Principles of Database Systems (CS307)

Lecture 14 part 1: Query Processing

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

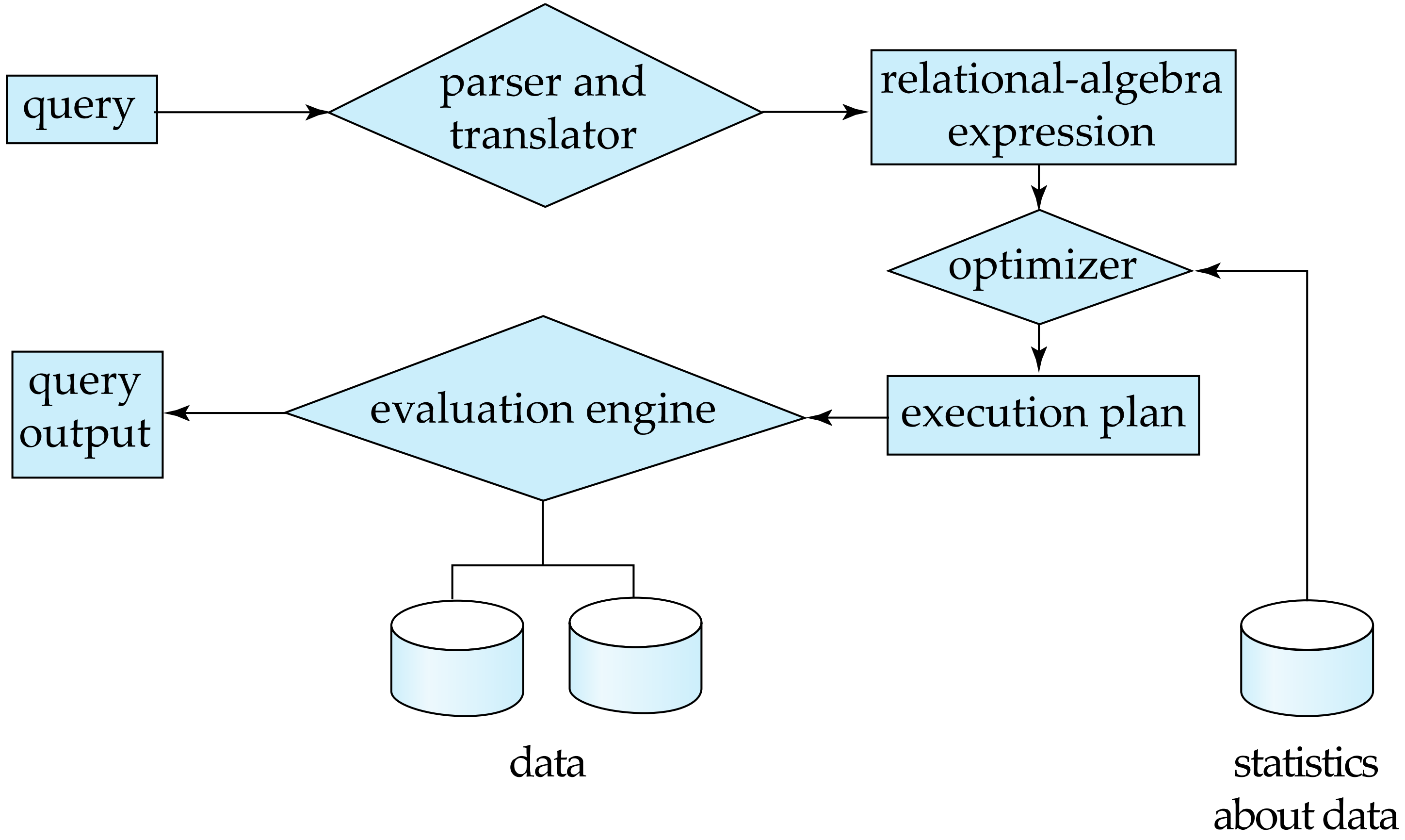
- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7^{th} Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Announcements

