

Database Technology Query Processing



Today

- We're still opening the mysterious RDBMS black box
 - We can query a database
 - e.g., queries across multiple tables
- Today
 - How are those queries executed?
 - Which parts are evaluated first?
 - How are sorts carried out?
 - ...



Outline

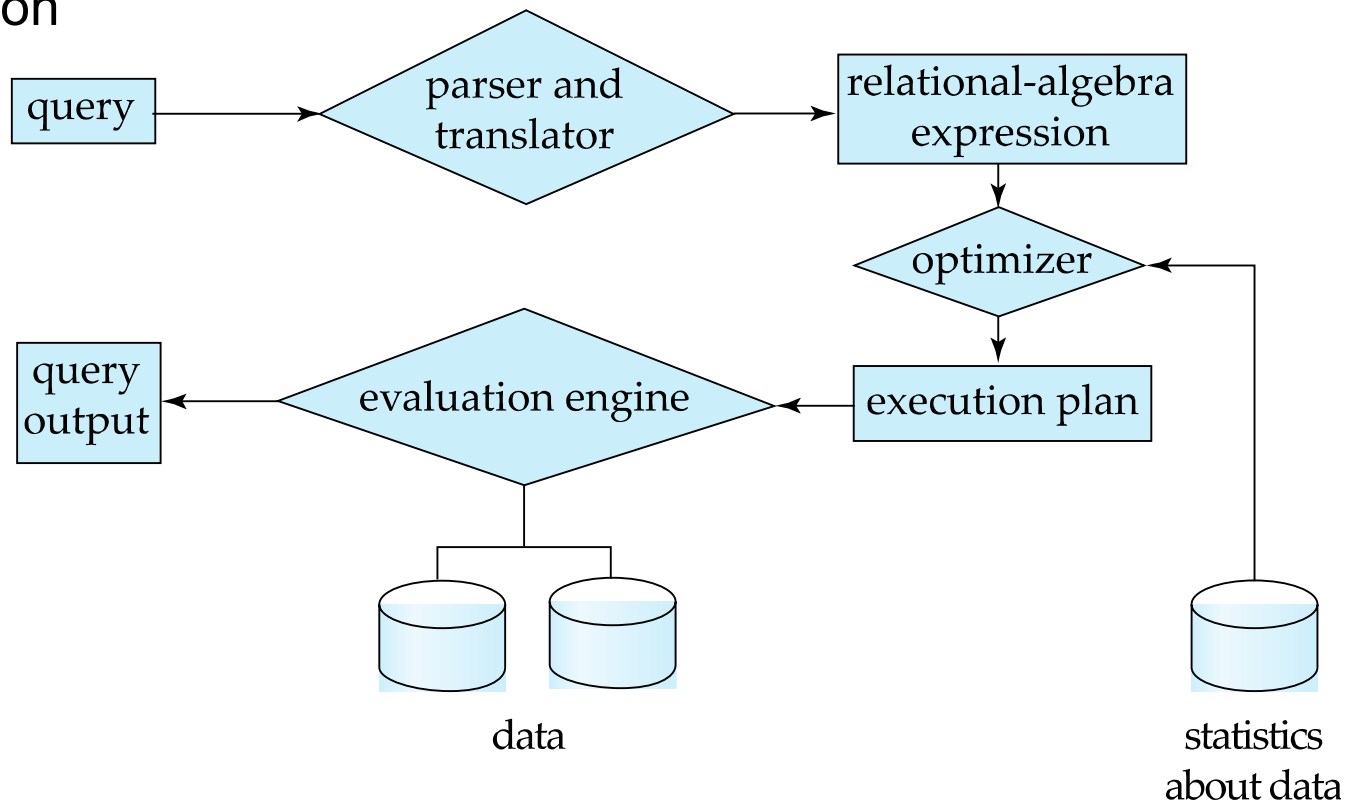
- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

Motivation

- Suppose you are a RDBMS
 - and you are asked to execute
SELECT name, building, salary
FROM instructor, building
WHERE instructor.dept_name = department.dept_name
AND salary > 75000
ORDER BY name
- How do you want to proceed?
 - Start with instructor or building relation?
 - Sort instructor by name table first, or filter by salary first?
 - ...

Basic Steps in Query Processing

- 1) Parsing and translation
- 2) Optimization
- 3) Evaluation



Basic Steps in Query Processing

- Parsing and translation
 - translate the query into its internal form
 - this is then translated into relational algebra
 - parser checks syntax, verifies relations
- Evaluation
 - The query execution engine takes a query evaluation plan,
 - executes that plan,
 - and returns the answers to the query

Basic Steps in Query Processing

- A relational algebra expression may have many equivalent expressions
 - e.g., $\sigma_{salary < 75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary < 75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
 - Correspondingly, a relational-algebra expression can be evaluated in many ways
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**
 - e.g., can use an index on *salary* to find instructors with $salary < 75000$,
 - or can perform complete relation scan and discard instructors with $salary \geq 75000$

Query Optimization

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost
 - Cost is estimated using statistical information from the database catalog
 - e.g. number of tuples in each relation, size of tuples, etc.
- Today's lecture:
 - How to measure query costs
 - Algorithms for evaluating relational algebra operations
 - How to combine algorithms for individual operations in order to evaluate a complete expression
- Next week's lecture
 - How to optimize queries,
 - i.e., how to find an evaluation plan with lowest estimated cost

Measuring Query Cost

- We want to execute the query as “cheap” as possible
- But what is “cheap”?
 - Execution time
 - Memory consumption
 - Electrical power consumption
 - ...
- Most approaches seek to minimize the *execution time*

