CPT-S 415

Big Data

Yinghui Wu

EME B45

CPT-S 415
Big Data

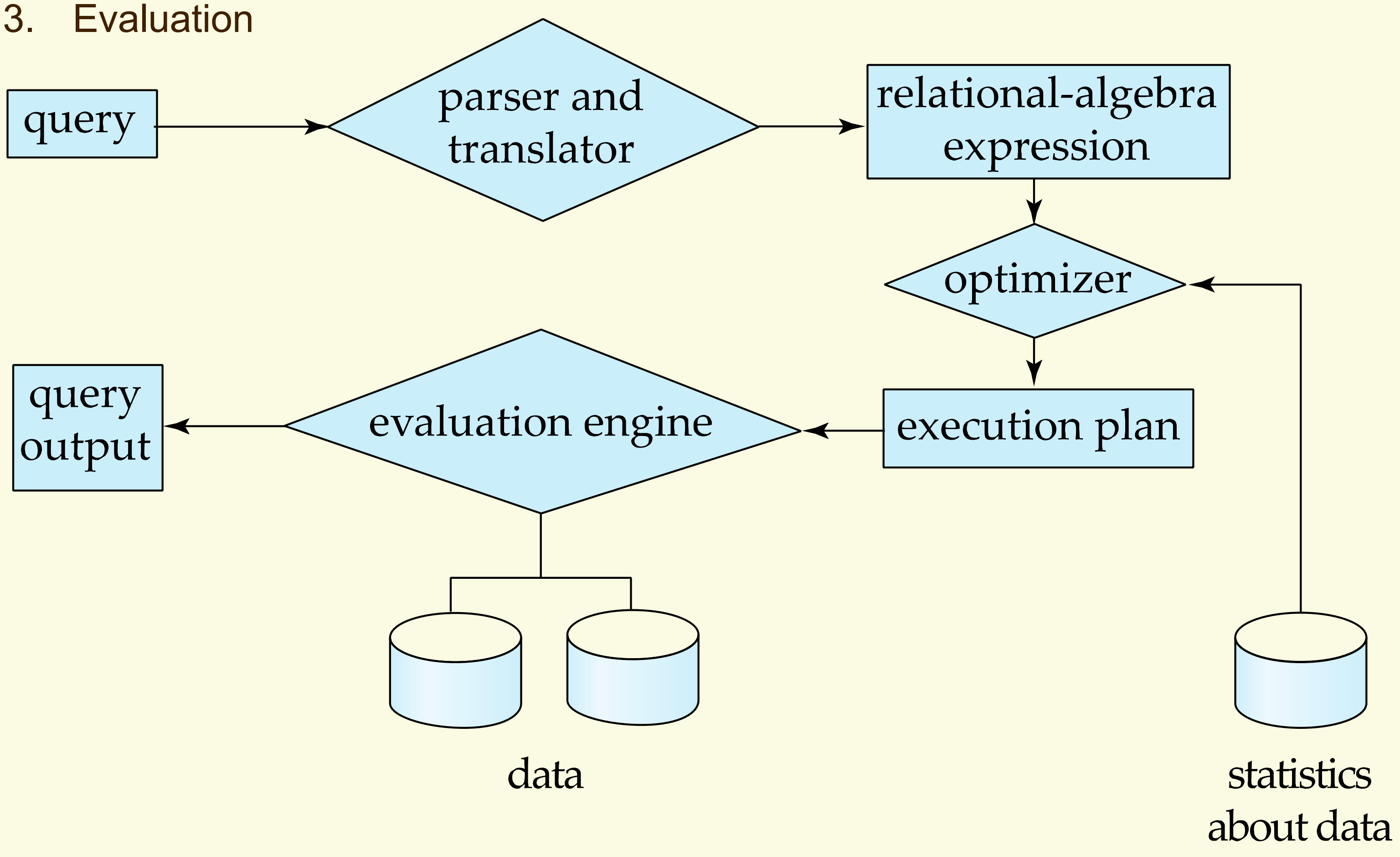
Query Processing

- ✓ Overview: query plan & optimization
- ✓ Measures of Query Cost
- ✓ Query optimization: Operators

Query processing: overview

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.
  - For RDBMS and SQL DB: relational algebra.
  - Parser checks syntax, verifies relations
- ✓ Optimization
  - Generate query(evaluation) plan from relational algebra
- ✓ 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 : Optimization

- ✓ 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
- ✓ 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

How to measure?

Example Query

```
SELECT title, startTime
FROM Movie M, Plays P, Reviews R
WHERE M.title = P.movie AND M.title
      = R.movie AND P.location ="New
      York " AND M.actor = “Morgan
      Freeman” AND R.avgStars >= 3
```

- ✓ Query is parsed, generally broken into predicates, then converted into a **logical query plan used by the optimizer**

Example Logical Query Plan

SELECT title, startTime

FROM Movie M, Plays P, Reviews R

WHERE M.title = P.movie AND M.title = R.movie AND P.location ="New York " AND M.actor = “Morgan Freeman” AND R.avgStars >= 3

$$\begin{tikzcd}
 & & \pi_{\text{title,startTime}} & \\
 & & \bowtie_{\text{title = movie}} \arrow[u] & \\
 & \bowtie_{\text{title = movie}} \arrow[ur] & & \sigma_{\text{avgStars} \geq 3} \arrow[ul] \\
\sigma_{\text{actor= “Morgan Freeman”}} \arrow[ur] & & \sigma_{\text{location=“New York”}} \arrow[ul] & \text{Reviews} \arrow[u] \\
\text{Movie} \arrow[u] & & \text{Plays} \arrow[u] &
\end{tikzcd}$$

Query Optimization

- ✓ Goal: compare all **equivalent** query expressions and their low-level implementations (**operators**)

- ✓ Can be divided into:
  - Plan enumeration ("search")
  - Cost estimation

Foundations of Query Plan Enumeration

- ✓ Exploit properties of the relational algebra to find equivalent expressions, e.g.:

