# CS 5/7330

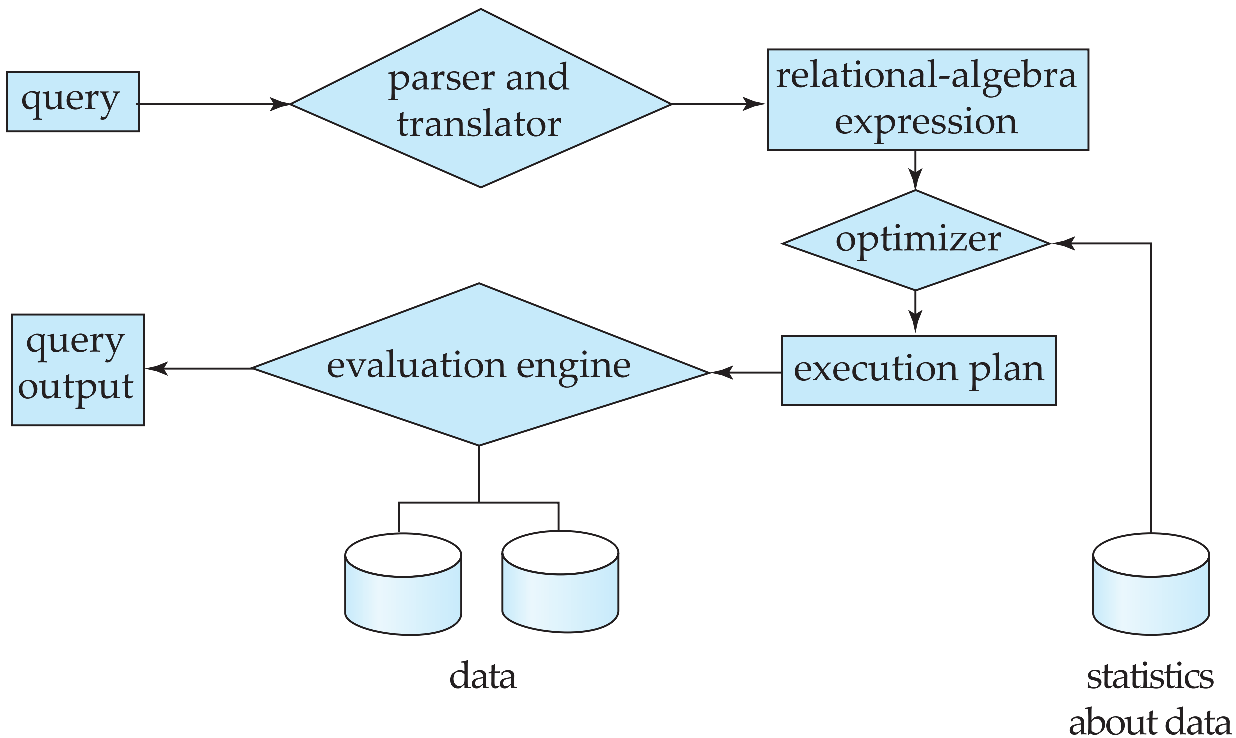
Query Processing / Optimization

# Query Processing / Optimization

- Study how database process query internally
  - i.e. Once a database system received an SQL query, what happens (until the database return the results)
- Why study?
  - Database project manager – understand whether the queries being written will likely be executed effectively by the database
  - Database administrator – able to restructure database / provide hints to the database system to speed up queries
  - Database developer – you may be hired by a database systems company to build the next version of a DBMS

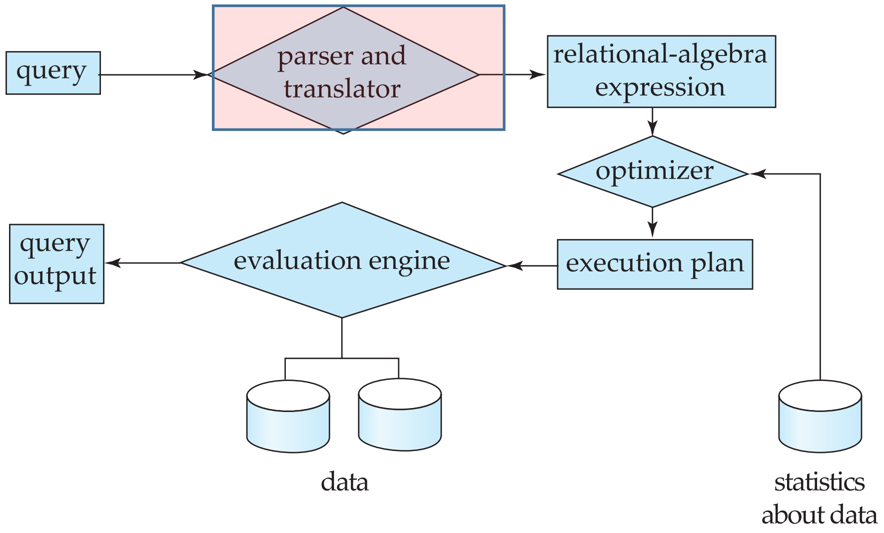
# Query Processing

- What happen when a DBMS received a query



# Query Processing

- Parser and translator
- Read the query
- Check for syntax
- Collect all relevant information about tables involved
- Break the query down into a set of basic operations (relational algebra + others)