- Final exam will be on Jan. 7, 2026 (Wed), 14:00-16:00, 三教106/107

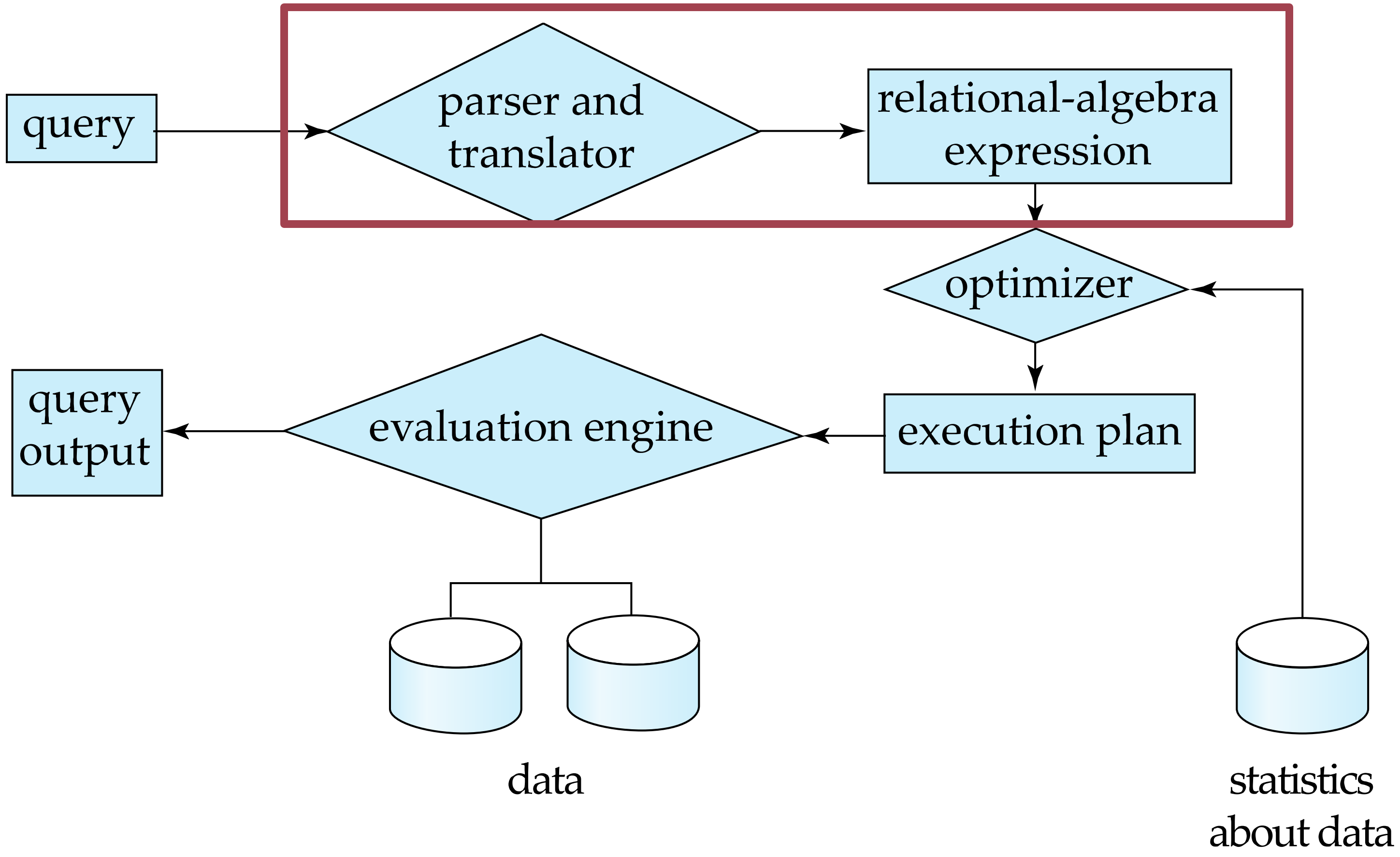
Basic Steps in Query Processing

- Parsing and Translation
- Optimization
- Evaluation



Basic Steps in Query Processing

- Parsing and Translation
  - Translate the query into its internal form
    - The internal form is then translated into relational algebra
  - Parser checks syntax and verifies relations