  $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

  $\sigma_\alpha(R \bowtie S) = (\sigma_\alpha(R) \bowtie S)$ if α only refers to R
- ✓ Assume selection, projection are always done as early as possible
- ✓ Joins (generally) satisfy **principle of optimality:**

  Best way to compute $R \bowtie S \bowtie T$ takes advantage of the best way of doing a 2 way join, followed by one additional join:

  $(R \bowtie S) \bowtie T$, $(R \bowtie T) \bowtie S$, or $(S \bowtie T) \bowtie R$

  (uses optimal way of computing this expression)

Enumerating Plans

Can formulate as a dynamic programming problem:

1. **Base case**: consider all possible ways of accessing the base tables, with all selections & projections pushed down
2. **Recursive case** (i=2 … number of joins+1): explore all ways to join results with (i-1) tables, with one additional table
    - Common heuristic – only considers **linear** plans
3. Then repeat process for all **Cartesian products**
4. Apply **grouping** and aggregation

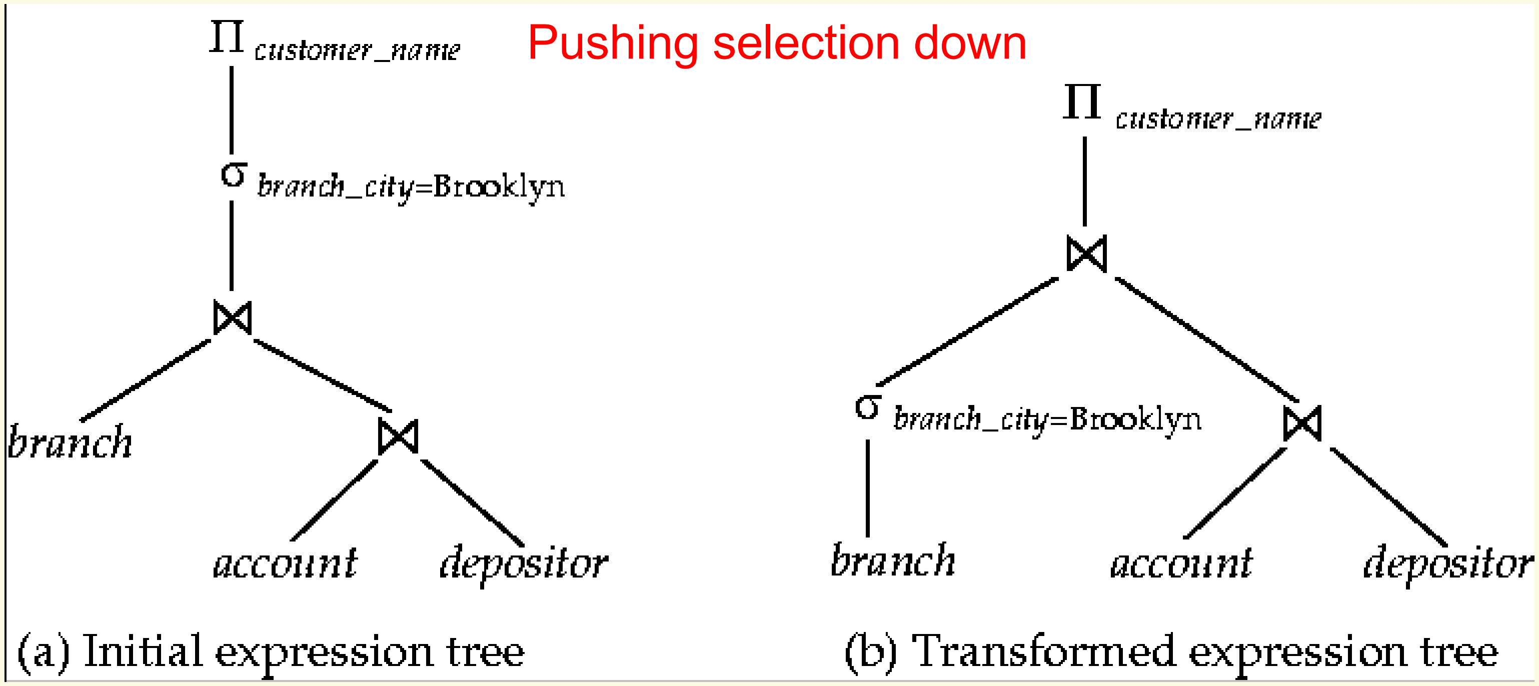
Find the Best Join Plan

Complexity: $O(3^n)$
Space complexity: $O(2^n)$

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    for each non-empty proper subset S1 of S
    P1= findbestplan(S1)  P2= findbestplan(S - S1)
    A = best algorithm for joining results of P1 and P2  cost = P1.cost +
    P2.cost + cost of A
    if cost < bestplan[S].cost
    bestplan[S].cost = cost
    bestplan[S].plan = "execute P1.plan; execute P2.plan;
    join results of P1 and P2 using A"
    return bestplan[S]
```

Query Optimization Example

- Query: find the names of all customers who have an account at any branch located in Brooklyn
- Relational expression:

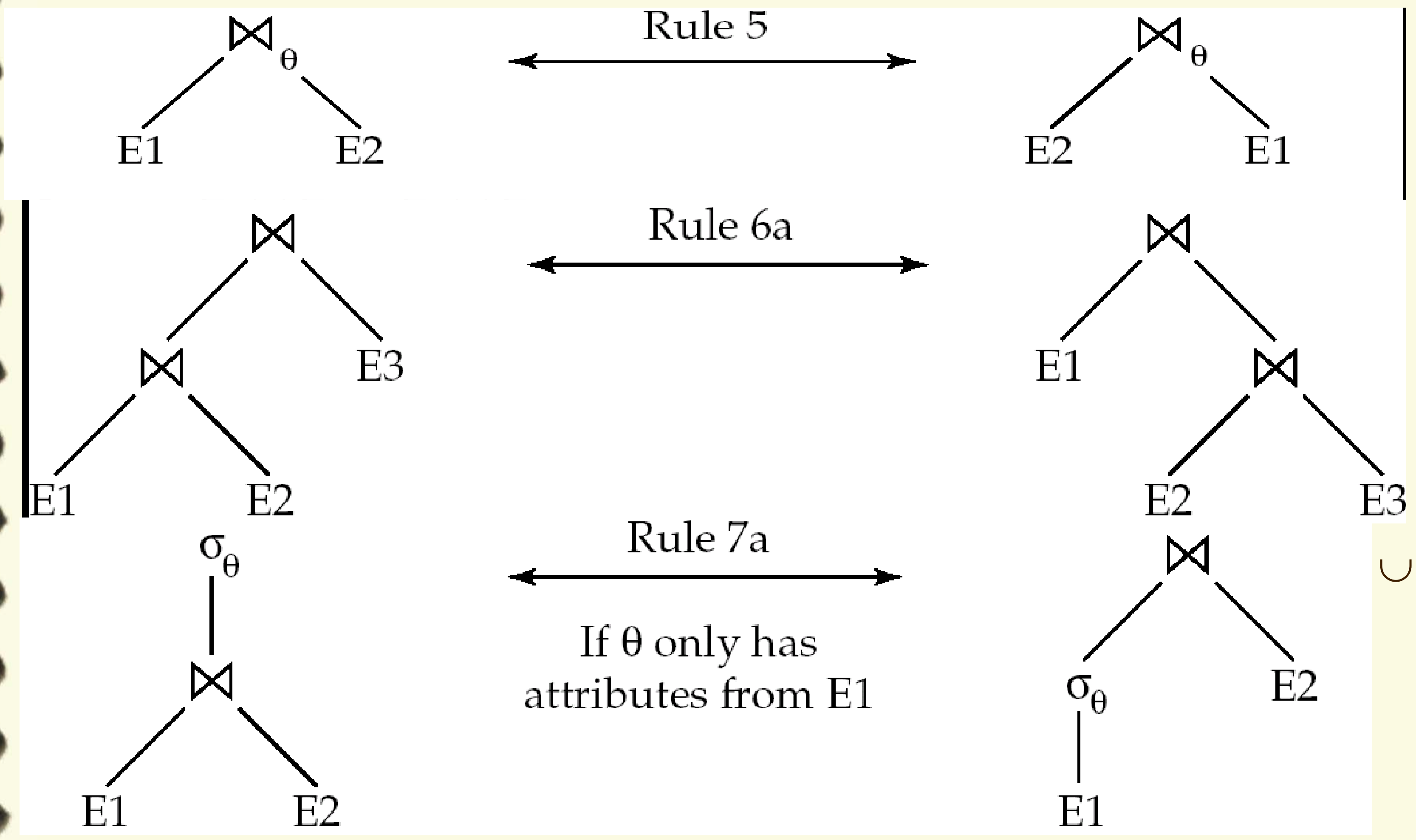


$\Pi_{customer\_name}$
Pushing selection down
$\sigma_{branch\_city=\text{Brooklyn}}$
$\bowtie$
branch
account
depositor
(a) Initial expression tree
(b) Transformed expression tree

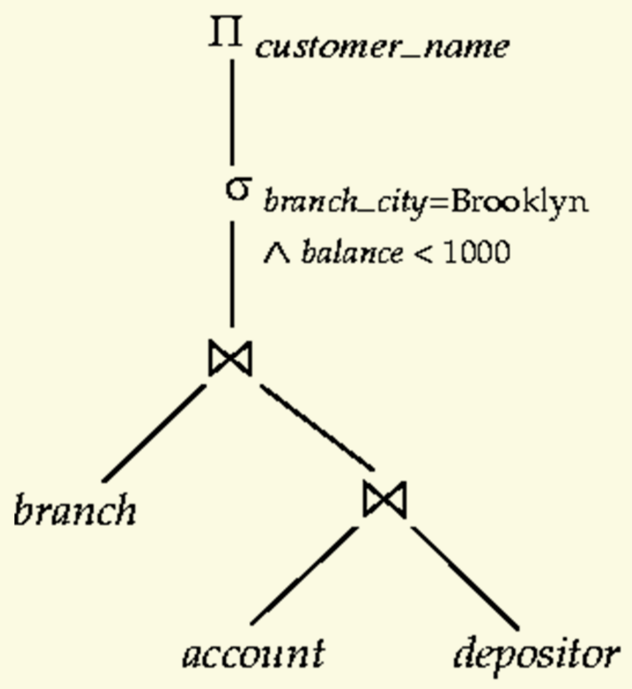
Enumeration of Equivalent Plans

