Topic 8: Query Processing and Optimization (Chapters 15, 16)

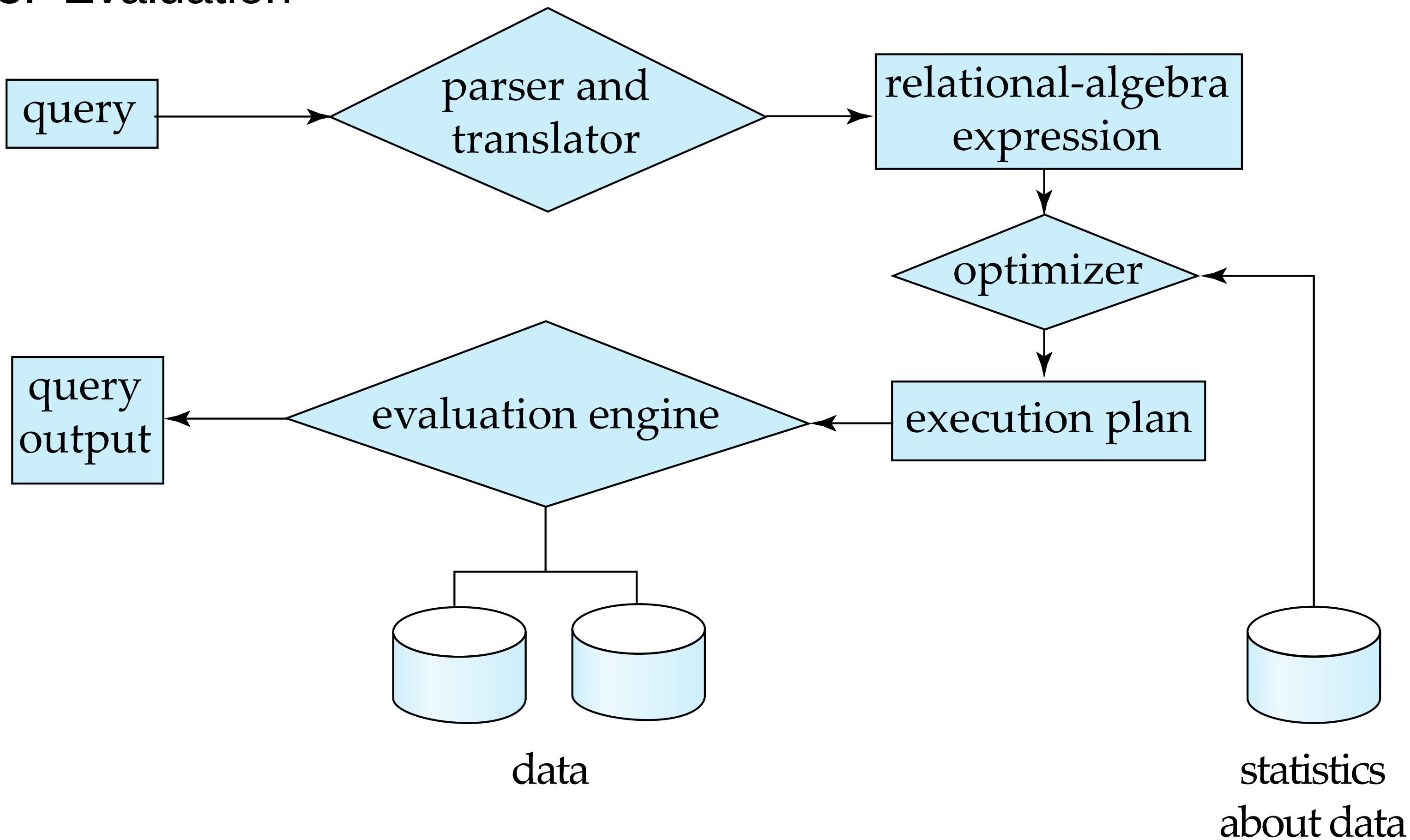
(Modified for CS 4513)

Topic 8: Query Processing and Optimization

- Basic Steps in Query Processing
- Transformation of Relational Expressions
- Estimation of Query Processing Cost
- Join Strategies

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
- 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