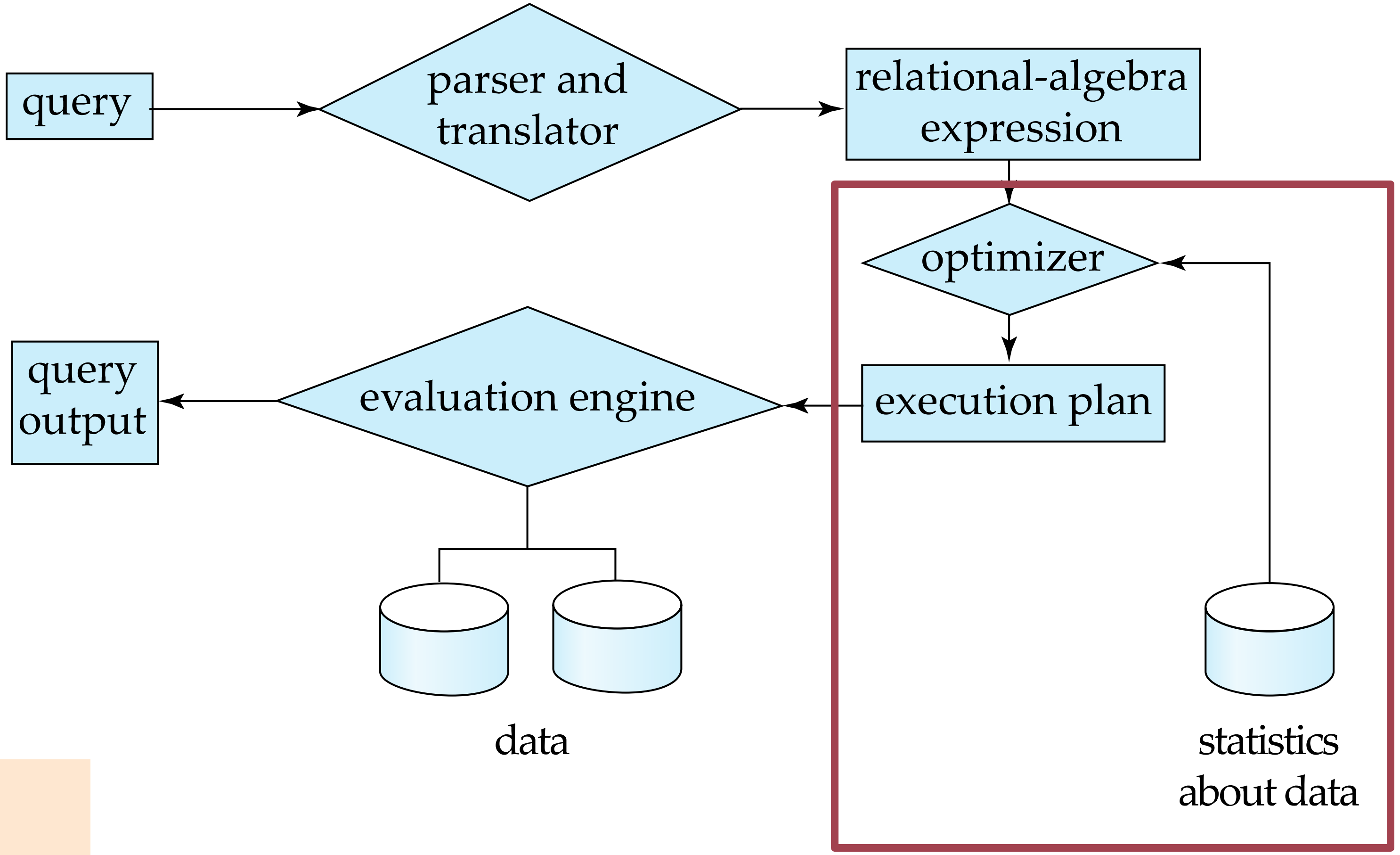
Basic Steps in Query Processing

- Optimization
  - A relational algebra expression may have many equivalent expressions
  - E.g.,

$$\sigma_{salary<75000}(\Pi_{salary}(instructor))$$

is equivalent to

$$\Pi_{salary}(\sigma_{salary<75000}(instructor))$$



But the number of rows involved in the projection operation may be (significantly) smaller in the second expression

Basic Steps in Query Processing

- Optimization
  - A relational algebra expression may have many equivalent expressions
    - … and each relational algebra operation can be evaluated using one of several different algorithms

  - *Correspondingly, a relational-algebra expression can be evaluated in many ways*

Basic Steps in Query Processing

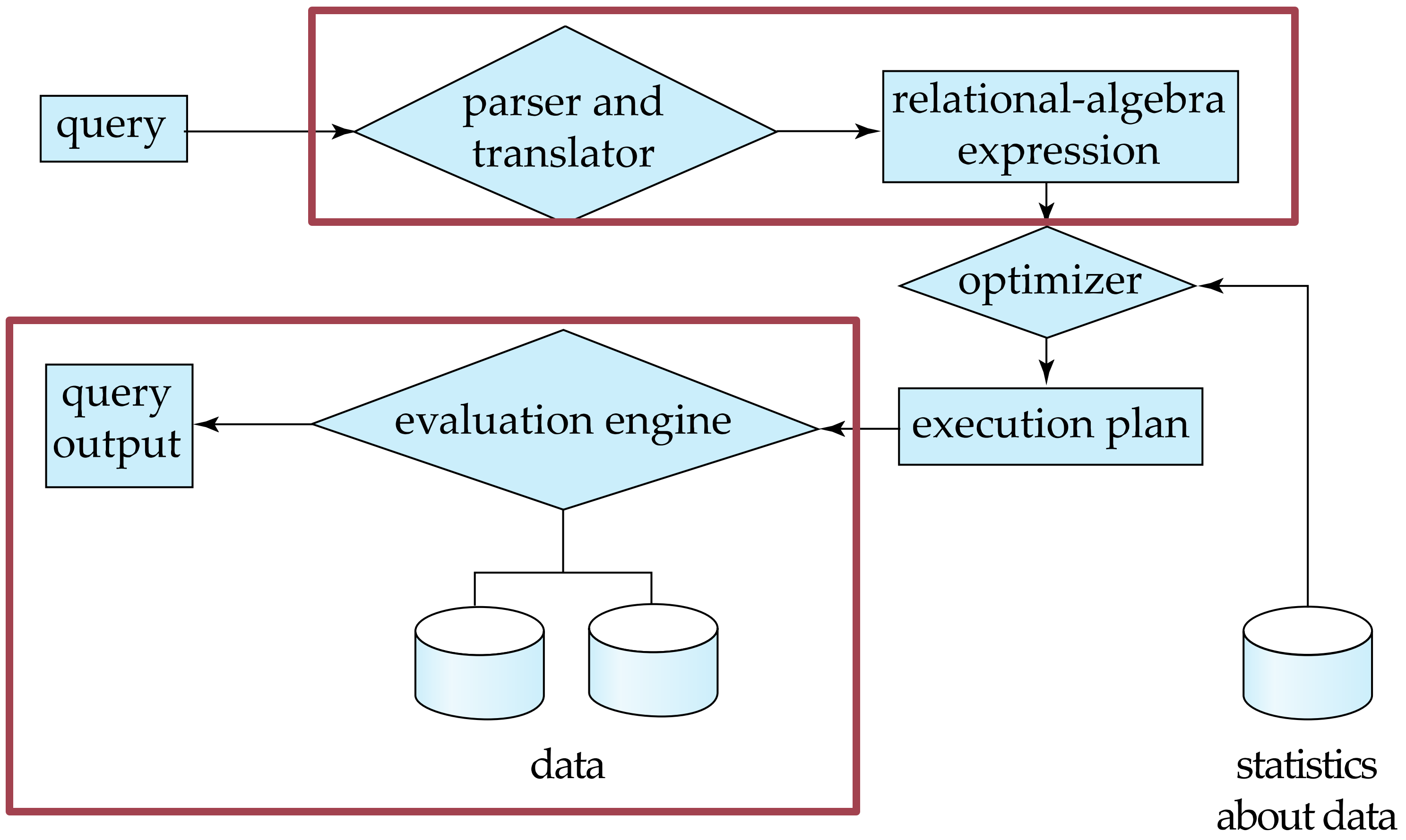
- Optimization
  - **Evaluation Plan**: Annotated expression specifying detailed evaluation strategy
  - E.g.,:
    - Use an index on `salary` to find instructors with `salary<75000`
    - Or perform complete relation scan and discard instructors with `salary<75000`