- $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- $\Pi_{L_1}(\Pi_{L_2}(\mathrm{K}(\Pi_{Ln}(E))\mathrm{K})) = \Pi_{L_1}(E)$
- $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
- $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$
- $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
- $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
- $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 2 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$
- $\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$
- $\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$

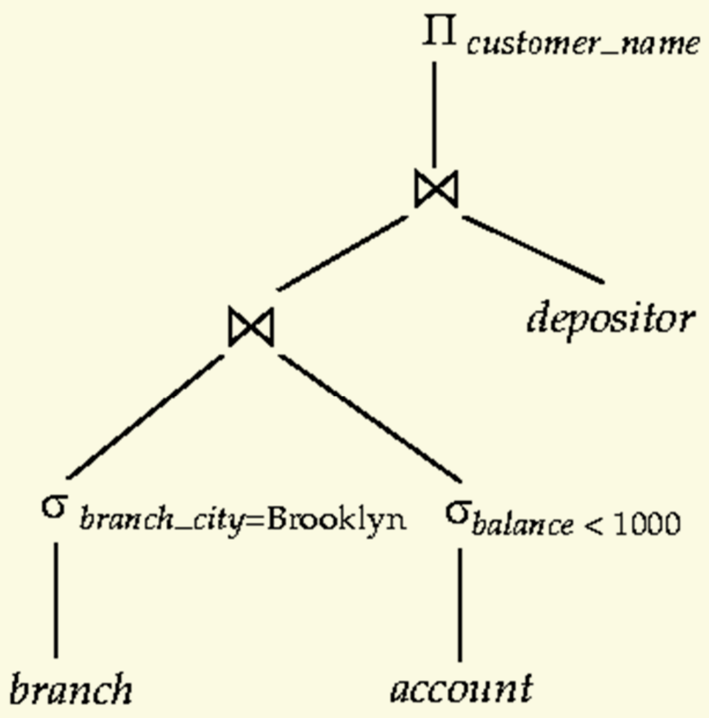
Enumeration of Equivalent Plans



Example



(a) Initial expression tree


$\Pi_{customer\_name}$
$\bowtie$
$\bowtie$
depositor
$\sigma_{branch\_city=\text{Brooklyn}}$
$\sigma_{balance < 1000}$
branch
account

(b) Tree after multiple transformations

Query optimization：cost analysis

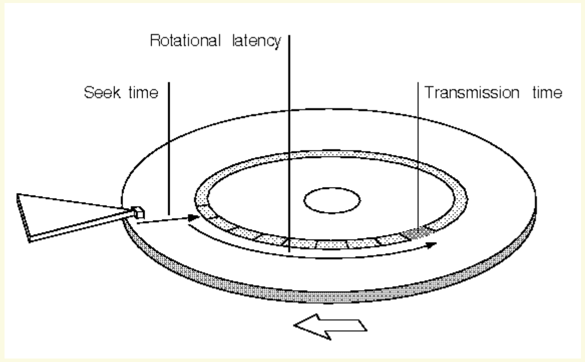
Measures of Query Cost

- ✓ Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or network *communication*
- ✓ Disk access is the predominant cost (I/O). Measured by
  - 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

Measures of Query Cost (Cont.)

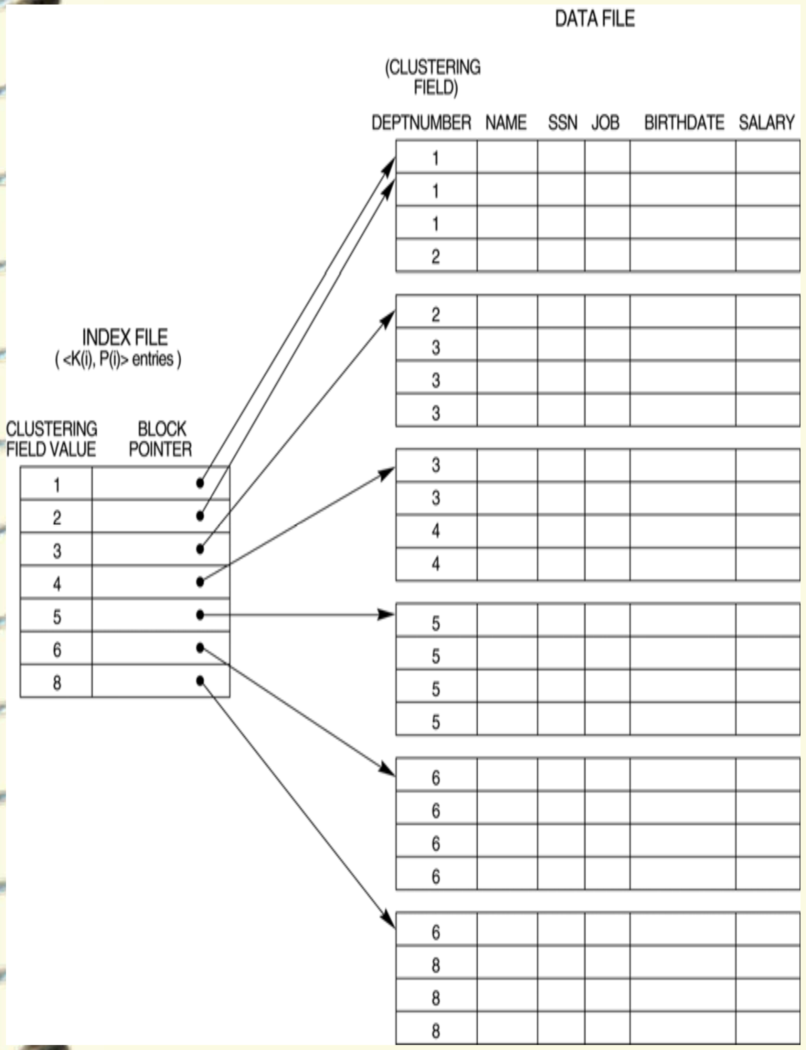
- ✓ I/O cost
  - Use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

    $b * t_T + S * t_S$



- ✓ CPU cost
  - for main-memory algorithms
- ✓ Index are often used to reduce cost! – a flashback

Single-level index: Primary and Secondary



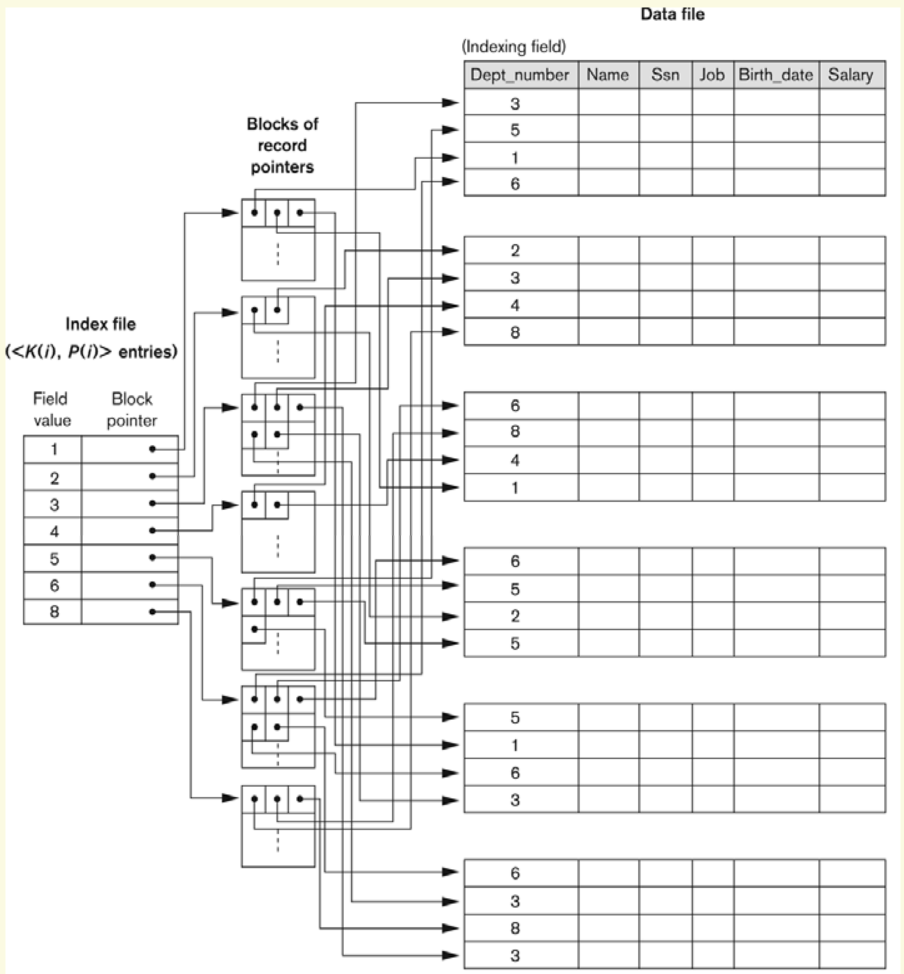
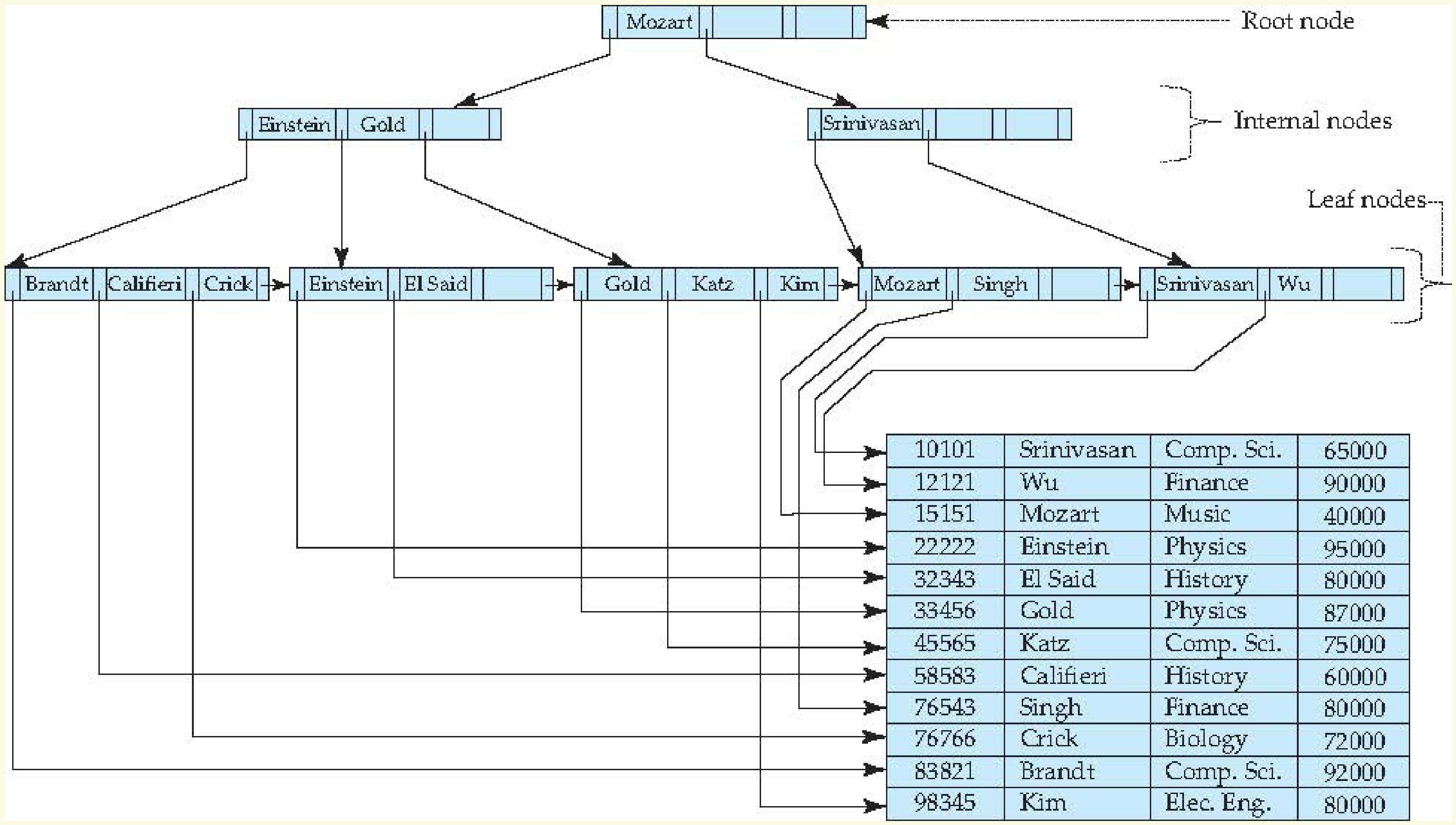


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Multi-level index: Example of B+-Tree



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Catalog information for cost estimation

- ✓ **NR**: # of tuples in a relation R
- ✓ **BR**: # of blocks that contain tuples of relation R
- ✓ **SR**: size of tuple of R
- ✓ **FR**: blocking factor; # of tuples from R that fit into one block: FR=NR/BR
- ✓ **V(A, R)**: # of distinct value for attribute A in R.