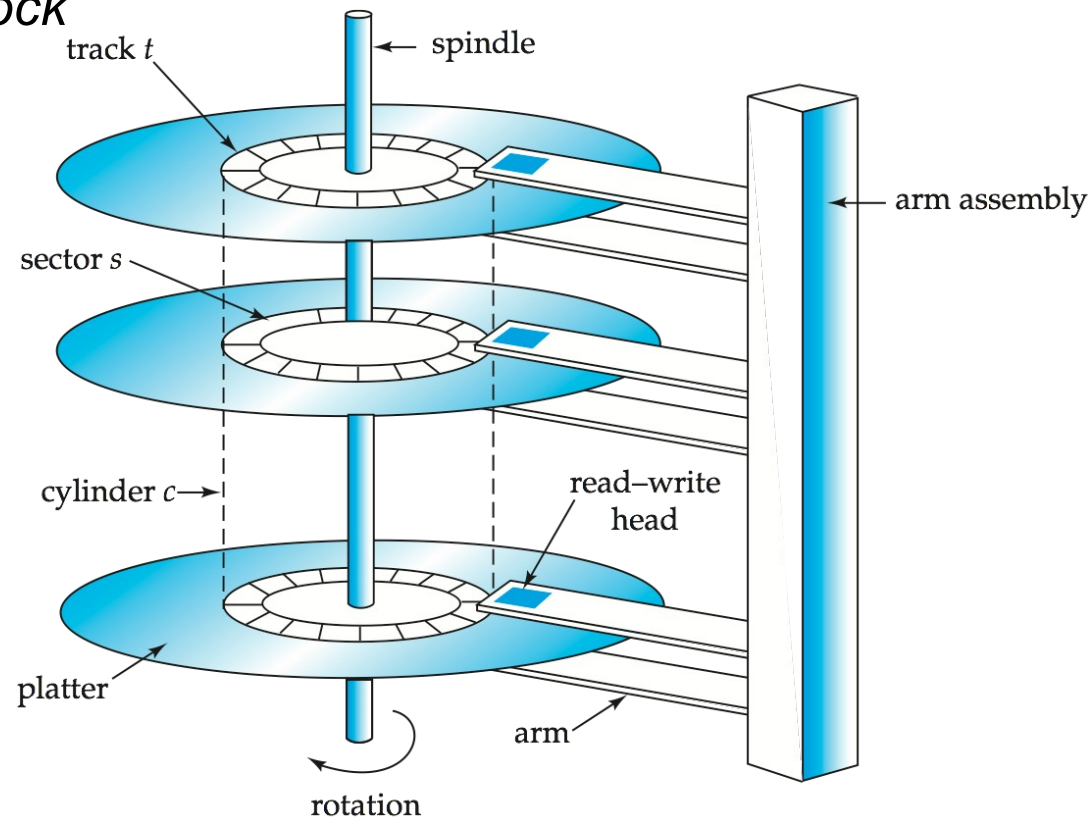


Measuring Query Cost

- Cost is generally measured as *total elapsed time* for answering query
- Many factors contribute to time cost
 - *disk accesses, CPU, or even network communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate
- Measured by taking into account
 - Number of seeks * average-seek-cost
 - Number of blocks read * average-block-read-cost
 - Number of blocks written * average-block-write-cost
- Cost to write a block is greater than cost to read a block
 - data is read back after being written to ensure that the write was successful

Recap: Data Access from Hard Disks

- Typically, not all the database can be kept in memory
- Databases are stored on hard disks
- Minimal unit of transfer: *block*
 - optimizing cost means minimizing block transfer

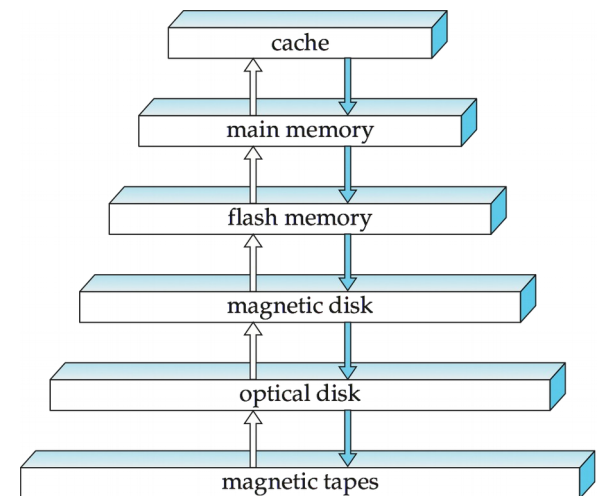


Measuring Query Cost

- 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
 - t_S – time for one disk seek (i.e., finding a block on the disk)
 - Cost for b block transfers plus S seeks
$$b * t_T + S * t_S$$
- We ignore CPU costs for simplicity
 - Real systems do take CPU cost into account
 - We do not include cost of writing output to disk

Measuring Query Cost

- 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
 - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
 - But hard to take into account for cost estimation



Selection Operation

- **File scan**
- Algorithm **A1** (**linear search**).
 - Seek first block
 - Scan this and each consecutive file block and test all records to see whether they satisfy the selection condition
 - $cost = b_r * t_T + t_S$
 - b_r denotes number of blocks containing records from relation r

