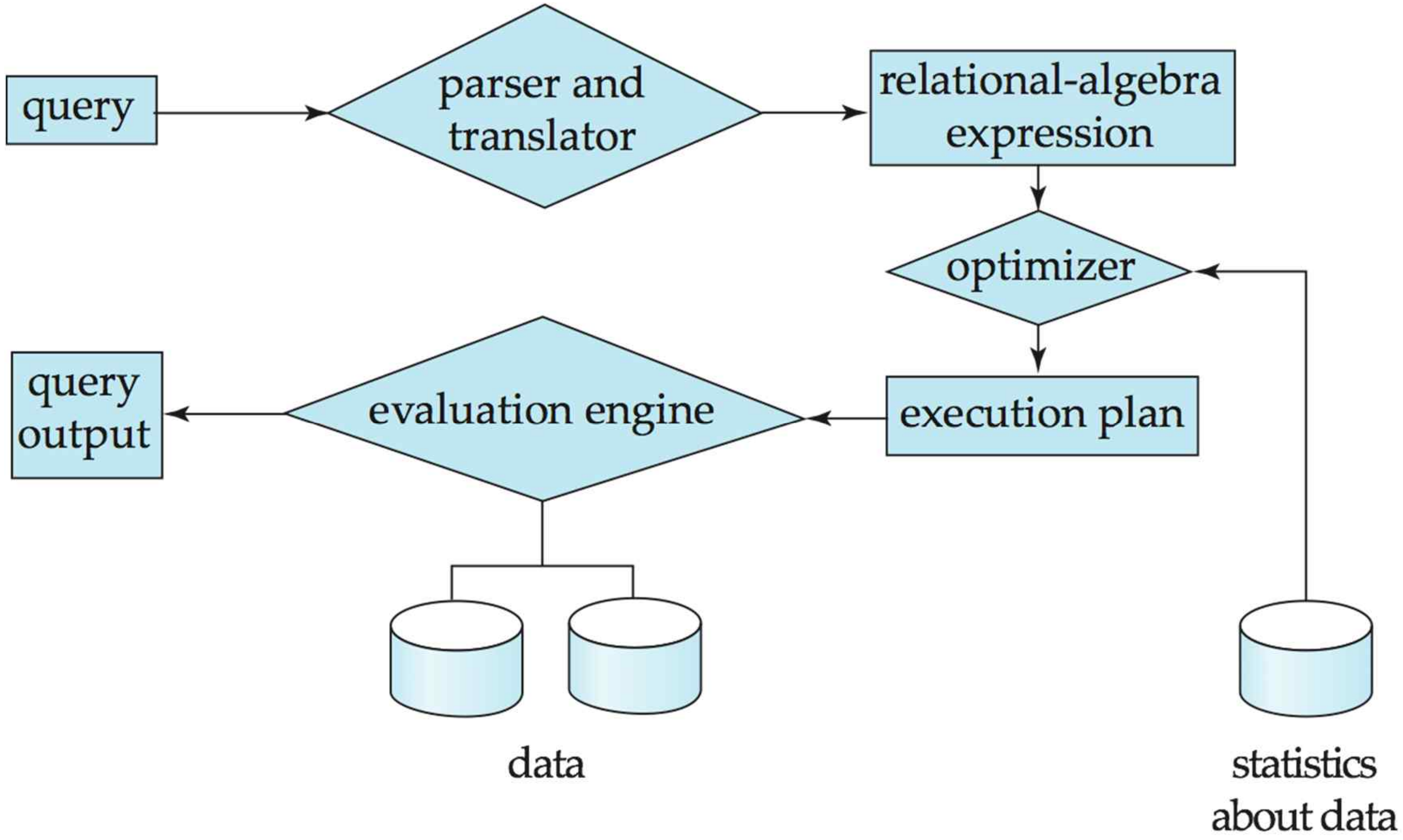
Chapter 12: Query Processing

Database System Concepts, 6th Ed.