# Query Processing

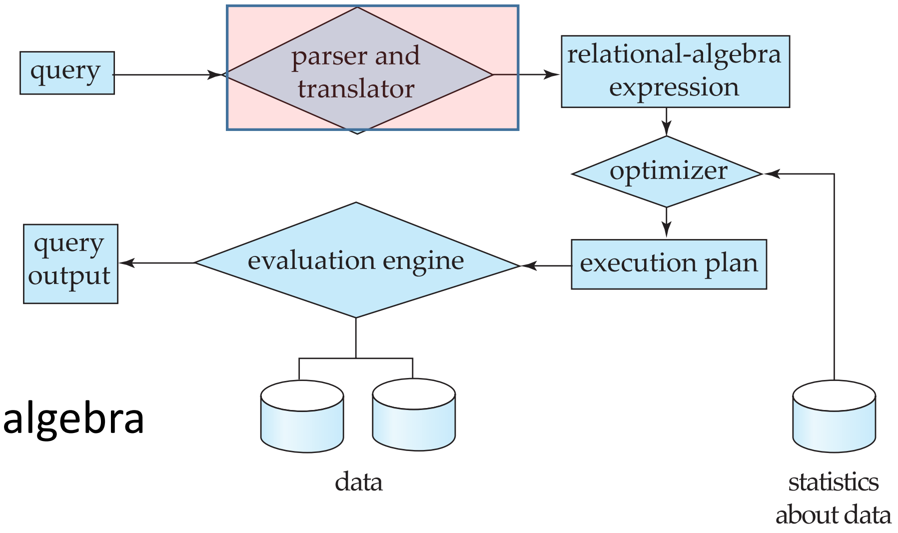
- Recall

Projection in relational algebra

**select** $A_1, A_2, \ldots, A_n$
**from** $r_1, r_2, \ldots, r_m$
**where** $P$

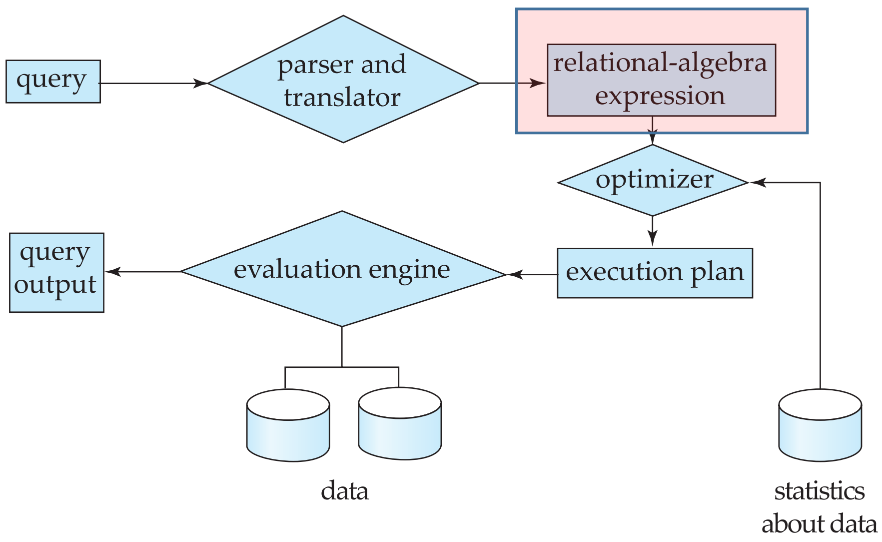
Cartesian Product in relational algebra

Selection in relational algebra (remember, join = Cartesian product + selection)



# Query Processing

- Query is converted to a list of operators ($\sigma$, $\prod$, $\bowtie$,Group, Order ….)
- Thus running the query becomes execution of a list of operators