Query Optimization: Choose the one with the lowest cost among all equivalent evaluation plans

- Cost can be estimated using statistical information from the database catalog
  - E.g., Number of tuples in each relation, size of tuples, etc.

Basic Steps in Query Processing

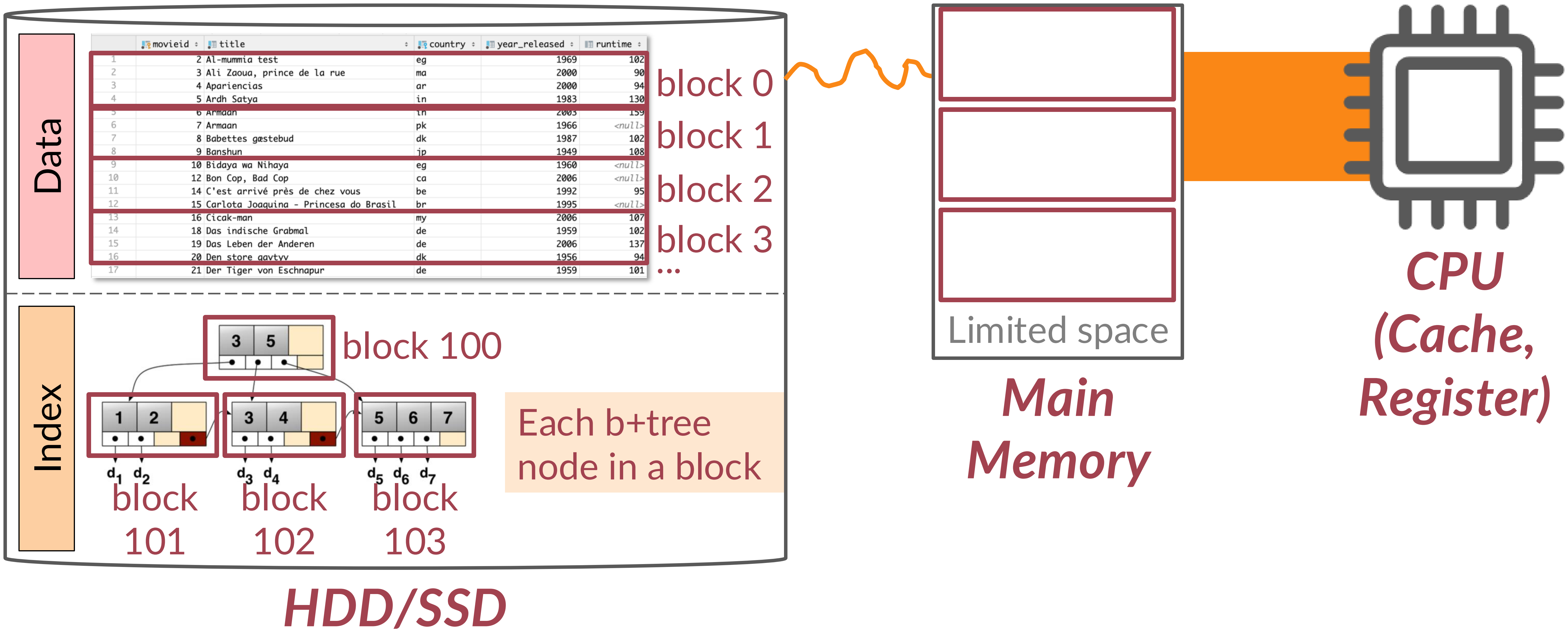
- Evaluation
  - The query-execution engine <u>takes a</u> <u>query-evaluation plan</u>, <u>executes</u> that plan, and <u>returns the answers</u> to the query