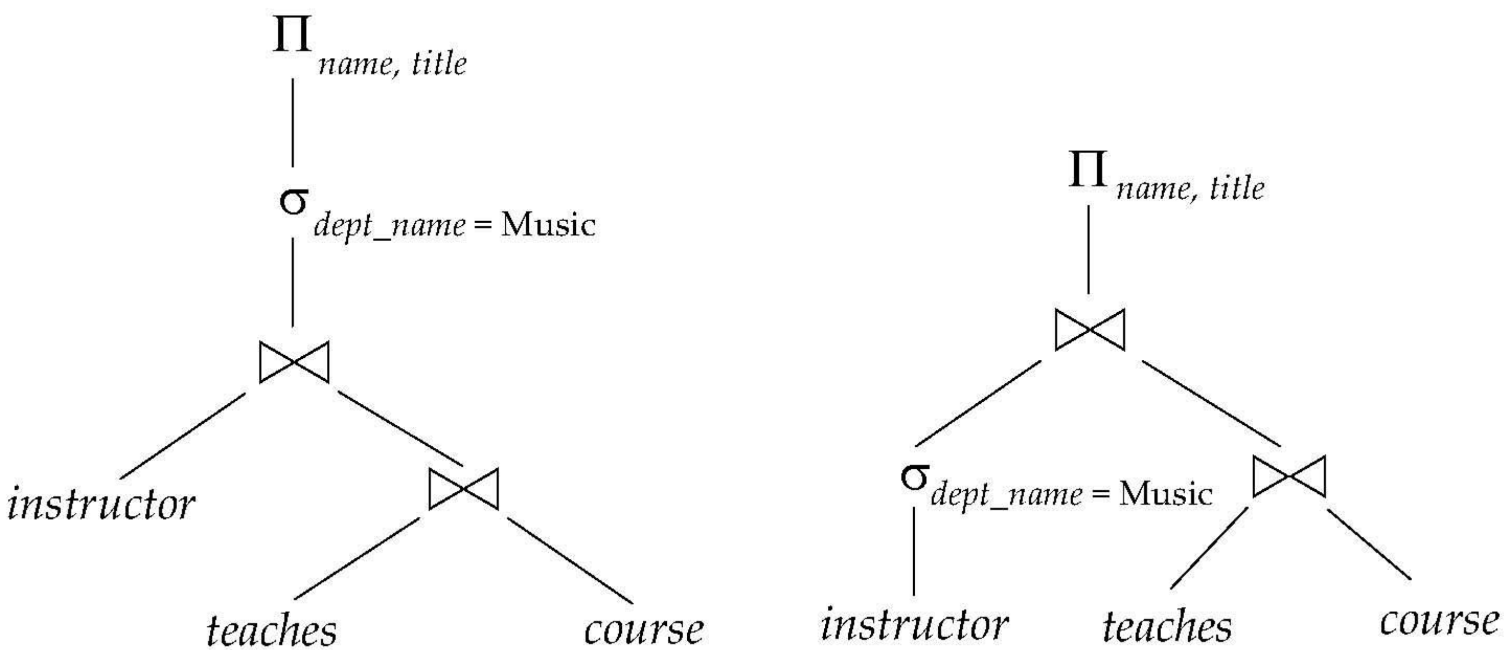
- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **execution plan or 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

Basic Steps: Optimization (Cont.)

- **Query Optimization**: has two phases:
  - Phase 1: find an equivalent expression to the given query expression that is more efficient to execute
  - Phase 2: select a detailed strategy for processing the query; choose the one with the lowest cost
    - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.