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - Translate the query into its internal form
  - This is then translated into relational algebra
  - Parser checks syntax, verifies relations
- Optimization
  - Enumerate all possible query-evaluation plans
  - Compute the cost for the plans
  - Pick up the plan having the minimum cost
- 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

- There are more than one way to evaluate a query
  - 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
    - E.g., to find instructors with salary < 75000,
      - can use an index on *salary,*
      - or can perform complete relation scan and discard instructors with salary $\geq$ 75000
- **Query Optimization**
  - Amongst all equivalent evaluation plans choose the one with lowest cost
  - Cost is estimated using statistical information from the data dictionary
    - E.g., number of tuples in each relation, size of tuples, etc.

Basic Steps: Optimization (Cont.)

- In this chapter we study
  - 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
- In Chapter 13
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

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 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

Measures of Query Cost (Cont.)

- 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 seek (seek time + rotational latency)
  - 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 to writing output to disk in our cost formulae

Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition

- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = $b_r * t_T$ (block transfers) + 1 * t_S (seek)
    - b_r : number of blocks containing records from relation r
    - Extra seeks may be required, but we ignore this for simplicity
  - If selection is on a key attribute, can stop on finding record
    - cost = (b_r/2) block transfers + 1 seek
  - 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 consecutively

Selections Using Indices

- **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* = $(h_i + 1) * (t_T + t_S)$
    - h_i: the 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* = $h_i * (t_T + t_S) + t_S + t_T * b$

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!
    - each record may be on a different block
      - one block access for each retrieved record

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 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)**.
  - 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, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper if many records are to be fetched!

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 and algorithms A1 through A7 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

Sorting

- Sorting of data is important in DBMS for two reasons:
  - SQL queries can specify that the output be sorted
  - Several of the relational operations (e.g., joins) can be implemented efficiently if the input relations are first sorted
- We may 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

M: memory size (in blocks) available for sorting

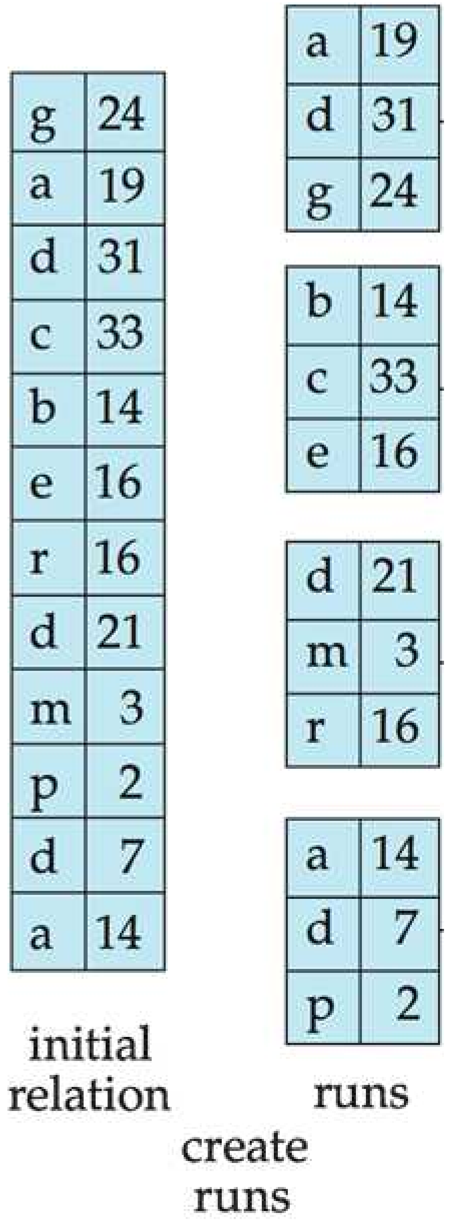
1. **Create sorted runs**

Let *i* be 0 initially.