Cost of Query

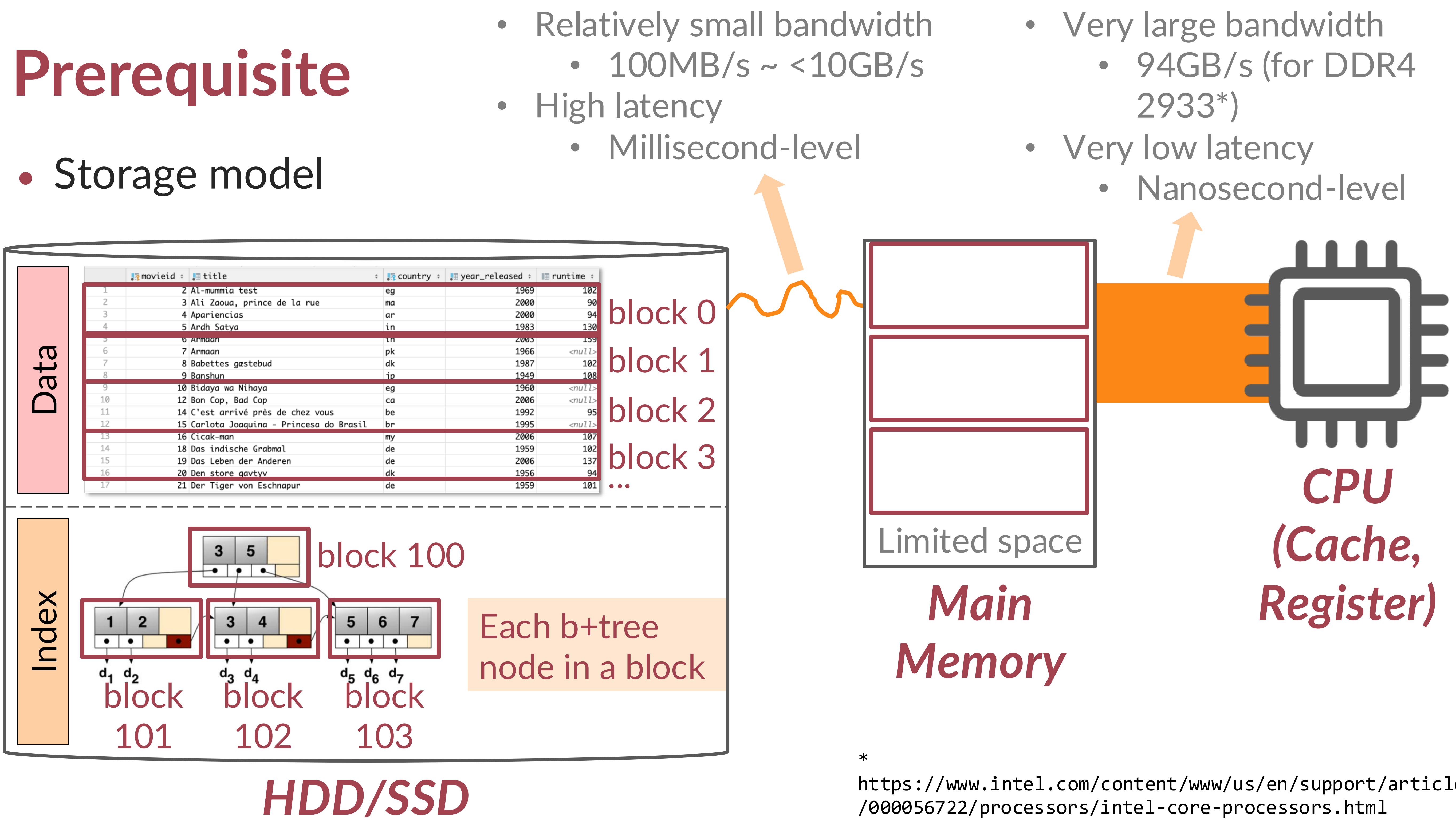
Prerequisite

- Storage model



Prerequisite

- Storage model



* https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html

Prerequisite

- Measuring query cost
  - Disk cost can be estimated as:
    - Number of seeks * average-seek-cost
    - Number of blocks read * average-block-read-cost
    - Number of blocks written * average-block-write-cost
  - For simplicity, we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
    - t_T – time to transfer one block
      - Assuming for simplicity that write cost is same as read cost
    - t_S – time for one seek
  - E.g., cost for b block transfers plus S seeks

    $b * t_T + S * t_S$

Prerequisite

- Measuring query cost
  - t_S and t_T depend on where data is stored. With 4 KB blocks:
    - High end magnetic disk: t_S = 4 msec and t_T =0.1 msec
    - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB
  - Required data may be buffer resident already, avoiding disk I/O
    - But hard to take into account for cost estimation
  - Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
    - But more optimistic estimates are used in practice
  - We ignore CPU costs for simplicity
    - Real systems do take CPU cost into account
    - Network costs must be considered for parallel systems

Overview

- Selection
- Joining

Selection Operation

- Let's start from this simple query:

```
select * from movies where [CONDITION];
```

  - If you are the designer of the database engine, what do you think is the best way to fulfill this requirement?
  - Two factors to consider:
    - What comparison is it in the CONDITION (equality / comparison)?
    - Does the column involved in the CONDITION have an index?