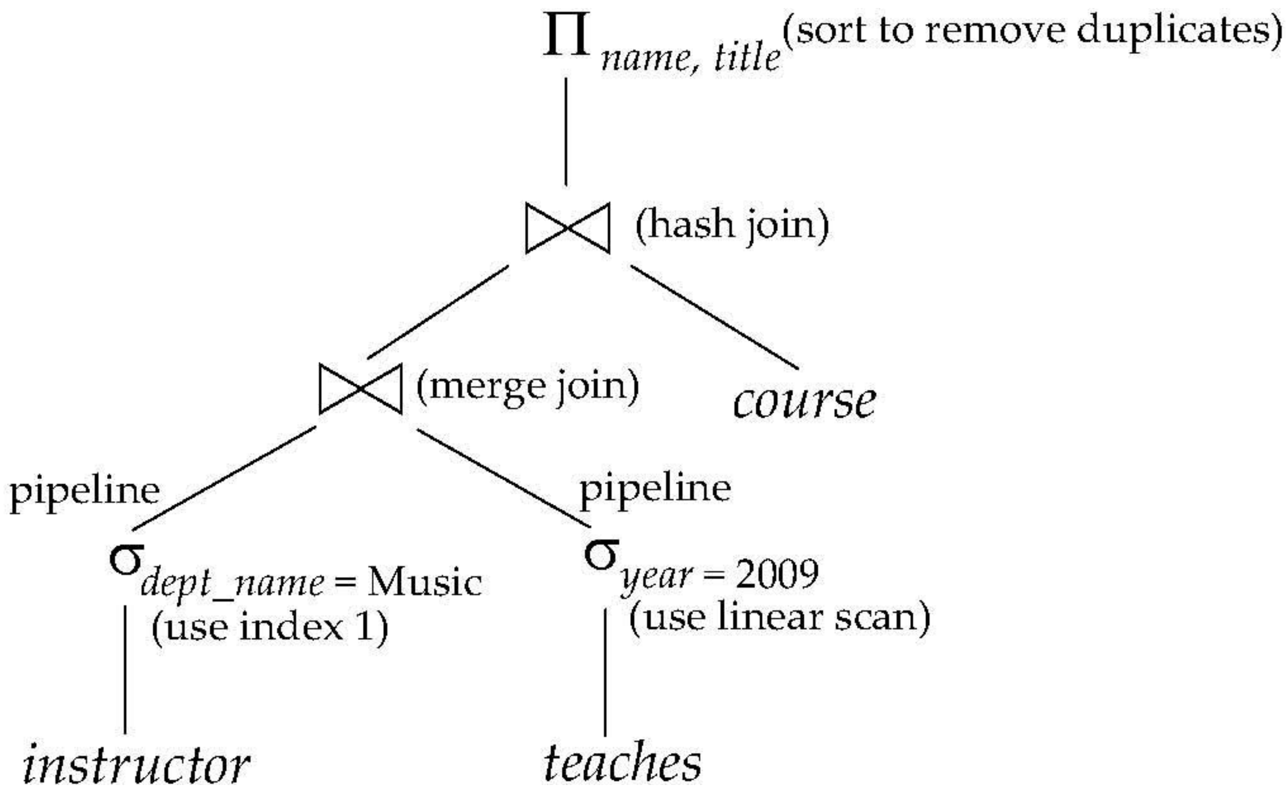
Basic Steps: Optimization (Cont.)

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



Basic Steps: Optimization (Cont.)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Basic Steps: Optimization (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Generating Equivalent Expressions

Equivalence of Expressions (1)

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant
  - we do not care if they generate different results on databases that violate integrity constraints
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa
- Goal: find an equivalent expression that gives fewer number of tuples to be accessed to produce a query answer

Equivalence of Expressions (cont.)

- a) Selection Operation:
  - Transformation Rule 1: Perform selection as early as possible
  - Example:
    - Customer (custname, street, customercity)
    - Deposit (branchname, accnumber, custname, balance)
    - Branch (branchname, assests, branchcity)
    - *Query: find assets and name of all banks which have depositors living in Norman*
    - Relational algebra expression:
    - An equivalent relational algebra expression:

Equivalence of Expressions (cont.)

- a) Selection operation (cont):
  - Transformation Rule 2:
    - Replace $\sigma_{\theta_1 \wedge \theta_2}(r) = \sigma_{\theta_1}(\sigma_{\theta_2}(r))$

Equivalence of Expressions (cont.)

- b) Projection Operation:
  - Transformation Rule 1: Perform projections early

    $\Pi_A(r \text{ x } s) = \Pi_A(r) \text{ x } \Pi_A(s)$

Equivalence of Expressions (cont.)

- c) Join Operation:
  - Transformation Rule 1: Choose the one that produces fewer number of tuples in intermediate results
  - For all relations $r_1, r_2,$ and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

(Join Associativity)

  - If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$$

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}(course)$ first, and join result with

$$\sigma_{dept\_name=\text{"Music"}}(instructor)$$

but the result of the first join is likely to be a large relation.

- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute

$$\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches$$

first.

Equivalence Rules (Cont.)