- ✓ **Sc(A,R)**: selectivity of attribute A = average number of tuples of R that satisfy an equality condition on A; Sc(A,R)=NR/V(A,R)

Query Plan Cost Estimation

For each expression, predict the **cost and output size** given what we know about the inputs

Requires significant information:

- A **cost formula** for each algorithm, in terms of disk I/Os, CPU speed, …
- **Calibration parameters** for host machine performance
- Information about the **distributions** of join and grouping columns

Query optimization：

case study: selection

Selections

- ✓ **Selection: File scan**
  - **A1: Linear scan:** Scan each file block and test. Cost: BR* $t_T + t_S$
- ✓ **Selection: Index**
  - **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
    - *Cost* = $(h_i + 1) * (t_T + t_S)$ -- h_i: *height of the (B+-tree)*
  - **A3** (**primary index, equality on Nonkey**) Retrieve multiple records.
    - Records will be on consecutive blocks
      - Let b = number of blocks containing matching records
    - *Cost* = $h_i (t_T + t_S) + b^* t_T$

      $= h_i (t_T + t_S) + Sc(A,R)/FR * t_T$

NR: # of tuples
BR: # of blocks
SR: size of tuple of R
FR: block size
Sc(A,R): selectivity

Selections Using Indices

- ✓ **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!

NR: # of tuples
BR: # of blocks
SR: size of tuple of R
FR: block size
Sc(A,R): selectivity

Selections Involving Comparisons

- **Comparison: $\sigma_{A \leq V}$ (r) or $\sigma_{A \geq V}$(r)**
  - a linear file scan,
  - or by using indices in the following ways:
- **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
- **A6** (**secondary index, comparison**).
  - scan index sequentially to find pointers to records
  - retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper

NR: # of tuples
BR: # of blocks
SR: size of tuple of R
FR: block size
Sc(A,R): selectivity

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of θ_i 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.