Repeatedly do the following
till the end of the relation:

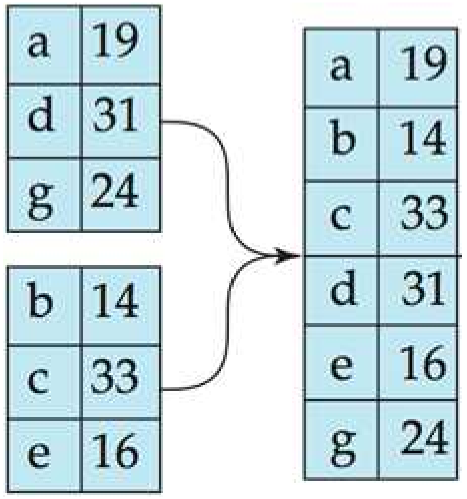
(a) Read *M* blocks of relation into memory
(b) Sort the in-memory blocks
(c) Write sorted data to run R_i; increment *I*

Let the final value of *i* be *N* (the number of runs).



External Sort-Merge (Cont.)

2. **Merge the runs (N-way merge)**. (Assume for now 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:

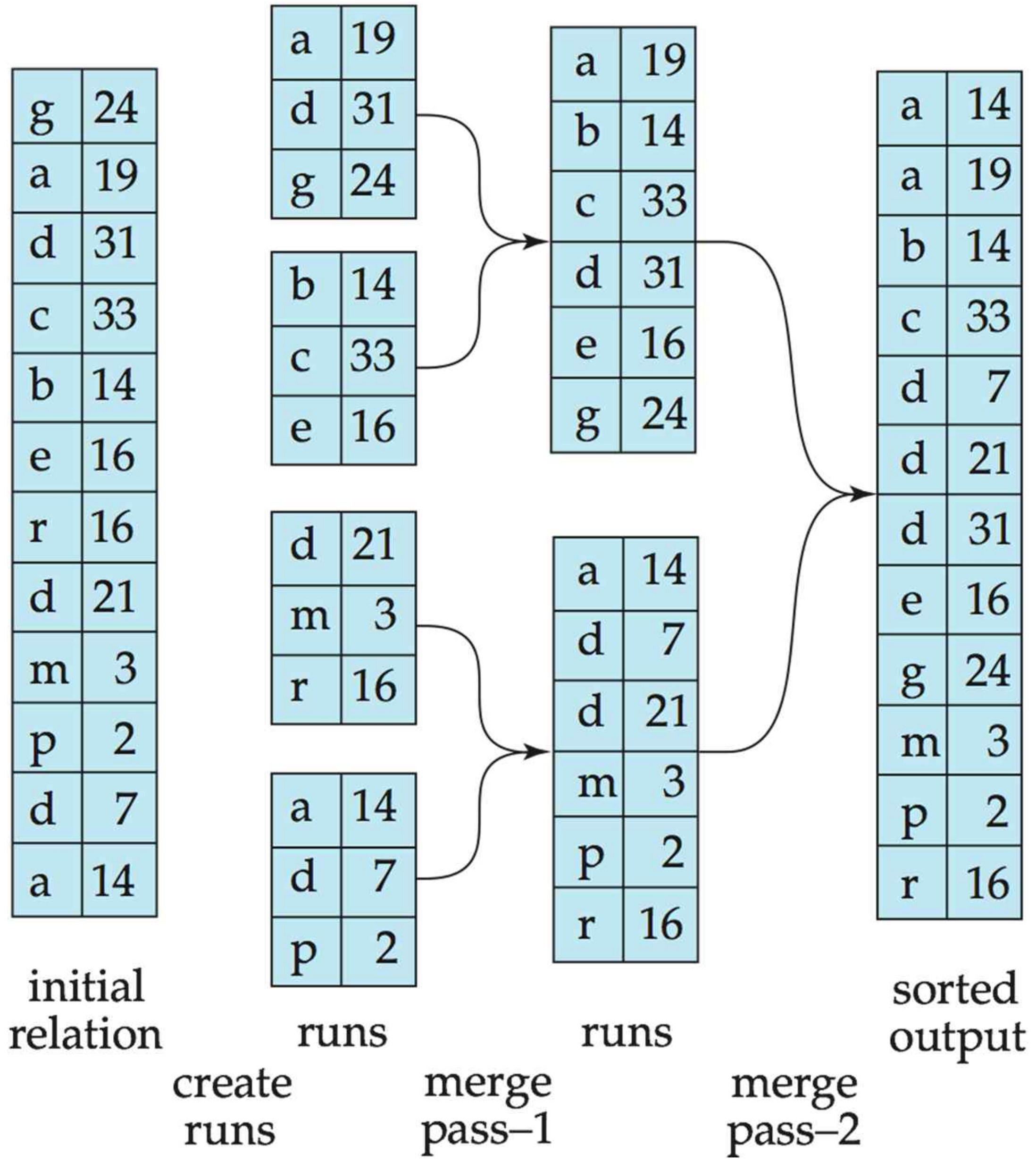


External Sort-Merge (Cont.)

2. **Merge the runs (*N* ≥ *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

- M = 3
- Only one tuple fits in a block



External Merge Sort – Cost Analysis

- Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$
  - b_r / M: the initial number of runs
- Block transfers
  - For initial run creation as well as in each pass: $2b_r$
  - Thus total number of block transfers for external sorting:
    $b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$
    - We don't count final write cost for all operations
- Seeks
  - During run generation: 1 seek to read and 1 seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Buffer size: b_b (read/write b_b blocks at a time)
    - 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:
      $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil -1)$

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 – Example

student

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Nested-Loop Join – Cost Analysis

- Worst case
  - There is memory only to hold one block of each relation

    Cost = $n_r * b_s + b_r$ block transfers and $n_r + b_r$ seeks
    - n_r, n_s : number of record in R and S
    - b_r, b_s: number of disk blocks in R and S
    - Extra seeks may be required, but we ignore this for simplicity
- Best case
  - Smaller relation fits entirely in memory – use that as the inner relation

    Cost = $b_r + b_s$ block transfers and 2 seeks
- Example
  - 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
    - Cost estimate will be 500 block transfers