- d) Other Operations:
  - Transformation (equivalence) rules:
    - When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

    - When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

    - Read section 16.2.1 “Equivalence Rules” for other rules (see the next five slides)

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_{\theta} E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E_2))$$

Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$:

   $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

   $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   - (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

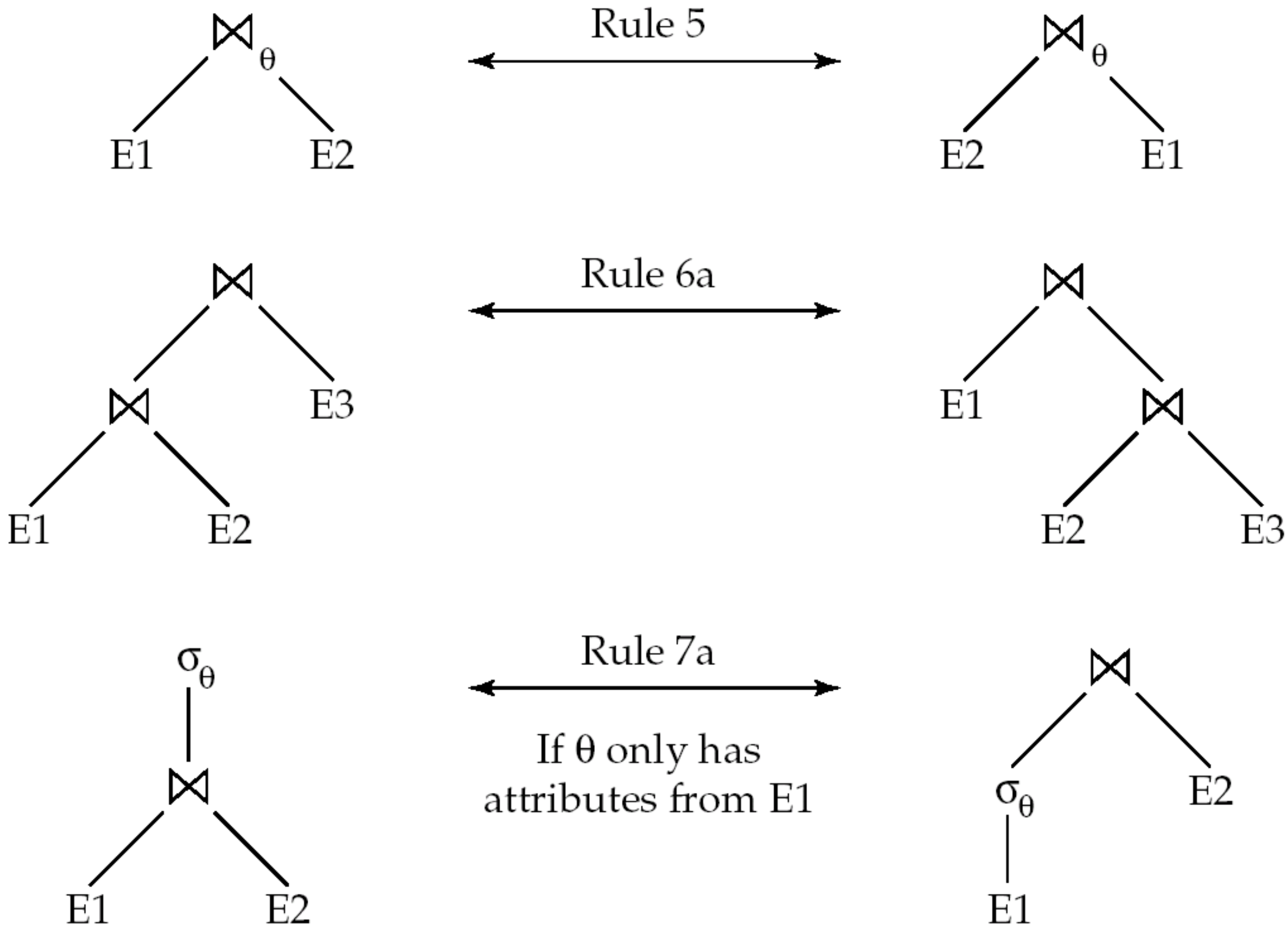
Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Pictorial Depiction of Equivalence Rules (Query (Parse) Tree)