- **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.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
- **A10** (**disjunctive selection by union of identifiers**).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
    - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Query optimization：

case study: Sorting

Sorting

- ✓ We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- ✓ For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.

External Sort-Merge

Let M denote memory size (in pages).

1. **Create sorted runs**. Let i be 0 initially.

   Repeatedly do the following till the end of the relation:

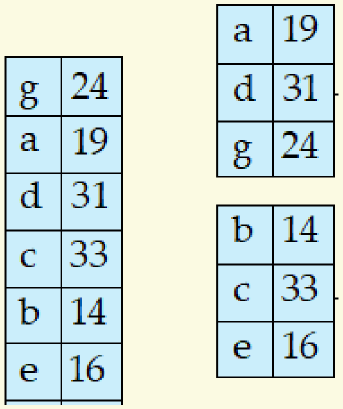
   (a) Read M blocks of relation into memory

   (b) Sort the in-memory blocks

   (c) Write sorted data to run file R_i; increment i.

   Let the final value of i be N

2. *Merge the runs (next slide)…..*



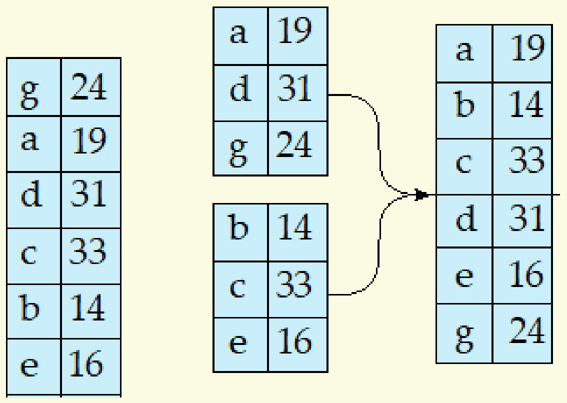
g	24
a	19
d	31
c	33
b	14
e	16

a	19
d	31
g	24

b	14
c	33
e	16

External Sort-Merge (Cont.)

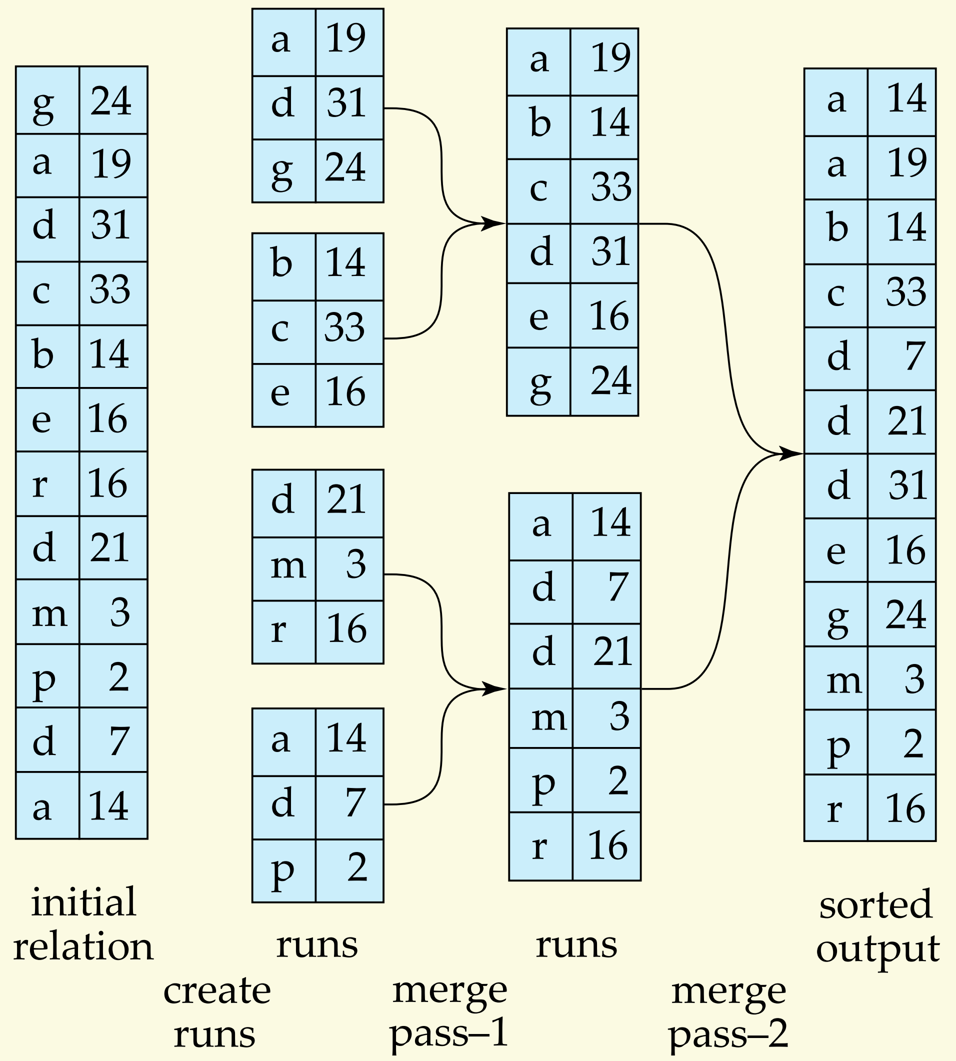
2. **Merge the runs (N-way merge)**. Assume that $N < M$.
   1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If the output buffer is full write it to disk.
      3. 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.
   3. **until** all input buffer pages are empty:



External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

Example: External Sorting Using Sort-Merge



External Merge Sort (Cont.)

- ✓ Cost analysis:
  - 1 block per run leads to too many seeks during merge
    - Instead use b_b buffer blocks per run

      ➔ read/write b_b blocks at a time
    - Can merge $\lfloor M/b_b \rfloor - 1$ runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r/M) \rceil$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus total number of block transfers for external sorting:

$$b_r (2 \lceil \log_{\lfloor M/bb \rfloor - 1} (b_r / M) \rceil + 1) \lceil$$

  - Seeks: next slide

NR: # of tuples
BR: # of blocks
SR: size of tuple of R
FR: block size