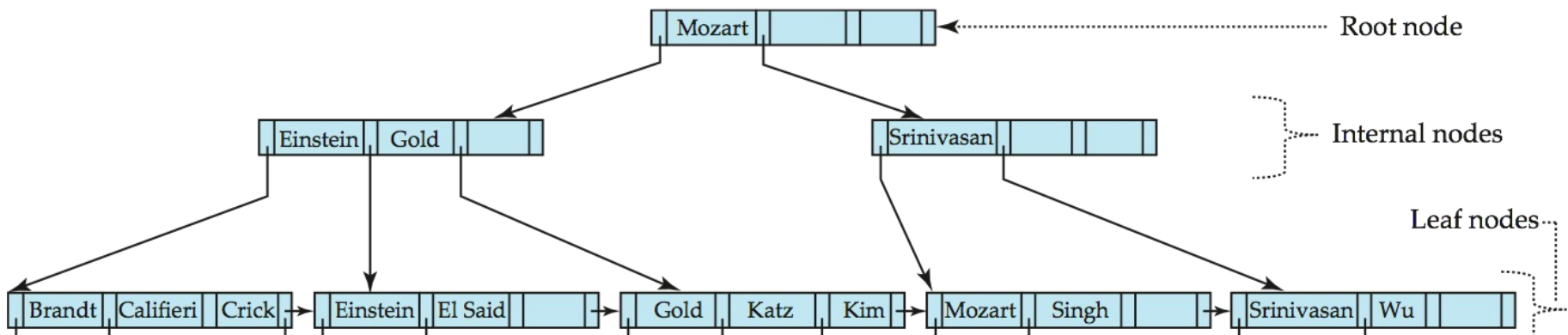
Assumption:
File is stored in
consecutive blocks

Selection Operation

- If selection is on a key attribute, can stop on finding the single record (if it exists)
 - $cost = (b_r/2) * t_T + t_S$
- 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 sense since data is not stored in order
 - except when there is an index available
 - and binary search requires more seeks than index search

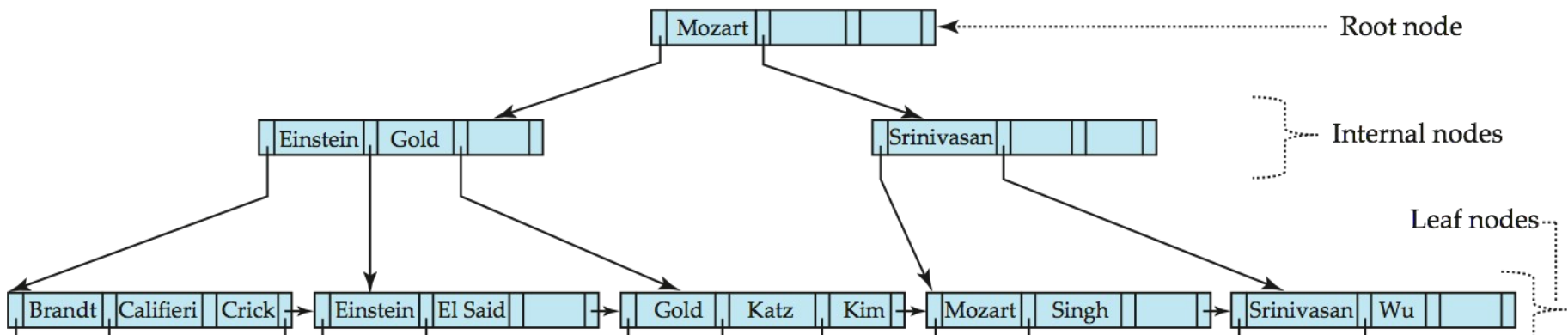
Selections Using Indices

- Assuming a B+ tree of height h_i
- **Index scan** – search algorithms that use an index
 - selection condition must be on search-key of index
- **A2 (primary index, equality on key)**. Retrieve a single record that satisfies the corresponding equality condition
- $Cost = (h_i + 1) * (t_T + t_S)$



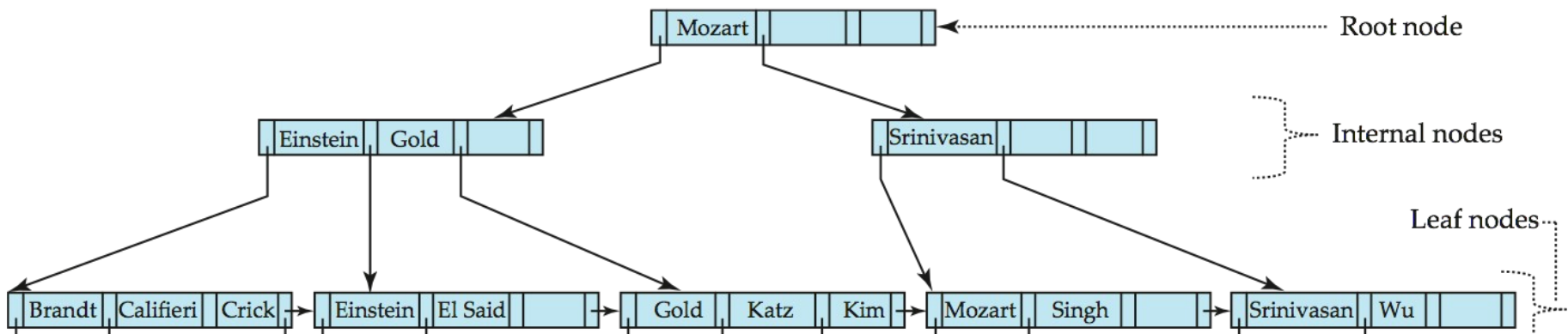
Selections Using Indices

- Assuming a B+ tree of height h_i
- A3 (primary index, equality on nonkey)** Retrieve multiple records.
 - Records will be on *consecutive* blocks
 - Let b = number of blocks containing matching records
 - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$



Selection Using Secondary Index

- **A4 (secondary index, equality on nonkey).**
- Retrieve a single record if the search-key is a candidate key
 - $Cost = (h_i + 1) * (t_T + t_S)$
- Retrieve multiple records if search-key is not a candidate key
 - each of n matching records may be on a different block
 - $Cost = (h_i + n) * (t_T + t_S)$
- Can be very expensive!



Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq v}(r)$ or $\sigma_{A \geq v}(r)$ by using
 - a linear file scan,
 - or by using an index
- **A5 (primary index, comparison)**. (Relation is sorted on A)
 - For $\sigma_{A \geq v}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
 - For $\sigma_{A \leq v}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
 - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$
 - identical to A3 (index on nonkey)

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq v}(r)$ or $\sigma_{A \geq v}(r)$ by using
 - a linear file scan,
 - or by using an index
- **A6** (secondary **index, comparison**). (Relation not sorted on A)
 - For $\sigma_{A \geq v}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
 - For $\sigma_{A \leq v}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
 - In either case, retrieving records that are pointed to
 - requires an I/O for each record
 - may be more expensive than linear file scan
 - $Cost = (h_i + n) * (t_T + t_S)$
 - identical to A4 (index on nonkey)

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
 - e.g., all students enrolled in the MMDS, in semester 4 or higher with GPA < 2.0
- **A7 (conjunctive selection using one index).**
 - Select a combination of θ_i and algorithms A2 through A6 that results in the least cost for $\sigma_{\theta_i}(r)$
 - Test other conditions on tuple after fetching it into memory buffer
- **A8 (conjunctive selection using composite index).**
 - Use appropriate composite (multiple-key) index if available
 - Use one of the algorithms A2-A4 with the least cost
 - Test other conditions on tuple after fetching it into memory buffer

Implementation of Complex Selections

- **A9 (conjunctive selection by intersection of identifiers)**
 - Requires indices with record pointers
 - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
 - all record pointers of students with program “MMDS”,
 - all record pointers of students with semester ≥ 4
 - all record pointers of students with GPA < 2.0
 - Then fetch records from file
 - minimizes block transfers as far as possible
 - If some conditions do not have appropriate indices
 - apply remaining tests in memory

Implementation of Complex Selections

- **Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$.
- **A10 (disjunctive selection by union of identifiers)**
 - Use corresponding index for each condition
 - collect pointers for each condition
 - use union of all the obtained sets of record pointers
 - Then fetch records from file
- Applicable only if *all* conditions have available indices
 - Otherwise use linear scan

Implementation of Complex Selections

- **Negation:** $\sigma_{\neg\theta}(r)$
 - Use linear scan on file
- Special case:
 - if very few records satisfy $\neg\theta$, and an index is applicable to θ
 - find satisfying records using index and fetch from file

Sorting

- Recap initial example:
SELECT name, building, salary
FROM instructor, building
WHERE instructor.dept_name = department.dept_name
AND salary > 75000
ORDER BY name
- Assuming we have indices on dept_name and salary
 - how do we sort the results efficiently?
- Variant 1: build an index on the sorting relation
 - and read from that index
 - hard to combine with other conditions
- Variant 2: sort in memory (e.g., QuickSort)
- Variant 3: use *external sort merge*

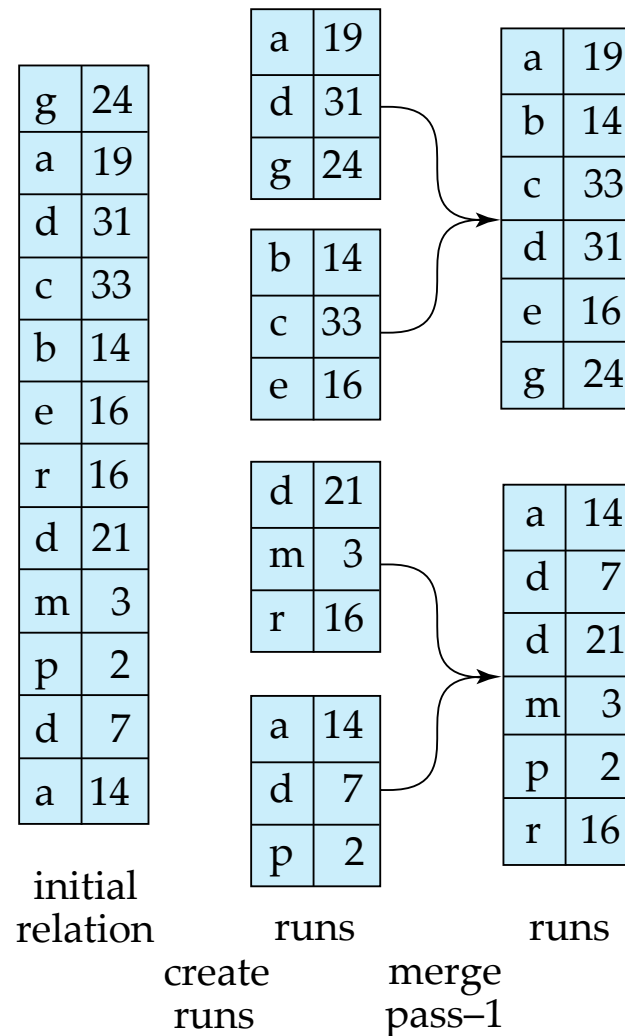
External Sort-Merge

- Two steps:
 - 1) Created partially sorted data chunks
 - 2) Merge the partially sorted chunks
- First step:
 - Let M be the memory capacity
 - **Create sorted runs**. Let i be 0 initially
Repeatedly do the following till the end of the relation:
 - (a) Read M blocks of relation into memory
 - (b) Sort the in-memory blocks
 - (c) Write sorted data to run R_i ; increment iLet the final value of i be N