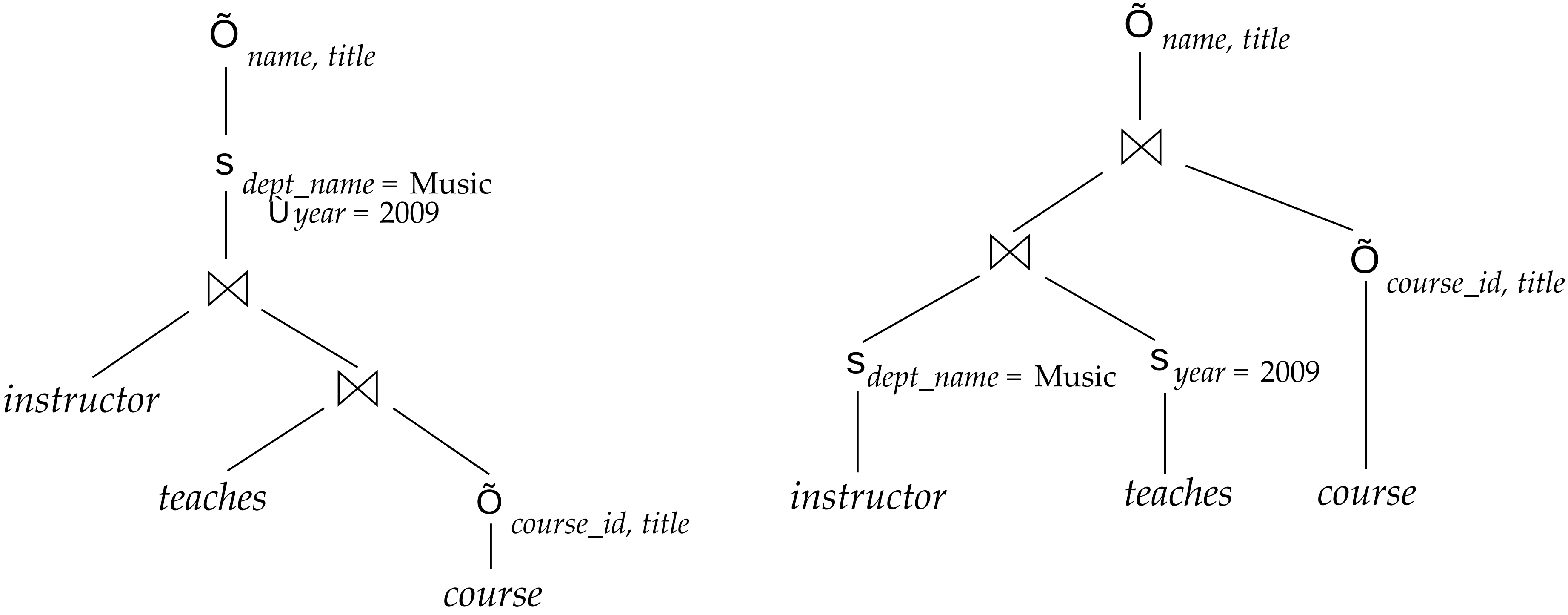


Example with Multiple Transformations

- Query: Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"} \wedge year = 2009}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using join associatively (Rule 6a):
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"} \wedge year = 2009}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course)))$
- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression (using rule 7a)

$$\sigma_{dept\_name = \text{"Music"}}(instructor) \bowtie \sigma_{year = 2009}(teaches)$$

Query (Parse) Tree



(a) Initial expression tree

(b) Tree after multiple transformations

Estimation of Query Processing Cost

Statistical Information for Cost Estimation

- In order to be able to choose a query processing strategy, a DBMS may store the following statistics for each relation r:
  - n_r: number of tuples in a relation *r.*
  - b_r: number of blocks containing tuples of *r.*
  - l_r: size of a tuple of *r.*
  - f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
  - *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
  - If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Cartesian Product Size Estimation

- ***r x s***
  - n_r and n_s allow accurate estimation of the size of a cartesian product
  - has $n_r * n_s$ tuples, each tuple is of ($l_r + l_s$) bytes

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - Assume each distinct value of A appears in a column with equal probability (uniform distribution)
  - $n_r / V(A,r)$: number of records that will satisfy the selection

Estimation of the Size of Joins

- The cartesian product $r \times s$ contains $n_r . n_s$ tuples; each tuple occupies $l_r + l_s$ bytes.
- If $R \cap S = \emptyset$, then size of $r \bowtie s$ is the same as size of $r \times s$.
- If $R \cap S = K_1$ a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s: size of $r \bowtie s$ <= size of s
- If $R \cap S = K_2$ a key for S, then a tuple of r will join with at most one tuple from s: size of $r \bowtie s$ <= size of r