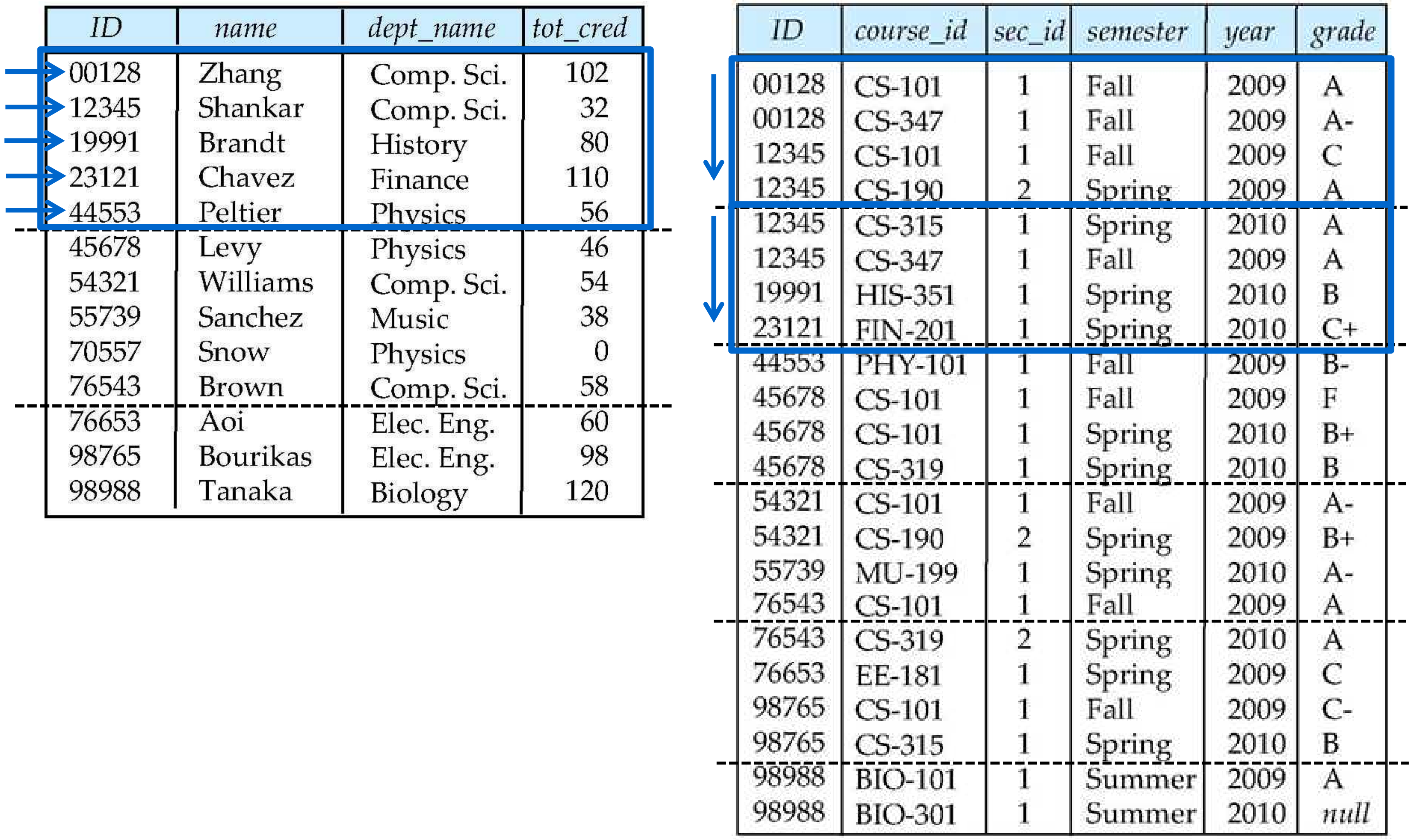
Block Nested-Loop Join

- 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
```

- **Won Kim's join method**
  - One chapter of '80 PhD Thesis at Univ of Illinois at Urbana-Champaign

Block Nested-Loop Join – Example



student

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Block Nested-Loop Join (Cont.)

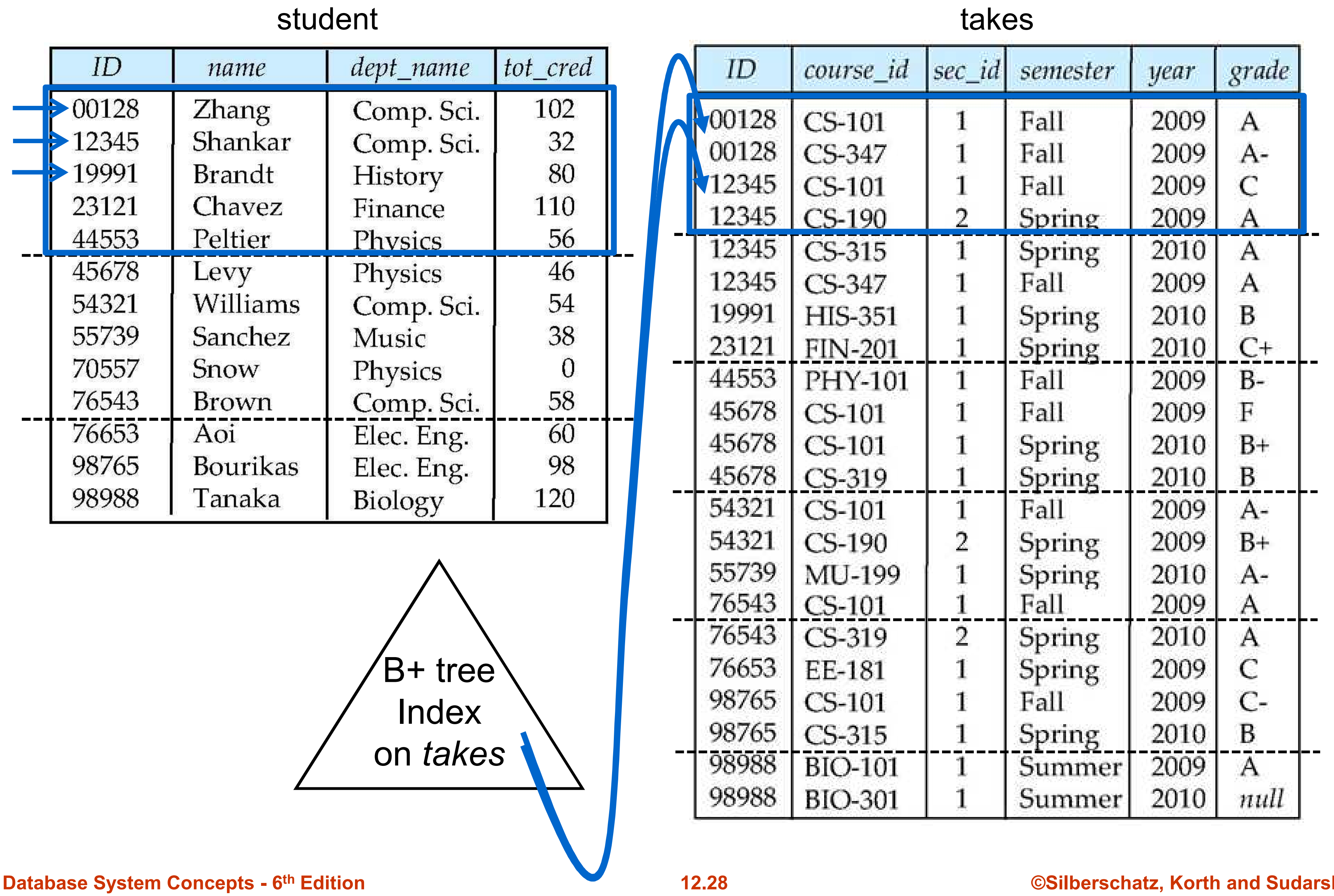
- Worst case estimate: $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks

- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use *M* - 2 disk blocks as blocking unit for outer relations, where *M* = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
  - If equi-join attribute forms a key or 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
    - Can construct an index just to compute a join
- For each tuple t_r in the outer relation *r,* use the index (B+ tree) 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

Indexed Nested-Loop Join – Example



student

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

takes

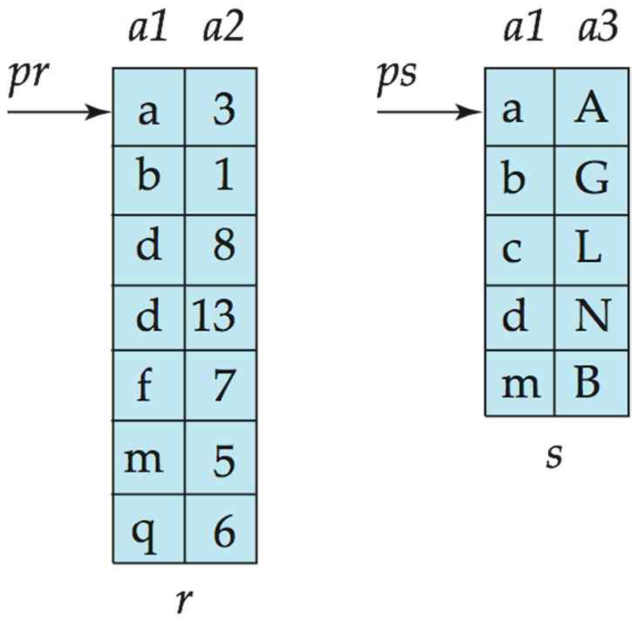
ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Example of Nested-Loop Join Costs

- Example: compute *student* ⋈ *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)
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

- 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

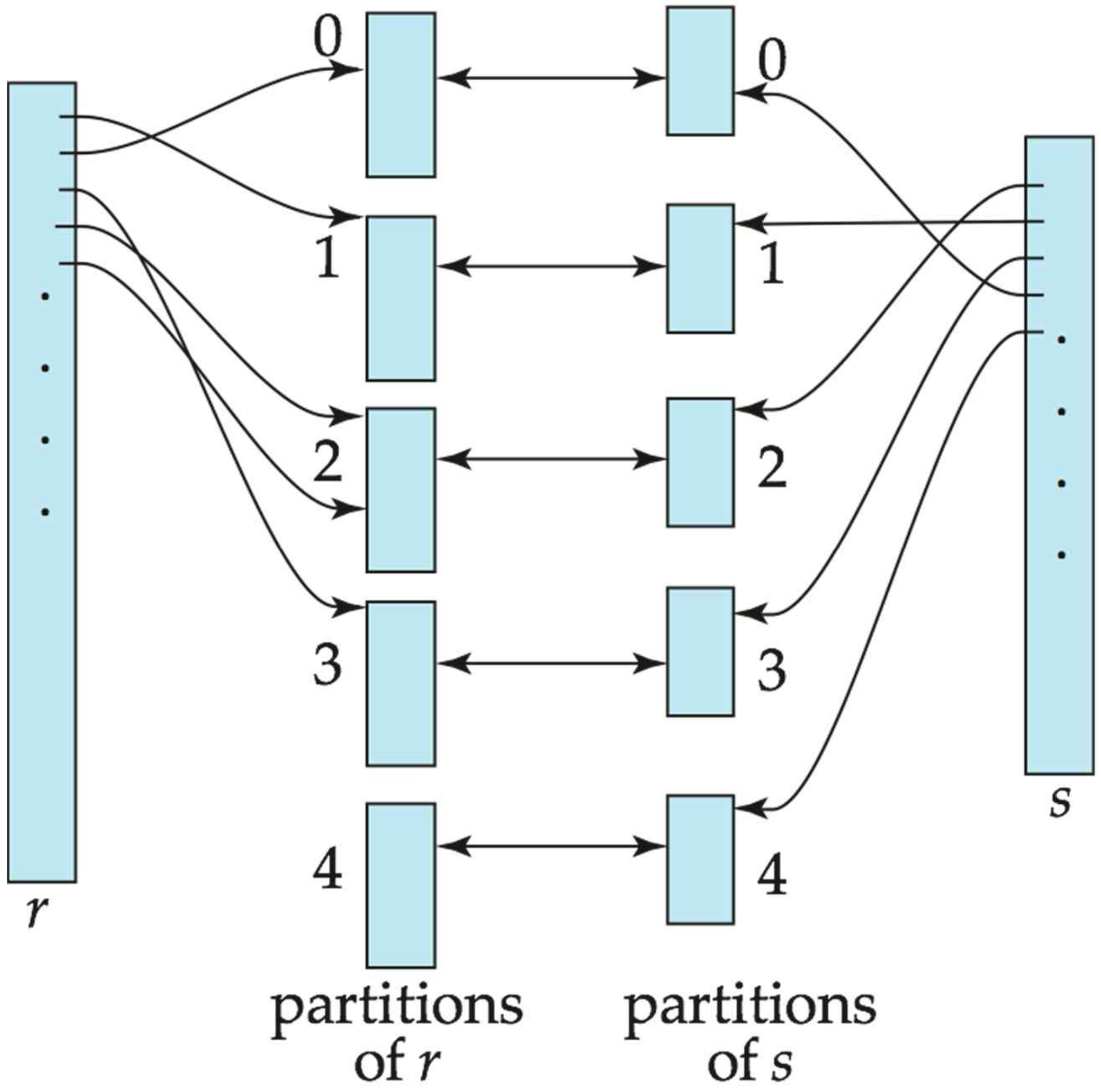
  + the cost of sorting if relations are unsorted

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
  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$
  - $r_0,, r_1. \ldots, r_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$
- *r* tuples in r_i need only to be compared with *s* tuples in s_i

(*Note:* In book, r_i is denoted as H_{ri}, s_i is denoted as H_{si}, and *n* is denoted as $n_{h.}$)

Hash-Join (Cont.)



Hash-Join Algorithm

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**

Handling of Hash Overflows

- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory
- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition s_i is further partitioned using different hash function.
  - Partition r_i must be similarly partitioned.