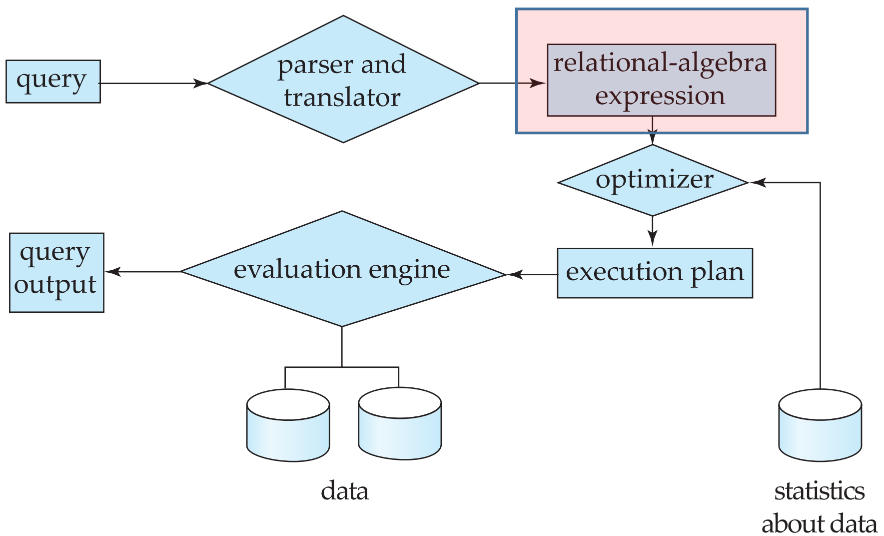
# Query Processing

- Example

SELECT ssn, age
FROM Instructor
WHERE age > 25

Becomes

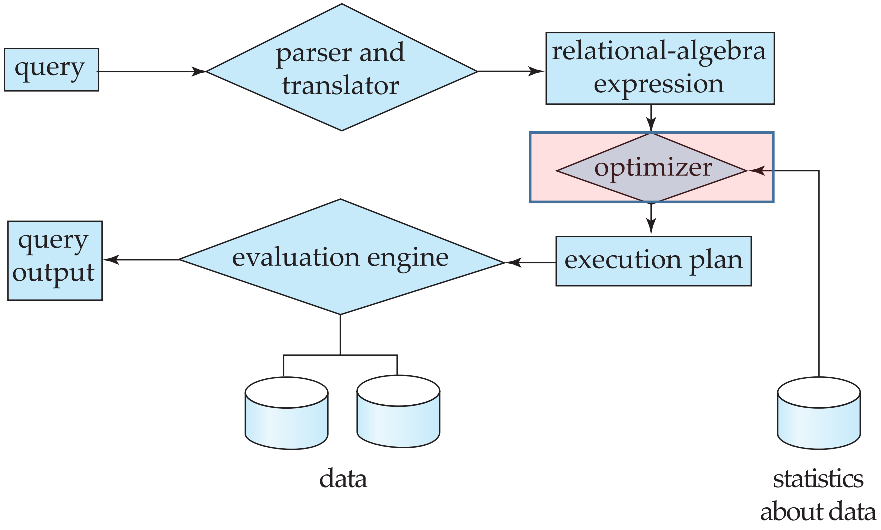
1. $A = \sigma_{age > 25}$ (Instructor)
2. Result = $\prod_{ssn,\ age}(A)$



# Query Processing

Optimizer

- For each operation, determine how it will be executed
- Determine the order of operations
- Other tasks (to be discussed later)

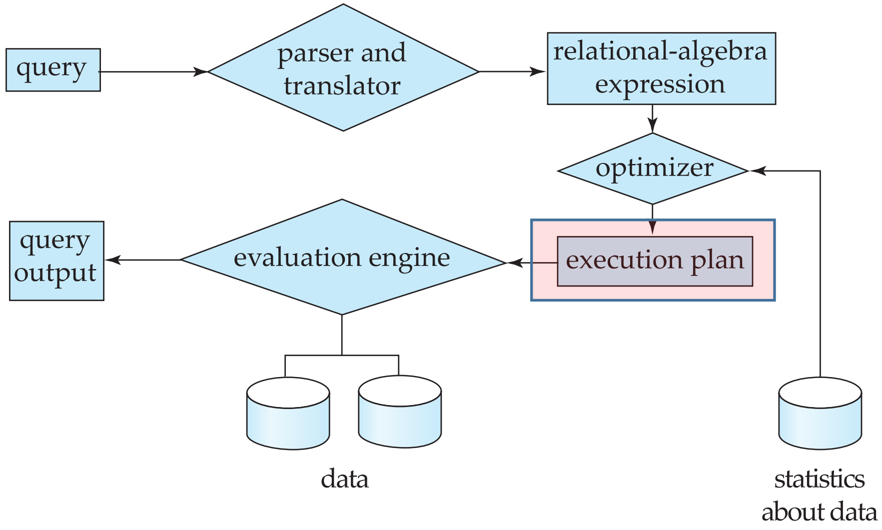


# Query Processing

Execution plan

- Example:

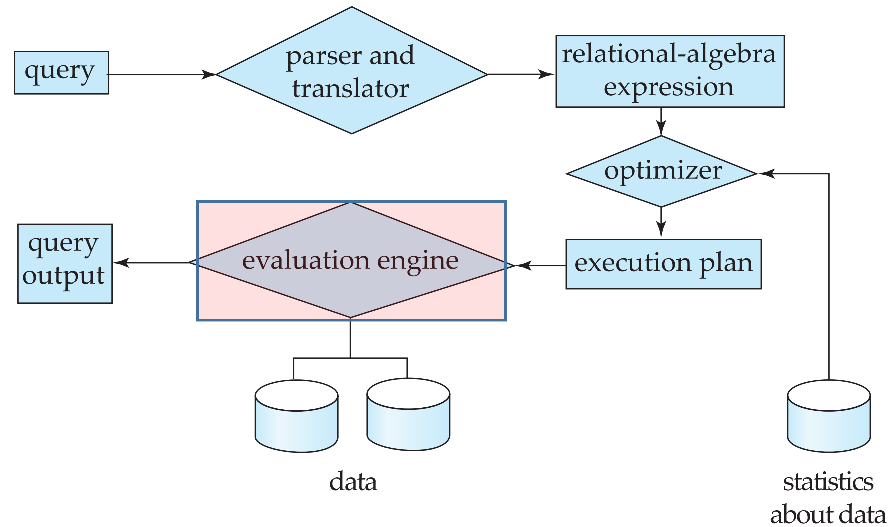
1. A = $\sigma_{age > 25}$ (Instructor)
    - Use the secondary index on A to retrieve the tuples
    - Do not store A on to the disk
2. Result = $\prod_{ssn,\ age}$ (A)
    - Directly pipe the result from 1 to execute the project
    - Just pick the attributes and output them



# Query Processing

Evaluation engine

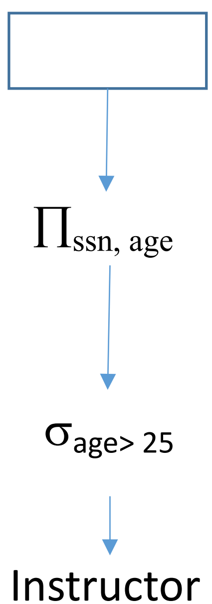
- Actual execute the plan
- Return the tuples to the output
  - Can be on screen, as a file, or as a stream through a network



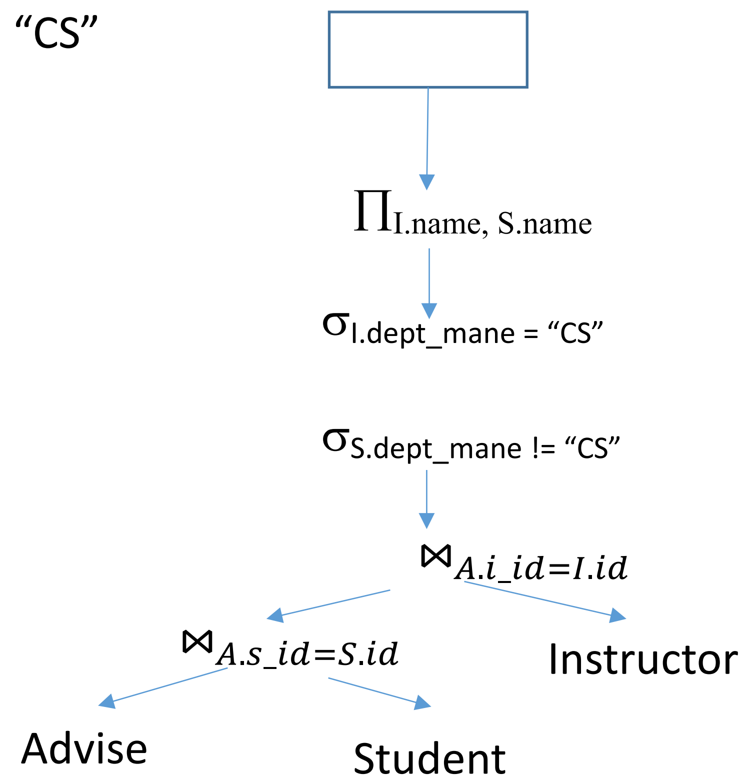
# Query Processing

- Notice that more often, a query can be represented as a parse tree:

SELECT ssn, age
FROM Instructor
WHERE age > 25




SELECT I.name, S.name
FROM Instructor I, Advise A, Student S
WHERE A.i_id = I.ID AND S.id = A.s_id
AND I.dept_name = “CS” AND
S.dept_name != “CS”