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ not a key for R or S.
  - Assume uniform distribution of distinct values of A
  - One tuple in r will join with $(n_s / V(A, s))$ tuples in s
  - All tuples in r will join with $\frac{n_r * n_s}{V(A,s)}$ tuples in s
  - This means the estimated size of r $\bowtie$ s is

$$\frac{n_r * n_s}{V(A,s)}$$

  - Similarly, the estimated size of $s \bowtie r$ is

$$\frac{n_r * n_s}{V(A,r)}$$

  - Choose the lower of these two estimates

Join Strategies

- Read Section 12.3 (Chapter 12) “Magnetic Disk”.

Join Operation

- Several different algorithms to implement join operations:
  - Nested-loop join
  - Block nested-loop join
  - Merge-join
  - etc.
- Choice based on cost estimate
- *Cost estimate = number of disk block transfers +*
  number of disk seeks + ….
- Examples use the following information:
  - Number of records of *student*: 5,000 *takes*: 10,000
  - Number of blocks of *student*: 100 *takes*: 400
  - Assume all records in each relation are physically stored together on disk

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 (Cont.)

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

  $n_r * b_s + b_r$ block transfers, plus

  $n_r + b_r$ disk 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:
    - 5,000 ∗ 400 + 100 = 2,000,100 block transfers and
    - 5,000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10,000 ∗ 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 $t_r \bullet t_s$ to the result.
            end
        end
    end
end

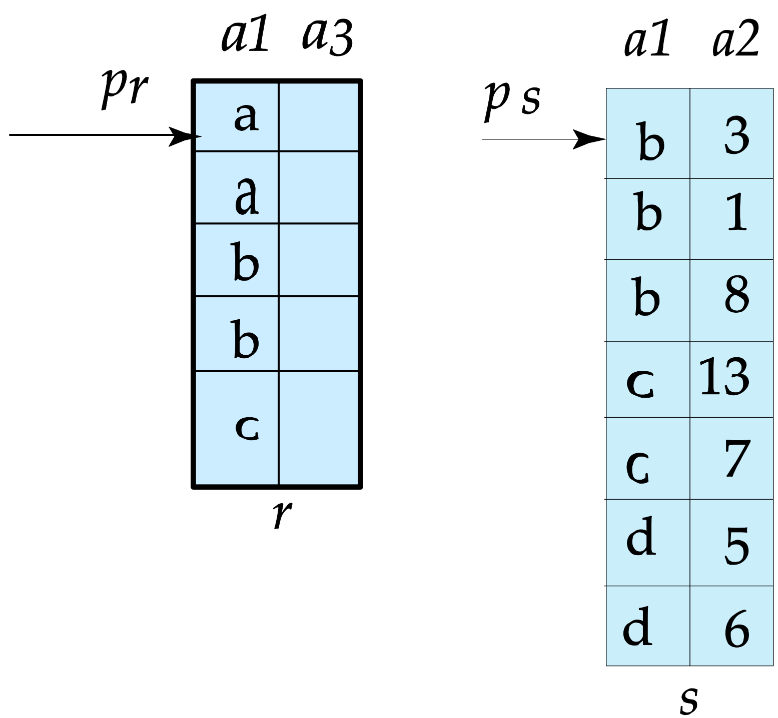
Block Nested-Loop Join (Cont.)

- Worst case estimate: the main memory can hold only one block for each relation:
  - Each block in the inner relation *s* is read once for each *block* in the outer relation
  - *Estimated Cost* = $b_r * b_s + b_r$ *block transfers* + $2 * b_r$ *seeks*

- Best case: the main memory can hold two entire relations simultaneously
  - Each scan of the inner relation *s* requires 1 seek
  - The scan of the outer relation *r* requires 1 seek
  - *Estimated Cost* = $b_r + b_s$ *block transfers* + *2 seeks.*

Merge-Join

- Assumption: each relation is sorted on the join attribute
- Can be used only for equi-joins and natural joins
- Example: r(R), s(S), $R \cap S$ = {a1} and a1 is sorted
- Merge the sorted relations *r* and *s* to join them
  - Detailed algorithm in book



Merge-Join (Cont.)

- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Assuming b_b buffer blocks (in the main memory) are allocated for each relation
- The estimated 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.

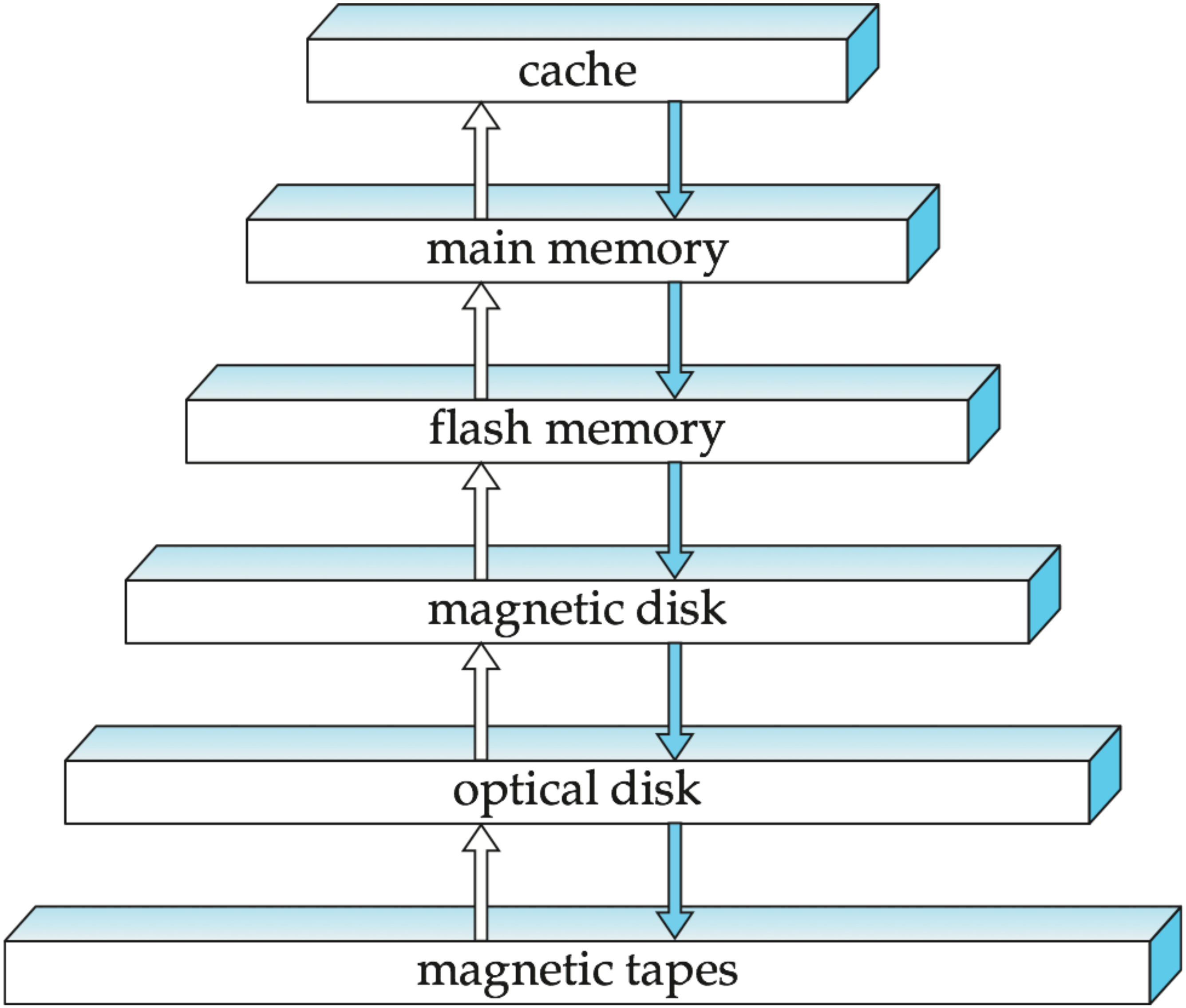
End of Topic 8

Additional Slides from Chapter 12: Physical Storage Systems

Database System Concepts

(Modified for CS 4513)