Basic Linear Scan

- Linear Search (displayed as `Seq Scan` in PostgreSQL)
  - Scan each file block and test all records to see whether they satisfy the selection condition

  - Cost estimate = b_r block transfers + 1 seek
    - Assuming blocks of the file are stored contiguously
    - b_r denotes number of blocks containing records from relation r

- Although slower than other algorithms for implementing selection, linear search can be applied regardless of
  - Selection condition
  - Ordering of records in the file
  - Availability of indexes

Basic Linear Scan

- However, a full-table linear scan on extremely-large tables can be a disaster
  - E.g., billions of records in database
  - That's why we need other optimized ways

Index Scan

- **Index scan** – Search algorithms that use an index
  - Selection condition must be on search-key of index

```
select * from movies where movieid = 125;
```

We have a B+ tree index on `movieid`

- Plan: `Index Scan`

```
select * from movies where runtime = 100;
```

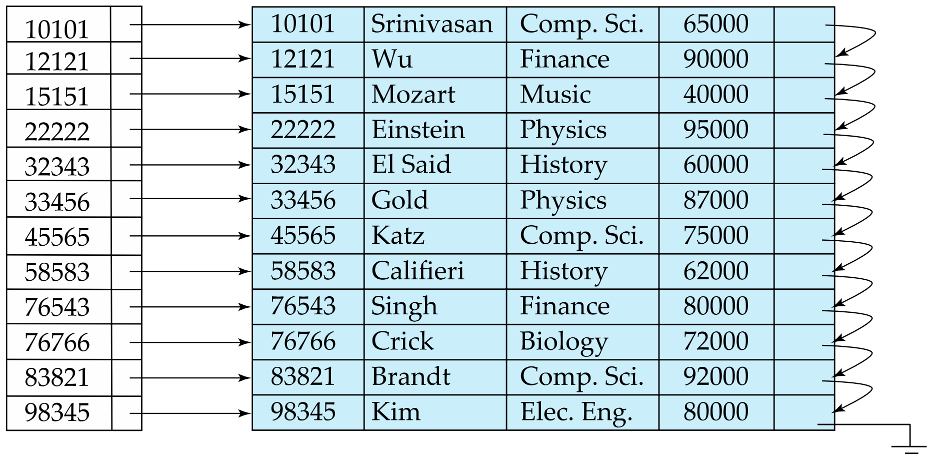
We don't have any index on `runtime`

- Plan: `Seq Scan`

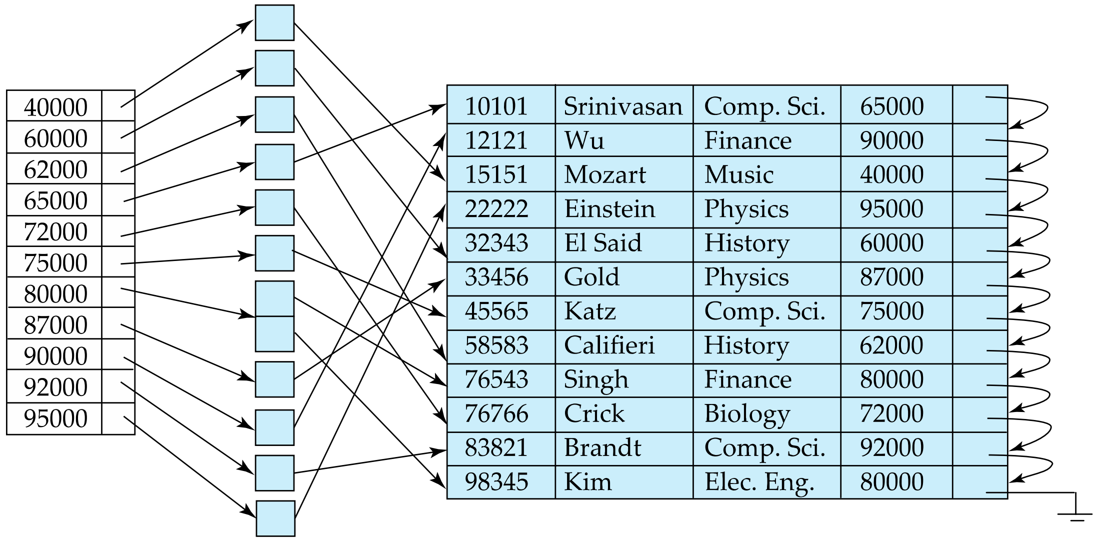
Index Scan

- **Index scan** – Search algorithms that use an index
  - Selection condition must be on search-key of index

- Unlike linear scan, we need to talk about different types of indexes and CONDITIONs
  - Clustered / Non-clustered index (Primary / Secondary index)
  - Equality / Comparison test



Clustered index



Non-clustered index

Index Scan

h_i: height of the B+-tree

Clustered index, equality on key

- Retrieve a single record that satisfies the corresponding equality condition
  - *key => no duplicated values*
  - *Cost* = $(h_i + 1) * (t_T + t_S)$
    - Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.