# Query Processing

- Query Processing/Optimization in terms of parse trees:
  - Parsing: Generate parse tree for the query
    - Notice that a query may generate MANY (TOO MANY) valid parse trees
    - Each node correspond to an operation
    - Evaluation of query is from bottom to top (think post order)
  - Optimization:
    - Decorating the nodes of a parse
      - For each node, determine how that operation is to be executed
    - Select the best parse tree
      - For all the parse tree generated, pick the one that will execute the query fastest

# Query Processing / Optimization

- Challenges
  - Combinatorics explosion
    - There are many valid parse tree for a query
    - There are many possible ways to decorate each node
    - The total number of options grows exponentially (since most combinations of parse tree and decorations are valid)
  - Limited time
    - You do not want to spend 2 hours optimizing a query such that you save 2 seconds.
  - Information needed
    - A lot of information is needed for the optimizer to work properly
    - Recall the fundamental problem of query optimization?

# Measuring cost of a query

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk

# Measuring cost of a query

- Disk cost can be estimated as:
  - Number of seeks/rotates * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - $t_T$ – time to transfer one block
    - Assuming for simplicity that write cost is same as read cost
  - $t_S$ – time to move from one block to a non consecutive block
    - If on the same track, then rotate
    - If on different track, then seek + rotate
  - Cost for b block transfers plus S seeks
    $b * t_T + S * t_S$
- $t_S$ and $t_T$ depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk: $t_S$ = 4 msec and $t_T$ =0.1 msec
  - SSD: $t_S$ = 20-90 microsec and $t_T$ = 2-10 microsec for 4KB

# Measuring cost of a query

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice

# Query processing for individual operations

- Selection
- Joins
- Projection
- Ordering
- Group By

# Selection

- $\sigma_{\text{condition}}$ (Table)
- A few considerations
  - Condition
    - Can be “attribute = value” (equality)
    - Or “attribute <= value” (or <, >, >=) (range/comparison)
    - Single condition or multiple condition
      - E.g. (x = 1 AND y = 3), (x = 3 OR y = 4)
  - Attribute
    - Primary key  or not
  - Organization
    - Index available?
    - What kind of index?

# Selection

- Basic case: File scan / sequential scan
- Just read the whole file block-by-block from beginning to end and check if each tuple satisfy conditions
  - Cost estimate = $b_r * t_T + b_s * t_S$
    - $b_r$ denotes number of blocks containing records from relation *r*
    - $B_s$ denotes number of tracks that store the table
  - If selection is on a key attribute, can stop on finding record
    - Average cost = $(b_r/2) * t_T + ? * t_S$
    - ? Is harder to predict, is roughly max(1, $b_s / 2$)
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- Note: binary search generally does not make even if data is sorted except when there is an index available, as each step will require a rotation

# Selection

- If you have an index: Index scan
  - Use an index to search.
  - Attribute of index needs to match condition
  - Also hash table is not useful for range queries
- Assume that the query return t tuples, stored in $b_r$ blocks (notice that t > $b_r$ – often by at least one order of magnitude)
  - t = 1 if equality search on an unique attribute
- Clustering index:
  - Cost = time for searching the index + $b_r * t_T + \alpha * t_S$
    - α is the number of tracks that contains retrieved tuples (if t = 1, α = 1)
    - α grow slowly (if at all) with t (since data is clustered)

# Selection

- If you have an index: Index scan
  - Use an index to search.
  - Attribute of index needs to match condition
- Assume that the query return t tuples, stored in $b_r$ blocks (notice that t > $b_r$ – often by at least one order of magnitude)
  - t = 1 if equality search on an unique attribute
- Non-Clustering index:
  - Worst case Cost = time for searching the index + $t$ * ($t_T$ + * $t_S$)
    - It may be faster if the DBMS read all the index record and determine which tracks the data are stored before fetching the record
    - Even this can be infeasible
      - For example, if there is an ORDER BY clause and there are too many records being selected to fit in main memory
        - Cannot bring them all in to sort

# Selection

- Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
  - Now multiple indices may be available
  - Option 1: Single index
    - Use one of the available index
    - Read in tuples into main memory, then apply other conditions
    - Some cost estimation as before, only with t (and br) are tuples/blocks that contain tuples that satisfy the condition that the index is correspond to
  - Option 2: Clustering multiple-attribute index (if available)
  - Option 3: Intersection of identifiers
    - Consider all secondary indices that associate with an attribute in the condition
    - Query the index, read in all index records into main memory (do not go to database yet)
    - Only select pages that are in the answers for all indices
    - Read the tuples and subsequently apply other conditions

# Selection

- Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
  - Now multiple indices may be available
  - Option 1: Intersection of identifiers
    - Assume indices present for ALL attributes in the condition
    - Query the indices, read in all index records into main memory (do not go to database yet)
    - Only select pages that are in at least on of the indices
    - Usually is expensive
    - Notice that it doesn’t work if indices for some attributes in the condition is not available
  - Option 2: Sequential scan, apply condition when tuple is read