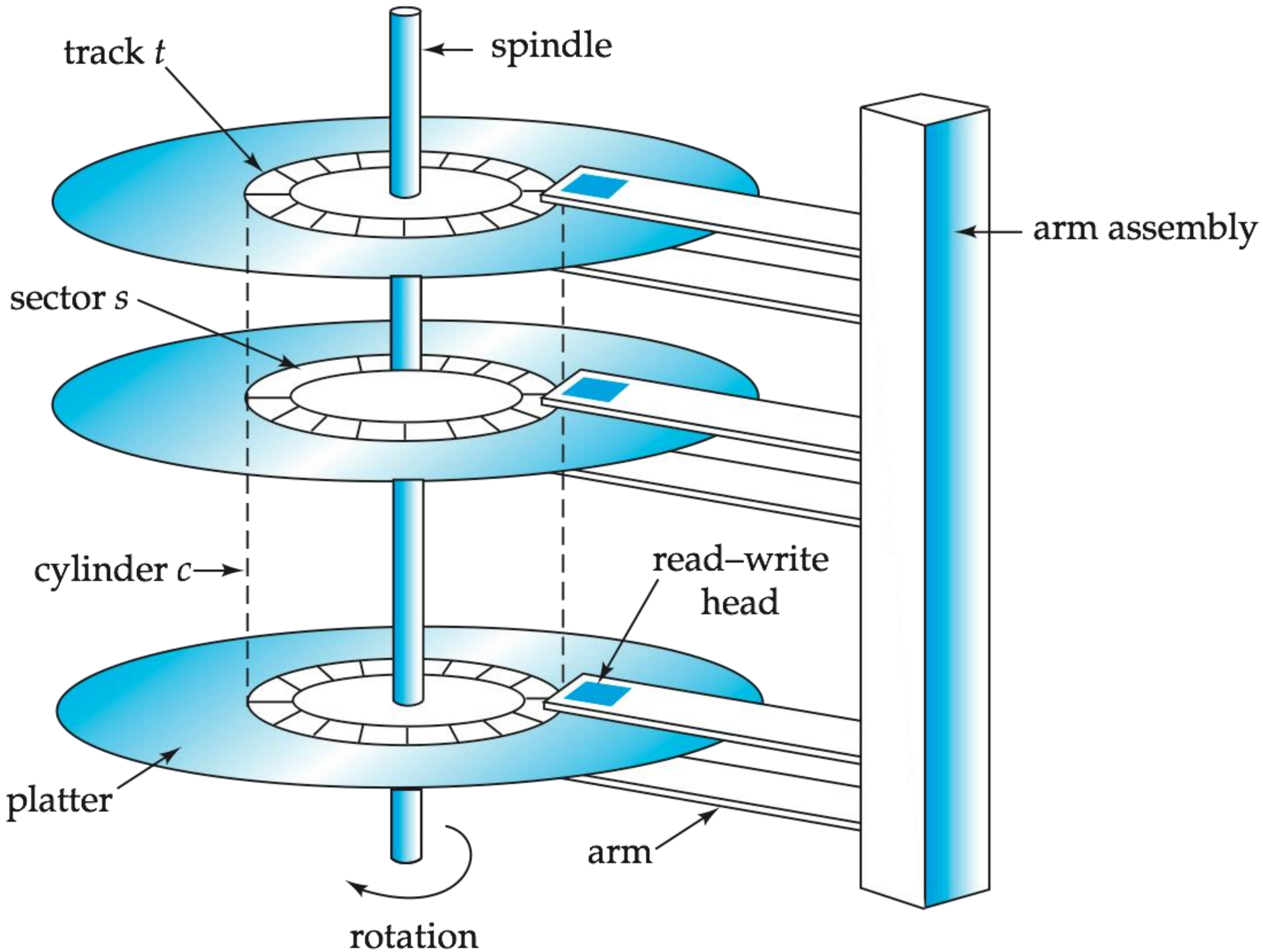
Storage Hierarchy



Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage
  - Magnetic tape
    - Sequential access, 1 to 12 TB capacity
    - A few drives with many tapes
    - Juke boxes with petabytes (1000’s of TB) of storage

Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

Magnetic Disks

- **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors.**
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.
- **Cylinder** *i* consists of i^{th} track of all the platters

Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
  - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
  - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
- **Disk controller** – interfaces between the computer system and the disk drive hardware.
  - accepts high-level commands to read or write a sector
  - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
  - Computes and attaches **checksums** to each sector to verify that data is read back correctly
    - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
  - Ensures successful writing by reading back sector after writing it
  - Performs remapping of bad sectors