Clustered index, equality on non-key

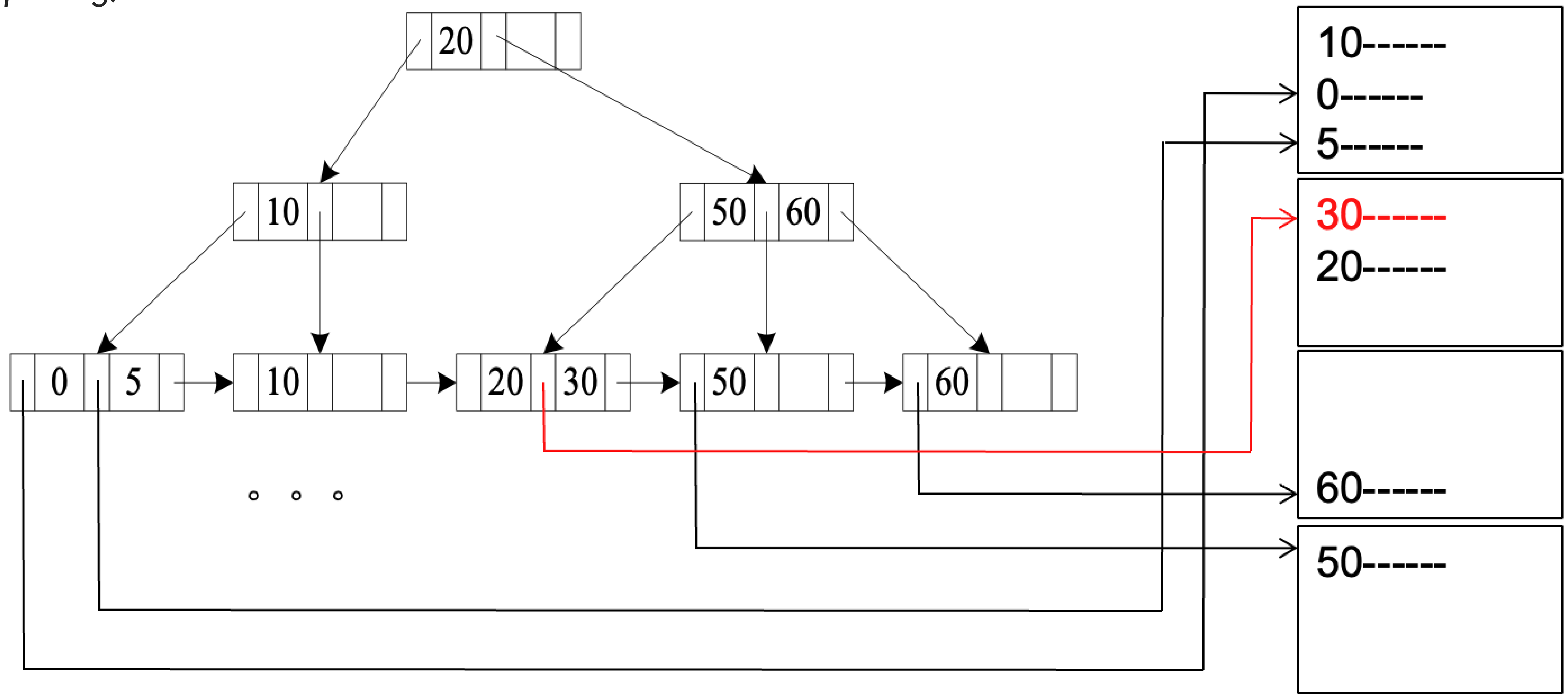
- Retrieve multiple records
  - *non key attributes => possible to have duplicated values*
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * \mathrm{b}$
    - One seek for each level of the tree, one seek for the first block
    - Let b = number of blocks containing matching records, which will be on consecutive blocks (since it is a clustering index) and don't require additional seeks

Index Scan

h_i: height of the B+-tree

Secondary index, equality on key/non-key

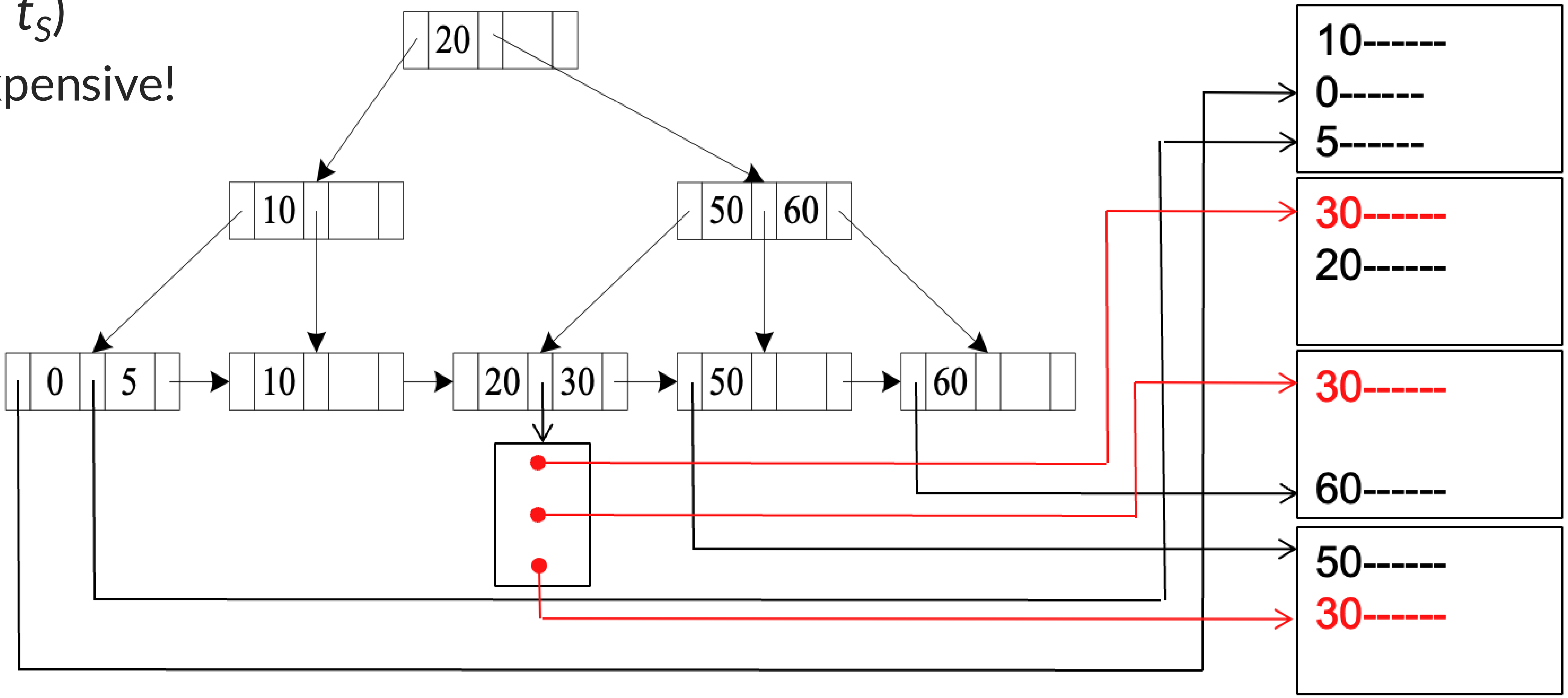
- Retrieve a single record if the search-key is a candidate key
  - $Cost = (h_i + 1) * (t_T + t_S)$