# Joins

- $R \bowtie_{\text{cond}} S$
- Most common case: condition is an equality condition between attributes of R and S
    - We call this equi-join
    - Example: linking an attribute with its foreign key
- With equi-join there are a lot of options
- With non equi-join there are very few

# Joins – Nested loop

- The naïve algorithm
- $R \bowtie_{\text{cond}} S$
- **for each** tuple $t_r$ **in** *R* **do begin**

  **for each tuple** $t_s$ **in** *S* **do begin**

  test pair $(t_r, t_s)$ to see if they satisfy the join condition *cond*
  if they do, add $t_r \bullet t_s$ to the result.

  **end**

  **end**
- R is called the outer relation (outer loop), S is the inner relation (inner loop)
  - Either relation can be in the outer loop
  - Flipping R and S will give you the same results
- Works for any condition

# Joins – Blocked Nested loop

- Modify the algorithm for secondary storage
- **for each block** $b_r$ **in** $R$ **do begin**

  Read $b_r$ from disk into main memory

  **for each block** $b_s$ **in** $S$ **do begin**

  Read $b_s$ from disk into main memory

  do a nested loop for each pair $(t_r, t_s)$ [$t_r \in b_r$ **,** $t_s \in b_s$ ) to see if they satisfy the join condition *cond*

  if they do, add $t_r \bullet t_s$ to the result.

  **end**

  **end**

# Joins – Blocked Nested loop

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 1: Buffer large enough to hold all blocks for both tables
  - Read both tables into main memory and then loop all the tuples inside
  - Cost = cost for sequential scan for R  + cost for sequential scan for S

# Joins – Blocked Nested loop

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 2: Minimum number of buffers (ctd)
  - Separate cost into page access and seeks
  - For page access, each block in the outer loop need to be read once
  - For inner loop, each block has to be read once for each block of the outer loop
  - Number of block access = $b_r + b_r * b_s$
  - Question: given R and S, which table should be in the outer loop

# Joins – Blocked Nested loop

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 2: Minimum number of buffers
  - For seeks/rotate
  - Each page in the outerloop need to be seeked
  - Assume query is not being interrupted, the inner loop is being read in consecutively
    - So only the minimum number of seeks
  - Number of seeks = $b_r$ * (number of seeks to read S)
    - However, it is more than likely that there will be interruptions
    - For example: the disk head may have moved pass while the joining is in process
    - Worst case scenario: $b_r * b_s$
  - Which table should be in the outerloop?

# Joins – Blocked Nested loop

- Case 3: Enough buffer to fit either R and S (plus k buffers for the other table), but not both
  - (Assume the buffer fits table S)
  - Read S into main memory
  - Then read R into main memory (in steps, because not enough memory to read it all at once)
  - Join pairs of tuples in R and S
  - Cost = cost of reading S (sequential scan) + cost of reading R
    - Depends on whether the query get interrupted, it can be as little as the same as sequential scan or as much as ceiling($b_r$ / k), where k is the number of buffers allocated to R

# Joins – Blocked Nested loop

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 4: None of the above (k buffers available between the two tables)
  - We will need to assign buffers to each table
  - Assume we assume k' buffers to R
    - Then k-k' buffers to S
    - **for each** k' **blocks in** *R* **do begin**

      Read k' blocks of R from disk into main memory

      **for each (k-k') block** $b_s$ **in** *S* **do begin**

      Read (k-k') blocks of S from disk into main memory

      do a nested loop for each pair $(t_r, t_s)$ $[t_r \in b_r$ **,** $t_s \in b_s)$ to see if they satisfy the join condition *cond*

      if they do, add $t_r \bullet t_s$ to the result.

# Joins – Blocked Nested loop

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 4: None of the above (k buffers available between the two tables)
  - Cost: Consider number of blocks read
    - Outer loop: every block in R need to be read once – cost = $b_r$
    - Inner loop: for each iteration of the outer loop, the whole table in the inner loop need to be read
    - The outer loop executed ceiling($b_r / k'$) times
    - So the total number of block reads for inner loop = $b_r * b_s / k'$
  - Given that, what should be the value of k'?