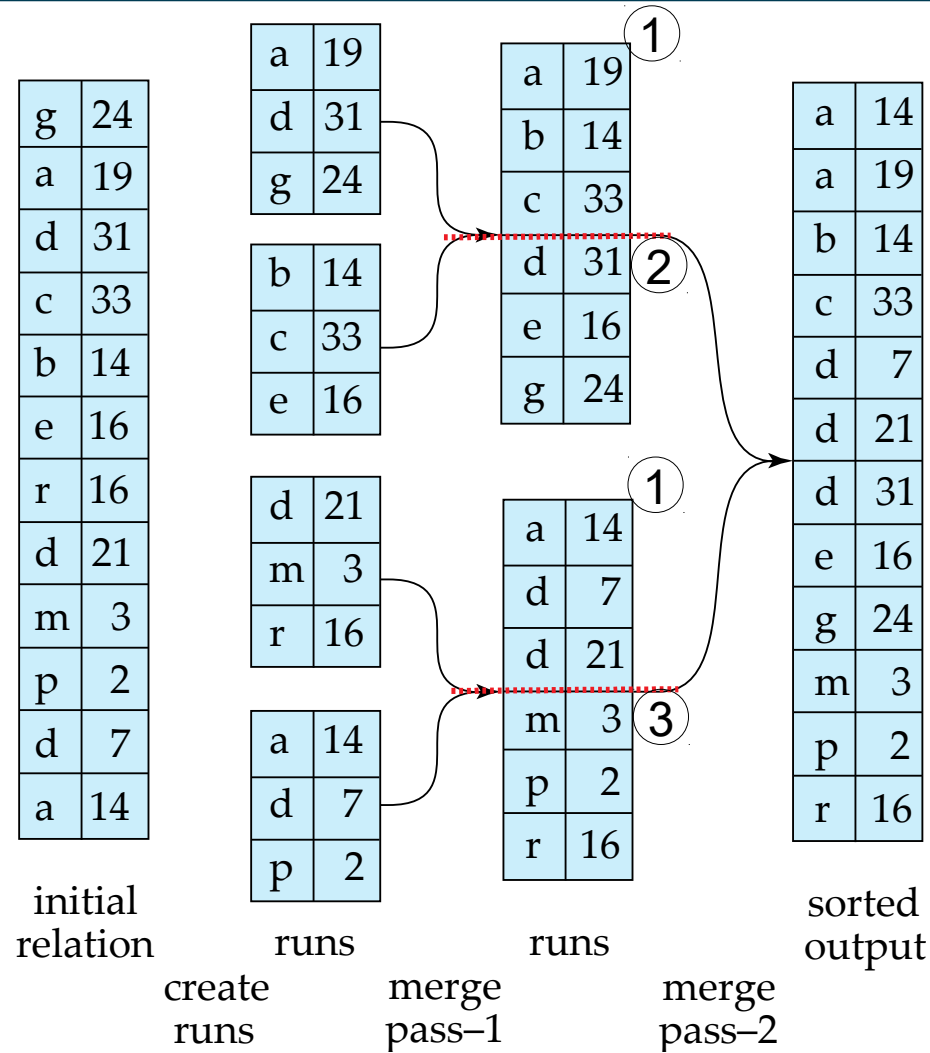
External Sort-Merge

- Second step: merge the runs
 - **Merge the runs (N-way merge).** We assume (for now) that $N < M$.
 - Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
- repeat**
- Select the first record (in sort order) among all buffer pages
- Write the record to the output buffer.
- If the output buffer is full write it to disk.
- Delete the record from its input buffer page.
- If** the buffer page becomes empty **then**
- read the next block (if any) of the run into the buffer.
- until** all input buffer pages are empty
- If $N \geq M$, several merge *passes* are required
 - In each pass, contiguous groups of $M - 1$ runs are merged

External Sort-Merge



External Sort-Merge



Join Operations

- Recap: Initial example:
SELECT name, building, salary
FROM instructor, building
WHERE instructor.dept_name = department.dept_name
AND salary > 75000
ORDER BY name
- Several different algorithms to implement joins, e.g.
 - Nested loop join
 - Block nested loop join
 - Indexed nested loop join
 - Merge join
 - Hash join
- Choice based on cost estimate
- Examples use the following information
 - Number of records of *student*: 5,000 *takes*: 10,000
 - Number of blocks of *student*: 100 *takes*: 400

Nested Loop Join

- To compute the theta join $r \bowtie_{\theta} s$
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 θ
 if they do, add $t_r \cdot t_s$ to the result.
 end
end
- r is called the **outer relation** and s the **inner relation** of the join
- Requires no indices and can be used with any kind of join condition
- Expensive since it examines every pair of tuples in the two relations

Nested Loop Join

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$\begin{array}{ll} n_r * b_s + b_r & \text{block transfers, plus} \\ n_r + b_r & \text{seeks} \end{array}$$

- Assuming worst case memory availability cost estimate is

- with *student* as outer relation:

$5000 * 400 + 100 = 2,000,100$ block transfers,

$5000 + 100 = 5,100$ seeks

- with *takes* as the outer relation

$10000 * 100 + 400 = 1,000,400$ block transfers and 10,400 seeks

records/blocks
student: 5,000/100
takes: 10,00/400

Nested Loop Join

- Best case: the smaller relation fits entirely in memory
 - use that as the inner relation
 - reduces cost to $b_r + b_s$ block transfers and two seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers
 - 100 blocks reading the *student* relation into memory
 - 400 blocks of the *takes* relation

records/blocks
student: 5,000/100
takes: 10,00/400

Block Nested Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation
- Algorithm uses four nested loops

```
for each block  $B_r$  of  $r$  do begin  
  for each block  $B_s$  of  $s$  do begin  
    for each tuple  $t_r$  in  $B_r$  do begin  
      for each tuple  $t_s$  in  $B_s$  do begin  
        Check if  $(t_r, t_s)$  satisfy the join condition  
        if they do, add  $t_r \bullet t_s$  to the result.  
      end  
    end  
  end  
end
```

Block Nested Loop Join

- Worst case: only one block of each relation fits in memory
 - estimate: $b_r * b_s + b_r$ block transfers + $2 * b_r$ seeks
 - Each block in the inner relation s is read once for each *block* in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks
- Improvements to nested loop and block nested loop algorithms:
 - If equi-join attribute forms a key on inner relation, stop inner loop on first match
 - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
 - Use index on inner relation if available (next slide)