Sc(A,R): selectivity

External Merge Sort (Cont.)

✓ Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2\lceil b_r / M \rceil$
  - During the merge phase
    - Need $2\lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:
      $2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{\lfloor M/bb \rfloor - 1}(b_r / M) \rceil - 1)$

Query optimization：

case study: Join

Join Operation

- ✓ Several different algorithms to implement joins
  - 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 $\theta$
      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 (Cont.)

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

$n_r * b_s + b_r$	block transfers, plus
$n_r + b_r$	seeks

- ✓ If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- ✓ Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 $*$ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 $*$ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- ✓ If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- ✓ Block nested-loops algorithm (next slide) is preferable.

Block Nested-Loop Join

- ✓ Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
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 $t_r \cdot t_s$ to the result.
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

- ✓ Worst case 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.
  - Use index on inner relation if available (next slide)

Indexed Nested-Loop Join

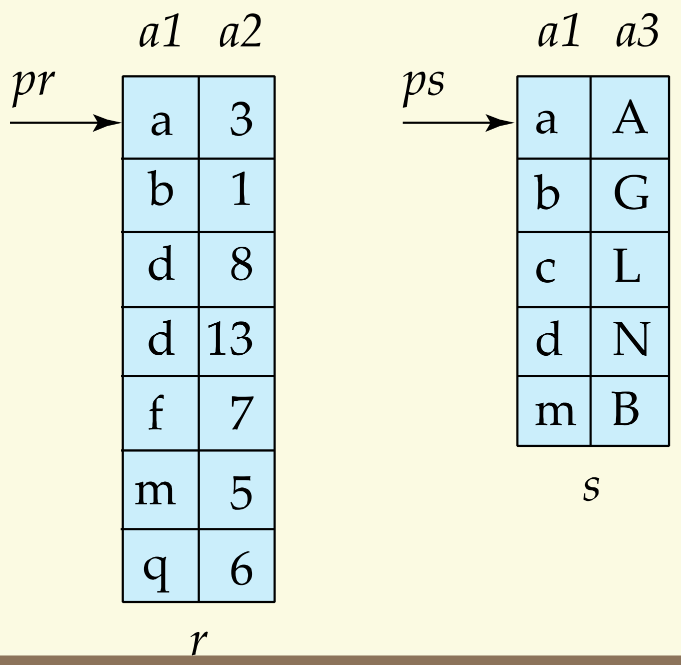
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r\,(t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- ✓ 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, and one more access is needed 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.
  - – CPU cost likely to be less than that for block nested loops join

Merge-Join

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



a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

a1	*a3*
a	A
b	G
c	L
d	N
m	B

Merge-Join (Cont.)

- ✓ 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:

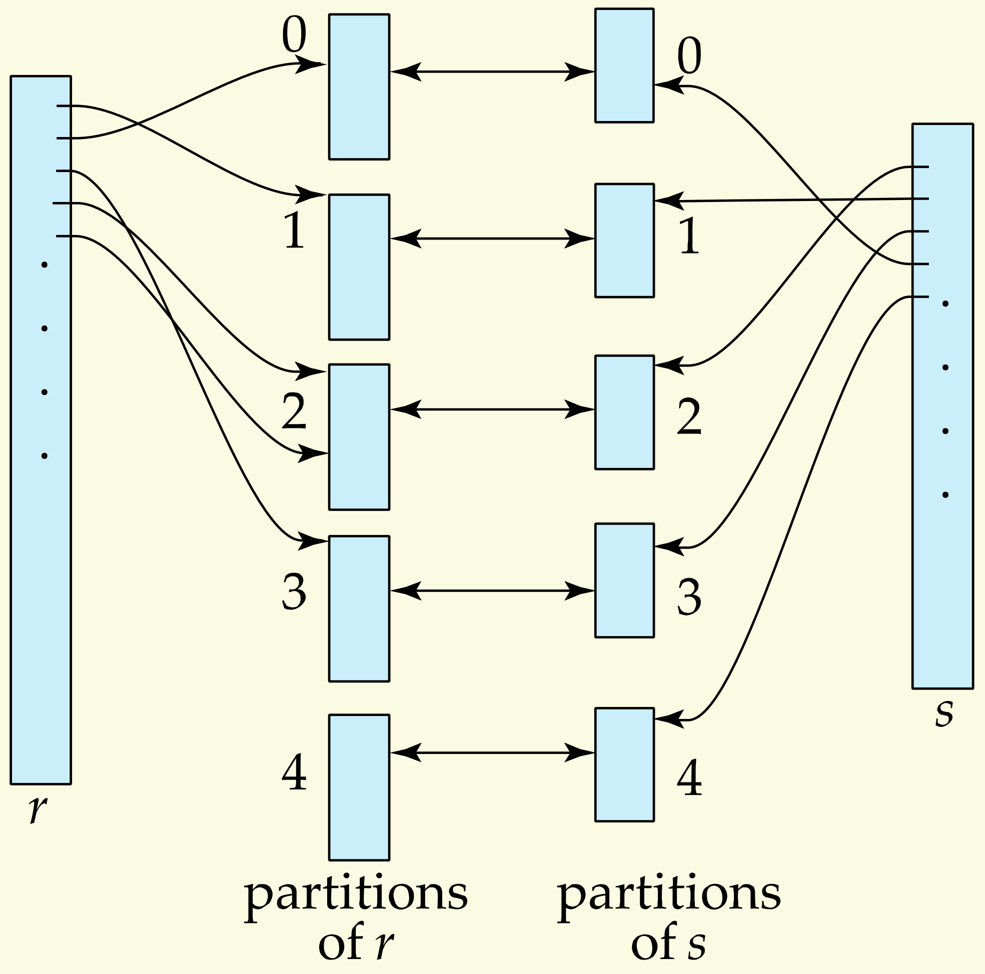
  $$b_r + b_s \text{ block transfers } + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ seeks}$$

  – + the cost of sorting if relations are unsorted.
- ✓ **hybrid merge-join:** If one relation is sorted, and the other has a secondary B$^+$-tree index on the join attribute
  - – Merge the sorted relation with the leaf entries of the B$^+$-tree .
  - – Sort the result on the addresses of the unsorted relation’s tuples
  - – Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - • *Sequential scan more efficient than random lookup*

Hash-Join

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

Hash-Join (Cont.)



Hash-Join (Cont.)

- ✓ *r* tuples in r_i need only to be compared with *s* tuples in s_i Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h.*
   (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. Output the concatenation of their attributes.

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

Hash-Join algorithm (Cont.)

- ✓ The # of partitions *n* and the hash function *h* is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
  - The probe relation partitions r_i need not fit in memory
- ✓ **Recursive partitioning** required if number of partitions *n* is greater than number of pages *M* of memory.
  - instead of partitioning *n* ways, use *M* – 1 partitions for s
  - Further partition the *M* – 1 partitions using a different hash function
  - Use same partitioning method on *r*
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Cost of Hash-Join

- ✓ If recursive partitioning is not required: cost of hash join is

  $3(b_r + b_s) + 4 * n_h$ block transfers +

  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks

- ✓ If recursive partitioning required:
  - number of passes required for partitioning build relation *s* to less than M blocks per partition is $\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:

    $2(b_r + b_s)\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil + b_r + b_s$ block transfers +