# Joins – Blocked Nested loop

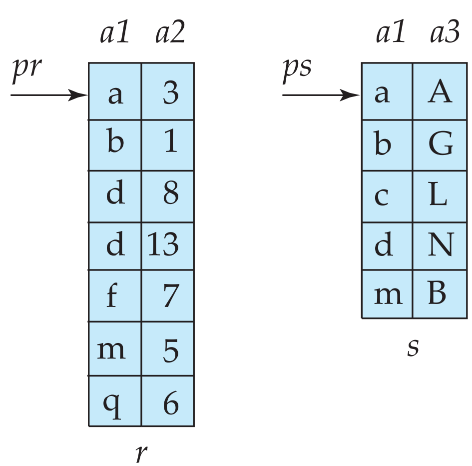
- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 4: None of the above (k buffers available between the two tables)
  - Cost: Consider number of seeks
    - Outer loop: every time the outerloop executes there need to be a seek (why? – similar to case 2)
      - Number of seeks = ceiling($b_r$ */ k')*
    - Inner loop: Similar to case 2
      - Best case scenario = $b_r$ ** (number of seeks for sequential read S) / k'*
      - Worst case = $b_r$ * $b_s$ / (k' * (k – k'))
  - Now what should the value of k' be?

# Joins – Blocked Nested loop

- Should we use an index?
- Index for outer loop is useless
  - Unless clustering index on join attribute where you can read the table based on the join attribute, and that attribute is not unique (why?)
- How about inner loop
  - To use an index, that means for each ***tuple*** there need to be  a search
    - Number of tuples is larger than number of blocks
    - If each tuples only match with very few tuples, it's fine
    - However, if there are potential large number of matches with a secondary index, things can get dicey…. (exercise)

# Joins – Sort-merge

- Consider an equi-join: $R \bowtie_{R.a = S.b} S$
- Assume
  - R is a sequential file ordered by attribute a
  - S is s sequential file ordered by attribute b
- Now to join the two tables
  - We can use the merge algorithm from merge sort



# Joins – Sort-merge

- Key difference
  - If attributes have duplicate values then one may have to go “back and forth”
- Cost for the merge
  - Assume we have k’ buffers for R and k – k’ buffers for S
  - Blocks read = $b_r + b_s$ (best case, or unique value of attributes)
    - In general, may multiply by some factor m to take account for duplicates
  - Seeks = ceiling($b_r$ / k) + *ceiling*($b_s$ / (k – k’)) (why?)

# Joins – Sort-merge

- What if the tables are not sorted?
- Sort them (!) and store the sorted table temporarily
- Then apply the merge algorithm
- This is known as the sort-merge algorithm
- Cost = cost to sort the tables + merge cost (as previous slide)

# Joins – Sort-merge

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
- Assume we have 11 pages of buffers
- Consider sorting R
  - First step: Divide page into segments of 10 pages = 100 segments (to make analysis easier)
  - Merging step: assume merge 10 segment as a time
    - Two iterations: 100 segments, 10 page each -> 10 segments, 100 page each -> sorted, 1000 pages
  - Total number of read/writes = 3 (total iterations) * 2 (read/write) * 1000 = 60000

# Joins – Sort-merge

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
- Assume we have 11 pages of buffers
- Consider sorting S
  - First step: Divide page into segments of 10 pages = 10000 segments (similar as R)
  - Merging step: assume merge 10 segment as a time
    - Four iterations : 10000 segments, 10 page each -> 1000 segments, 100 page each -> 100 segments, 1000 page each -> 10 segments, 10000 pages -> sorted, 100000 pages
  - Total number of read/writes = 5 (total iterations) * 2 (read/write) * 100000 = 1000000