Indexed Nested Loop Join

- Index lookups can replace file scans if
 - join is an equi-join or natural join and
 - an index is available on the inner relation's join attribute
- For each tuple t_r in the outer relation r , use the index to look up tuples in s that satisfy the join condition with tuple t_r
- Worst case: buffer has space for only one page of r , and, for each tuple in r , we perform an index lookup on s
- If indices are available on join attributes of **both** r and s
 - use the relation with fewer tuples as the outer relation

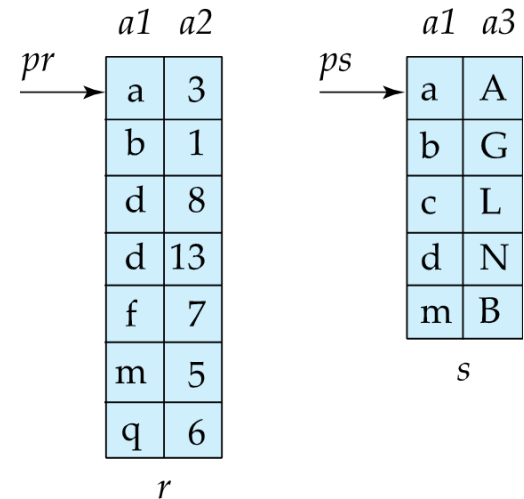
Cost of Nested Loop with and without Index

- Compute $student \bowtie takes$, with $student$ as the outer relation
 - Let $takes$ have a primary B⁺-tree index on the attribute ID , which contains 20 entries in each index node
 - Since $takes$ has 10,000 tuples, the height of the tree is 4
 - i.e.: five block transfers to find the actual data
- $student$ has 5000 tuples
- Cost of block nested loops join
 - $400 * 100 + 100 = 40,100$ block transfers + $2 * 100 = 200$ seeks
 - assuming worst case memory
(may be significantly less with more memory)
- Cost of indexed nested loops join
 - $100 + 5000 * 5 = 25,100$ block transfers and seeks

records/blocks
student: 5,000/100
takes: 10,00/400

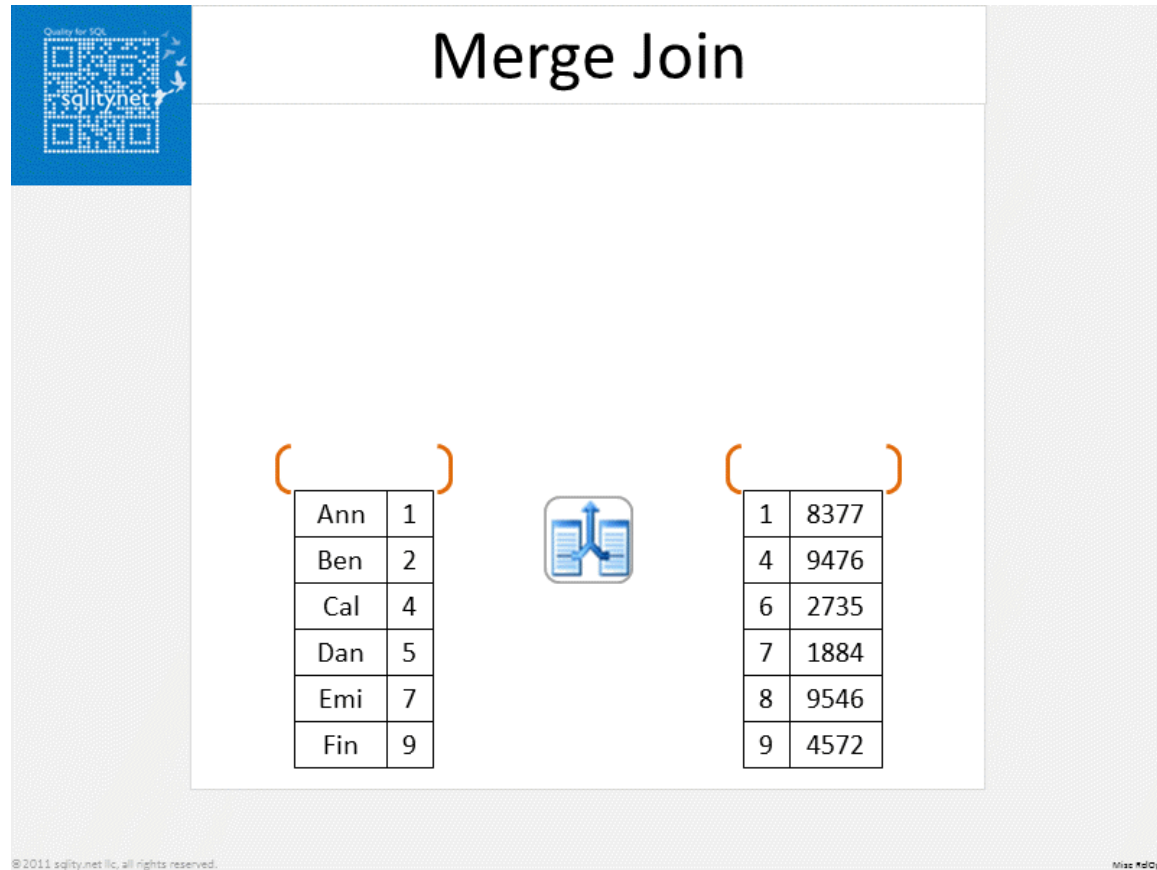
Merge Join

- Sort both relations on their join attribute (if not already sorted on the join attributes)
 - Merge the sorted relations to join them
 - Join step is similar to the merge stage of the sort-merge algorithm.
- Main difference is handling of duplicate values in join attribute:
 - every pair with same value on join attribute must be matched
- Detailed algorithm in book



