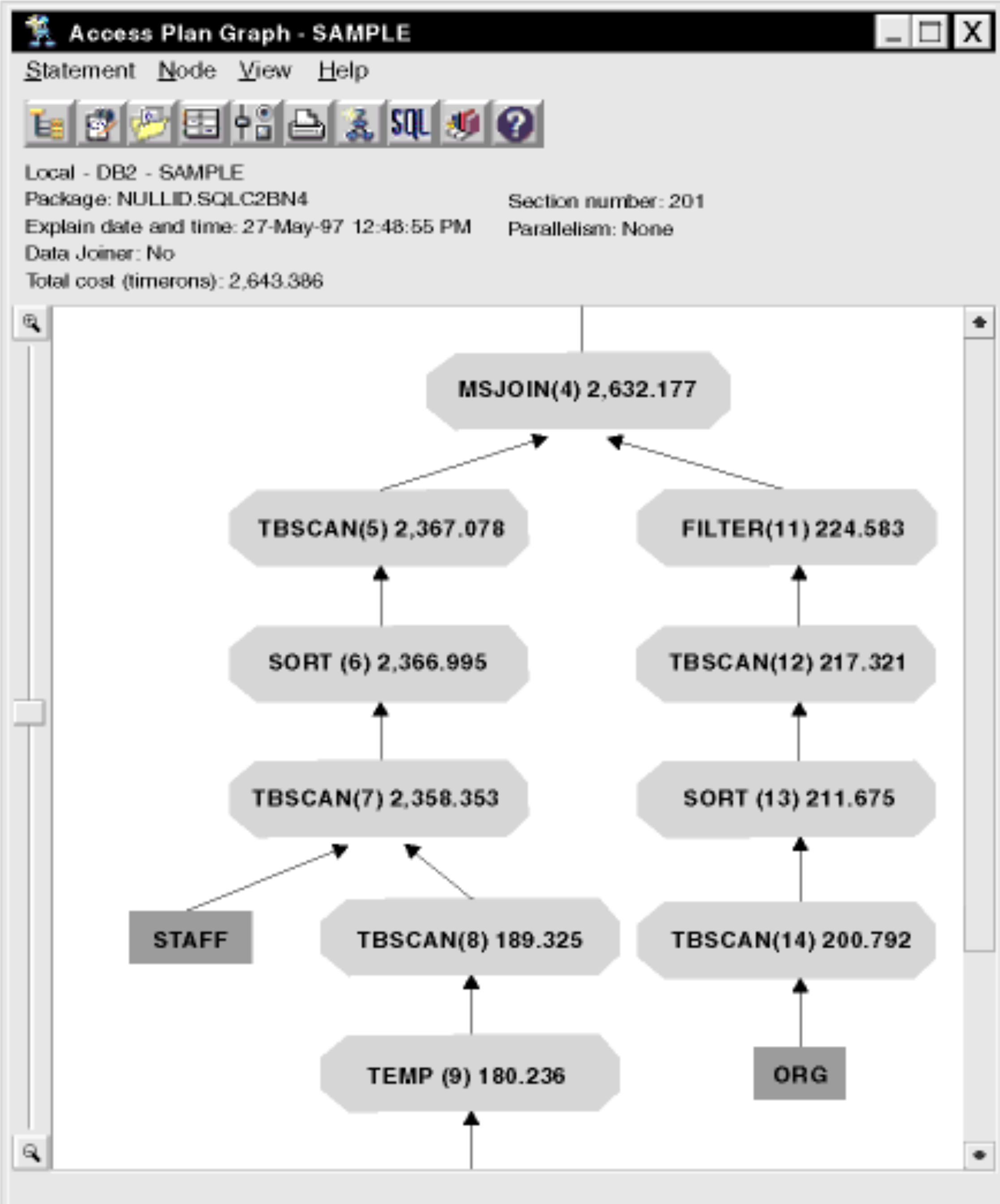
why did we learn all this

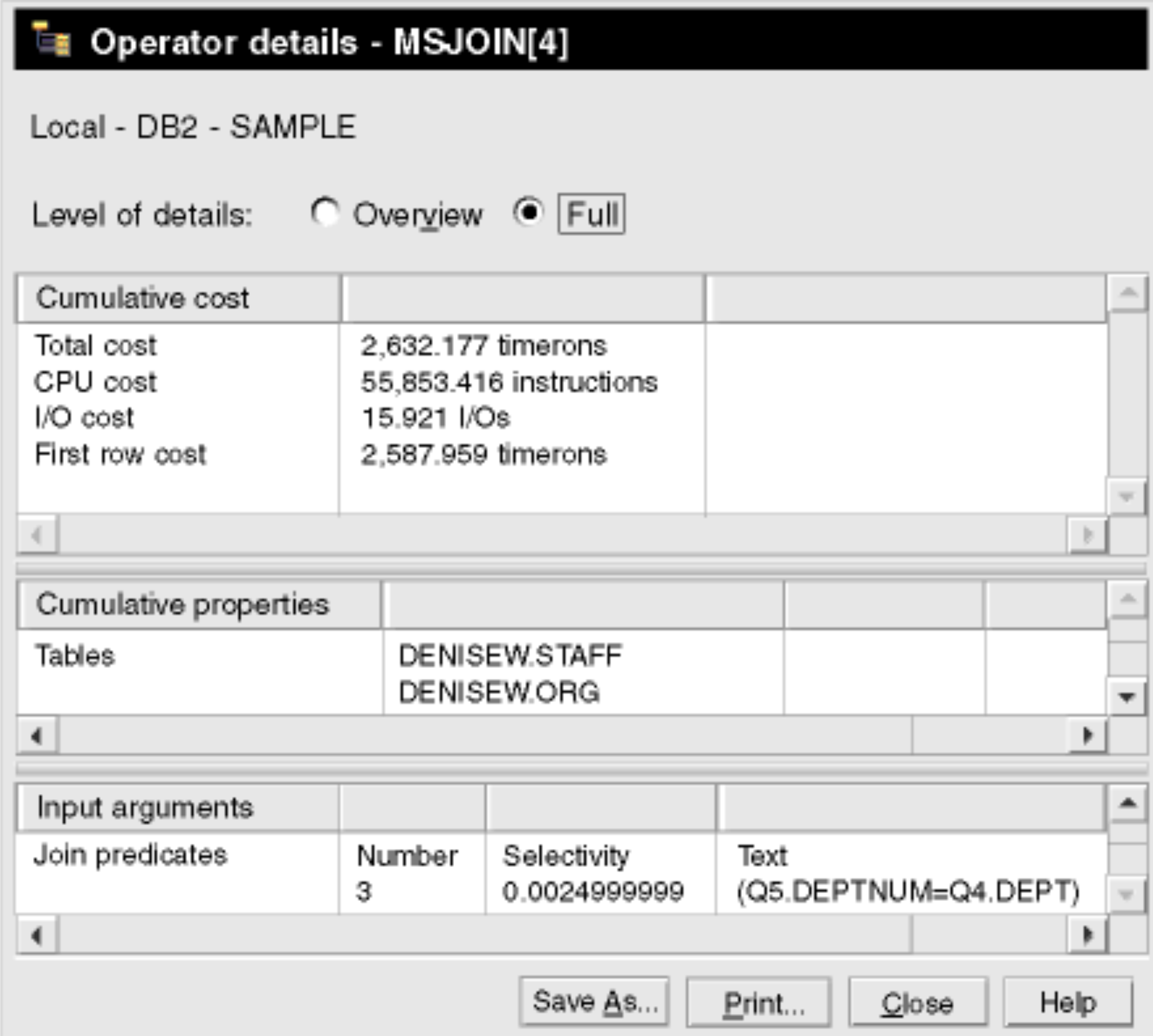
System Tuning

- If your application has some standard, well-known queries:
  - Create appropriate indices to speed up these queries
- If your application has many many updates and inserts
  - Be careful with creating indices
  - Each INSERT will insert new value in each tuple
  - Updates change some indices
- SQL Explain explains how a query is executed internally

Statistics in DB2

- Start Runstat to collect information about tables :
  - ☆SYSCAT.TABLES and SYSSTAT.TABLES:
    - Number of pages per table, number of tuples, etc.
  - ☆SYSCAT.INDEXES and SYSSTAT.INDEXES
    - Number of index leaf pages, index levels, degree of clustering, number of distinct values in first column, page fetch estimate for different buffer size, etc.


Access Plan Graph - SAMPLE
Statement Node View Help
Local - DB2 - SAMPLE
Package: NULLID.SQLC2BN4
Section number: 201
Explain date and time: 27-May-97 12:48:55 PM
Parallelism: None
Data Joiner: No
Total cost (timerons): 2,643.386
MSJOIN(4) 2,632.177
TBSCAN(5) 2,367.078
FILTER(11) 224.583
SORT (6) 2,366.995
TBSCAN(12) 217.321
TBSCAN(7) 2,358.353
SORT (13) 211.675
STAFF
TBSCAN(8) 189.325
TBSCAN(14) 200.792
TEMP (9) 180.236
ORG


Operator details - MSJOIN[4]
Local - DB2 - SAMPLE
Level of details: Overview Full
Cumulative cost
Total cost 2,632.177 timerons
CPU cost 55,853.416 instructions
I/O cost 15.921 I/Os
First row cost 2,587.959 timerons
Cumulative properties
Tables DENISEW.STAFF
DENISEW.ORG
Input arguments
Join predicates
Number 3
Selectivity 0.0024999999
Text (Q5.DEPTNUM=Q4.DEPT)
Save As...
Print...
Close
Help