    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)\lceil \log_{\lfloor M/bb \rfloor - 1}(b_s/M) \rceil$ seeks

- ✓ If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

instructor ⋈ *teaches*

- ✓ Assume that memory size is 20 blocks
- ✓ $b_{instructor}$= 100 and $b_{teaches}$ = 400.
- ✓ *instructor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- ✓ Similarly, partition *teaches* into five partitions,each of size 80. This is also done in one pass.
- ✓ Therefore total cost, ignoring cost of writing partially filled blocks:
  - $3(100 + 400) = 1500$ block transfers +
    $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ seeks

Complex Joins

- ✓ Join with a conjunctive condition:

  $$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - 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

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

- ✓ Join with a disjunctive condition

  $$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

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

    $$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

“Big O” notation

Given two functions, *f* and *g*, say that “*f* is of order *g*” if

- there is a constant *c*, and
- a value x_0

such that

Apart from a fixed multiplicative constant, the function *g* is an

- upper bound on the function *f*
- valid for large values of its argument.

Notation: write to mean “*f* is of order *g*”.

Sometimes write to remind us the arguments.

Example: N-by-N matrix, N-by-1 vector, multiply

`Y = zeros(N,1);`	initialize space, c_1N		
`for i=1:N`	initialize "for" loop, c_2N		
`  Y(i) = 0.0;`	Scalar assignment, c_3		
`  for j=1:N`	initialize "for" loop, c_2N		N times
`    Y(i) = Y(i) + A(i,j)*x(j);`	c_4	N times	
`  end`	End of loop, return/exit, c_5		
`end`	End of loop, return/exit, c_5		

$$\begin{aligned} \mathit{Total} &= c_1N+c_2N+N(c_3+c_2N+N(c_4+c_5)+c_5) \\ &= (c_2+c_4+c_5)N^2 + (c_1+c_2+c_3+c_5)N \\ &= c_6N^2 + c_7N \end{aligned}$$