Merge Join

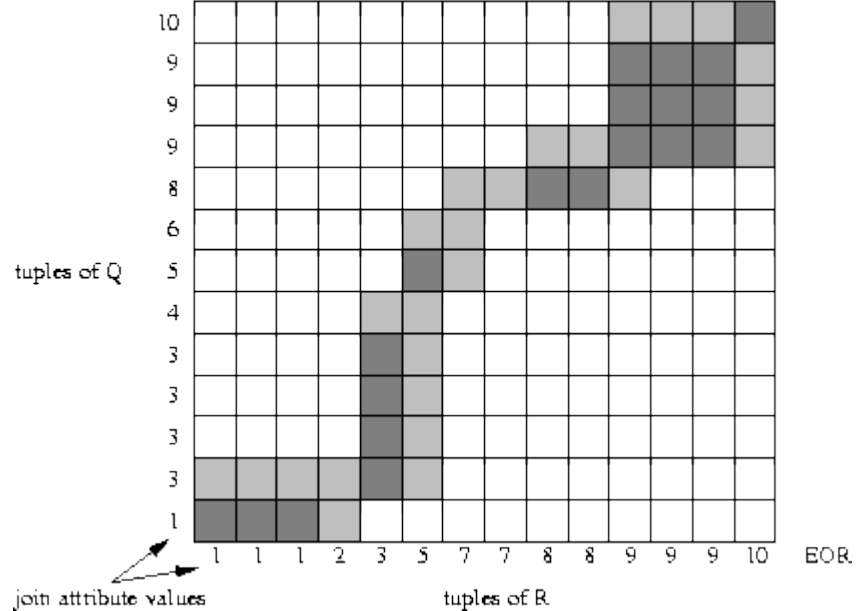
- Example from <http://sqlity.net/en/1480/a-join-a-day-the-sort-merge-join/>



Merge Join

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is:
 - $b_r + b_s$ block transfers + $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
 - plus the cost of sorting if relations are unsorted

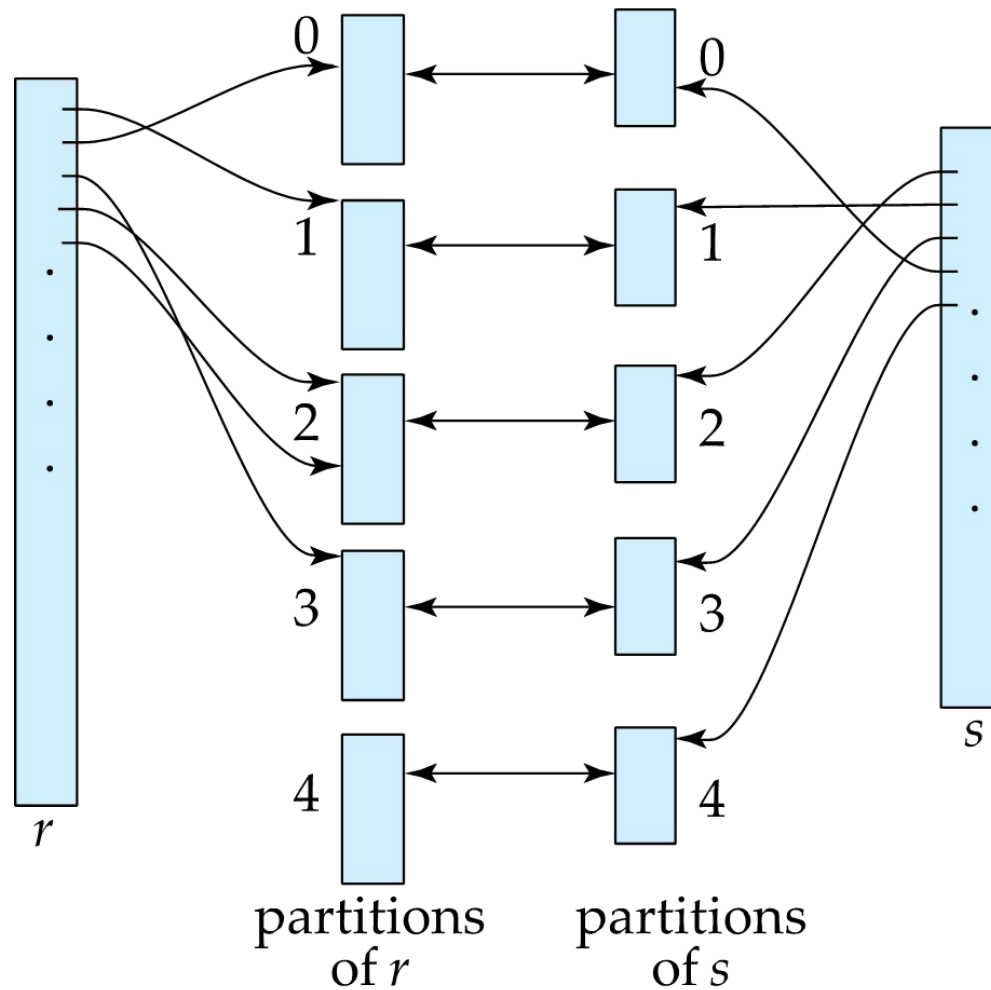
b_b : no- of buffer blocks
allocated to each relation



Hash Join

- Applicable for equi-joins and natural joins
 - idea: partition joined relations using hashes
 - only compute joins based on the hash partitions
- A hash function h is used to partition tuples of *both* relations
- h maps *JoinAttrs* values to $\{0, 1, \dots, n\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join
 - r_0, r_1, \dots, r_n denote partitions of r tuples
 - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[\text{JoinAttrs}])$
 - s_0, s_1, \dots, s_n denotes partitions of s tuples
 - Each tuple $t_s \in s$ is put in partition s_i where $i = h(t_s[\text{JoinAttrs}])$

Hash Join



Hash Join

Computing Hash Join:

1. Partition the relation s using hashing function h
2. Partition r similarly
3. For each i ($1 \leq i \leq \text{number of partitions}$):
 - (a) Load s_i into memory and build an in-memory hash index on it using the join attribute (using a different hash function)
 - (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index