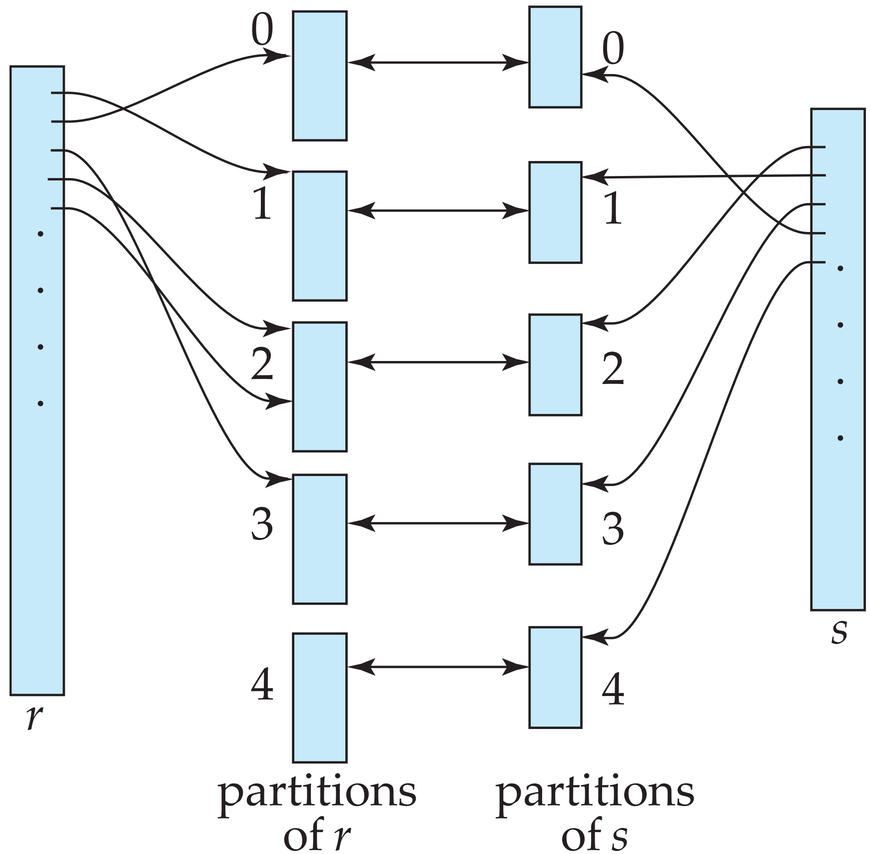
# Joins – Sort-merge

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
- Assume we have 11 pages of buffers
- Total cost = Cost of sorting R + Cost of sorting S + cost of merge
- = 60000 + 1000000 + (1000 + 100000) = 1161000 pages

# Joins – Hash joins

- $R \bowtie_{R.a = S.b} S$
- Build a hash table on the file for attribute a and b, using the same hash function for both tables
- Now only tuples in the same buckets in the corresponding table can join together

# Joins – Hash joins



# Join – Hash joins

- Now each pair of partition can be join by using any algorithm
  - If one of the partition is small enough to fit in the buffers, do a nested loop
  - Otherwise, one can recursively apply hash join – using a different hash function for each recursive call, until one of the partition is small enough

# Join – Hash joins

- Cost
  - Harder to estimate
  - Depends a lot on data distribution (as a skewed distribution will lead some segments very long)
  - However, potential advantages
    - As long as one table’s tuple behave nicely with the hash table, it should be fine
    - Even if distributions are skewed for both tables, it is possible that they complement each other (i.e. for a hash value, one table may have a lot of tuples, but the other may have little)

# Join – Hash joins

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
  - Assume we have 11 pages of buffers
  - Assume all hash function evenly distribute the tuples for both tables
  - First iteration:
    - R and S divided into 10 segement (100 page for R, 10000 page for S)
  - Second iteration
    - Each segment is further subdivided (10 page for R, 1000 page for S)
  - Now the segments for R is small enough to fit in main memory
    - Nested loop for each pair
- Running time = 4 * (100000 + 1000) + 100000 + 1000
- =505000
- Better than sort-merge

# Joins – hash-join vs. sort-merge

- Comparing sort-merge and hash-join
  - Hash-join is better than sort-merge when there is a large different between number of pages between the tables
    - The number of iteration is dominated by the larger table in sort-merge (need to completely sort both tables)
    - But is dominated by the smaller table in hash-join (need to recursive call until ONE of the segment is small enough)
  - Sort-merge has more predictable performance
    - Hash join's performance depend on how the hash function performs
  - Sort-merge has the output sorted by the join attribute
    - May be important (see later)

# Projection

- Seems straightforward – just picking the corresponding values out of the tuple
- But how about SELECT DISTINCT?
  - Need to remove duplicates
- Much trickier than you think
  - Data too large to fit in main memory…

# Projection

- Use external sort
  - At each step, eliminate duplicates for a segment before writing on the disk
- Hashing can be used instead

# Group by + Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization*:* **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the partial aggregates
    - For avg, keep sum and count, and divide sum by count at the end

# Union/Intersect/Except (set operations)

- **Set operations** ($\cup$, $\cap$ and $-$): can either use variant of sort-merge, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on $r_i$.
     2. Process $s_i$ as follows
        - $r \cup s$:
          1. Add tuples in $s_i$ to the hash index if they are not already in it.
          2. At end of $s_i$ add the tuples in the hash index to the result.

# Union/Intersect/Except (set operations)

- Set operations using hashing:
  1. as before partition *r* and *s*,
  2. as before, process each partition *i* as follows
     1. build a hash index on $r_i$
     2. Process $s_i$ as follows
        - $r \cap s$:
          1. output tuples in $s_i$ to the result if they are already there in the hash index
        - $r - s$:
          1. for each tuple in $s_i$, if it is there in the hash index, delete it from the index.
          2. At end of $s_i$ add remaining tuples in the hash index to the result.

        Difference in except: since operation is NOT communtative, restriction on which table can be "outer" loop.

# Outer joins

- Challenge: The unmatched tuples still need to be returned (with NULL value padded)
- Sort-merge:
  - Straight-forward: output also unmatched tuples during merge
- Hash-join:
  - Once again, restriction on what can be the “outer” loop