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- Cost of hash join
  $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If the entire build input can be kept in main memory, no partitioning is required
  - Cost estimate goes down to $b_r + b_s$
- Example: *instructor* ⋈ *teaches*
  - Memory size = 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

Other Operations

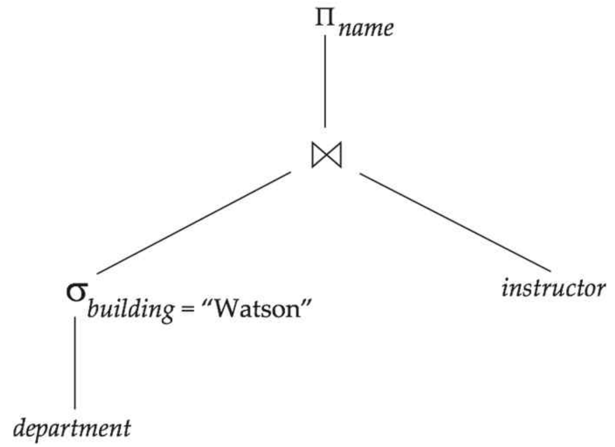
- **Duplicate elimination** can be implemented via hashing or sorting
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge
  - Hashing is similar – duplicates will come into the same bucket
- **Projection:**
  - Perform projection on each tuple
  - Followed by duplicate elimination
- **Aggregation** (count, min, max, sum, and avg):
  - Can be implemented in a manner similar to duplicate elimination
- **Set operations** ($\cup$, $\cap$ and —):
  - Can either use variant of merge-join after sorting, or variant of hash-join

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

Materialization

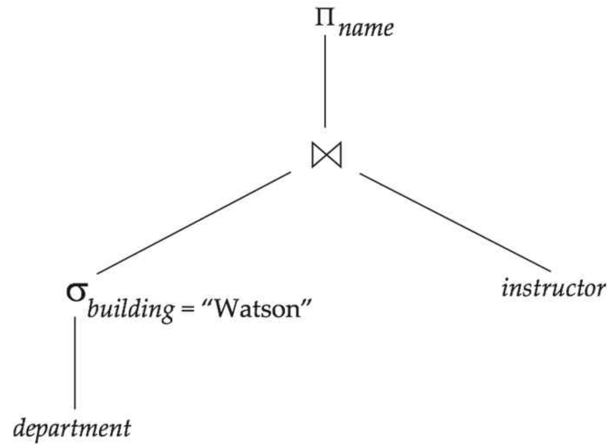
- **Materialized evaluation**
  - Evaluate one operation at a time, starting at the lowest-level
  - Use intermediate results materialized into temporary relations to evaluate next-level operations
- E.g.,
  - compute and store $\sigma_{building=\text{“Watson”}}$(department)
  - then compute the store its join with *instructor,*
  - and finally compute the projection on *name*



- 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
  - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

Pipelining

- **Pipelined evaluation**
  - evaluate several operations simultaneously,
  - passing the results of one operation on to the next
- E.g.,
  - don't store result of $\sigma_{building=\text{“Watson”}}$(department)
  - instead, pass tuples directly to the join.
  - Similarly, don't 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
- Pipelines can be executed in two ways:
  - **demand driven** and **producer driven**

Pipelining (Cont.)

- 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 name: **pull** and **push** models of pipelining

End of Chapter 12