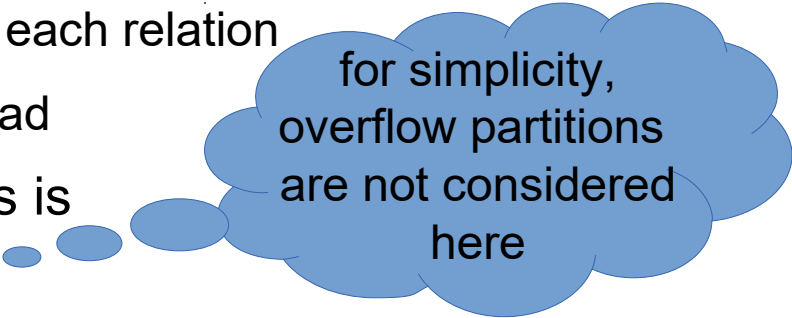


why different?

Relation s is called the **build input** and r is called the **probe input**

Hash Join

- Complexity
 - Building the hash: reading each block in each relation, and writing the partition back to disk: $2(b_r + b_s)$
 - Computing the join: reading each partition
- Partitions can also be underfull blocks
 - i.e., there might be n_h extra partitions for each relation
 - each of those needs to be written and read
- Thus, the total number of block transfers is
 - $3(b_r + b_s) + 4n_h$
- Number of seeks
 - need to seek original and partitioned blocks, respecting underfull blocks
 - i.e. $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$



for simplicity,
overflow partitions
are not considered
here

Joins with Complex Conditions

- Join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute the result of one of the simpler joins $r \bowtie_{\theta_i} s$
- final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$$

- Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$$

Duplicate Elimination & Projection

- In relational algebra, there are no duplicates by definition
 - i.e., each projection yields a unique result
- In SQL queries, they can be explicitly discarded
 - `SELECT DISTINCT ...`
- Duplicates can be eliminated either via sorting or hashing
 - After sorting, duplicates are adjacent, and can be easily removed passing over the data
 - with sort merge, duplicate elimination can be done early
 - With hashing, they are sorted into the same bucket, and can be detected locally
- Projection
 - perform projection on each tuple
 - then run duplicate removal

Aggregation

- **Aggregation** can be implemented similarly to duplicate elimination
- Sorting or hashing
 - bring tuples in the same group together
 - then apply aggregate functions on each group
- Optimization:
 - combine tuples in the same group during run generation and intermediate merges
 - compute partial aggregate values
 - count, min, max, sum: keep aggregate values on tuples found so far in the group
 - avg: keep sum and count, and divide sum by count at the end

Outer Joins

- **Outer join** can be computed either as
 - a join followed by addition of null-padded non-participating tuples
 - by modifying the join algorithms
- Modifying merge join to compute $r \sqsupset \bowtie s$
 - In $r \sqsupset \bowtie s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
 - During merging, for every tuple t_r from r that do not match any tuple in s , output t_r padded with nulls
 - Right outer join and full outer join can be computed similarly
- Modifying hash join to compute $r \sqsupset \bowtie s$
 - If r is probe relation, output non-matching r tuples padded with nulls
 - If r is build relation, keep track of which r tuples matched s tuples
 - at the end of s_i , output non-matched r tuples padded with nulls

Evaluation of Expressions

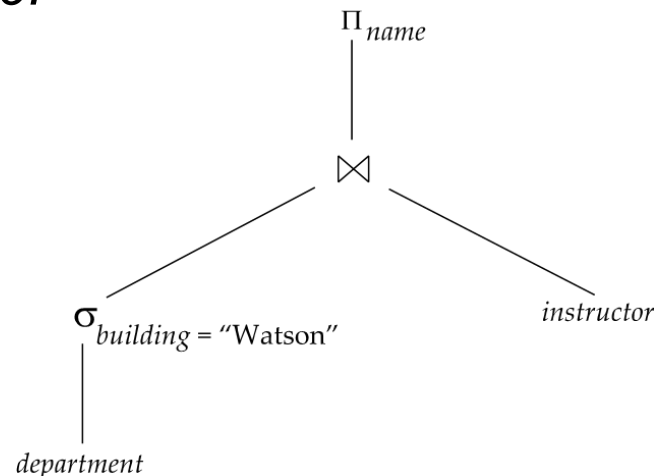
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
 - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
 - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building="Watson"}(department)$$

then compute the store its join with *instructor*, and finally compute the projection on *name*.



Cost of Materialization

- Materialized evaluation is always applicable
 - Cost of writing results to disk and reading them back can be quite high
 - Our cost formulas for operations ignore cost of writing results to disk, so
 - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
 - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation:** evaluate several operations simultaneously, passing the results of one operation on to the next
 - E.g., in previous expression tree, do not store result of $\sigma_{building="Watson"}(department)$
 - instead, pass tuples directly to the join
 - do not store result of join, pass tuples directly to projection
- Much cheaper than materialization: no need to store a temporary relation to disk
 - Pipelining may not always be possible – e.g., sort, hash-join
 - For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining

- In **demand driven** or **lazy** evaluation
 - system repeatedly requests next tuple from top level operation
 - Each operation requests next tuple from children operations as required, in order to output its next tuple
 - In between calls, operation has to maintain “**state**” so it knows what to return next
- In **producer-driven** or **eager** pipelining
 - Operators produce tuples eagerly and pass them up to their parents
 - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
 - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
 - System schedules operations that have space in output buffer and can process more input tuples
- Alternative names: **pull** and **push** models of pipelining

Summary

- How are queries executed?
 - Each query is a sequence of operations
 - Sequence: materialization vs. pipelining
- Implementation of different operations
 - Selection
 - Joins
 - Sorting
 - Projection
 - ...
- Estimation of query cost
 - block seeks and block transfers
 - gives way to query optimization (next lecture)

Questions?