Index Scan

Secondary index, equality on key/non-key

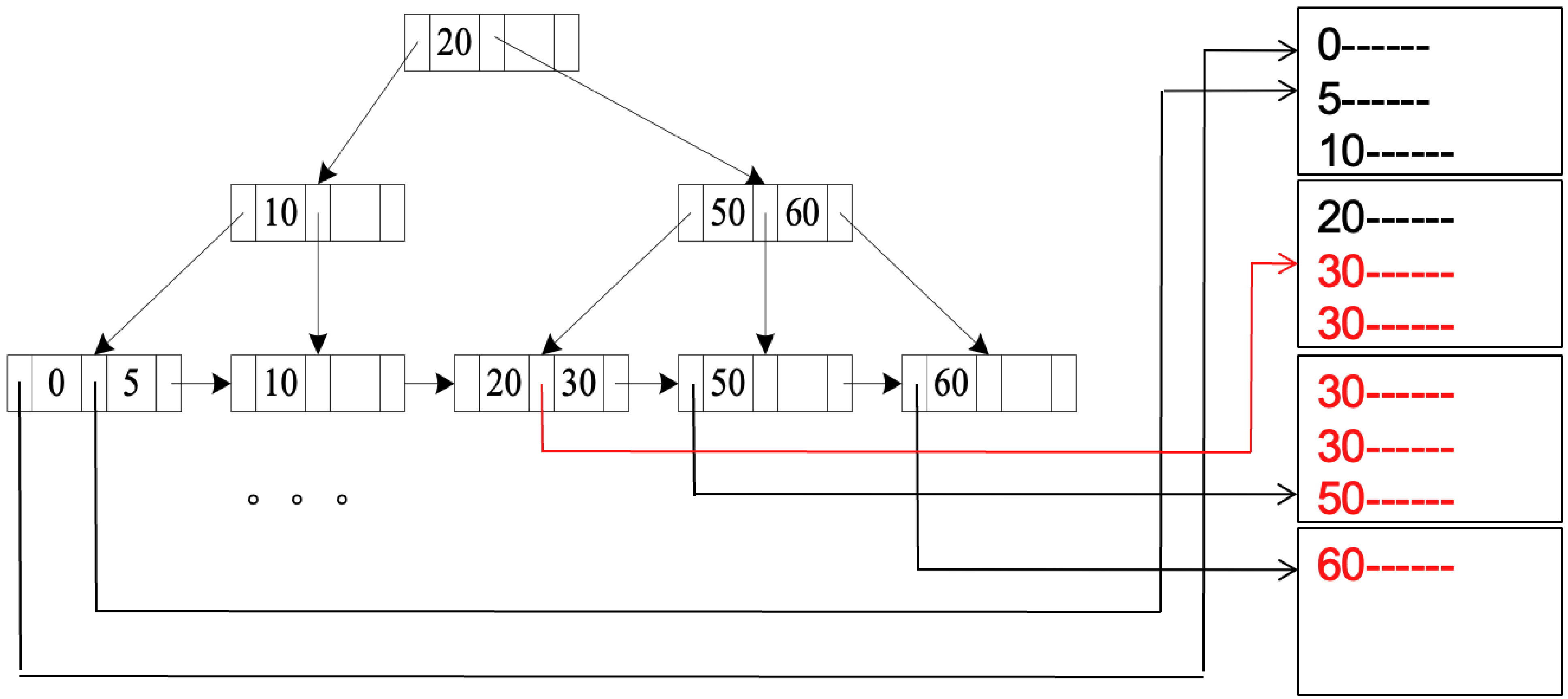
- Retrieve multiple records if search-key is not a candidate key
  - Each of *n* matching records may be on a different block
  - Cost = $(h_i + n) * (t_T + t_S)$
    - Can be very expensive!



Index Scan

Tip: Comparison tests can always be fulfilled with *linear scans*, which is the fallback solution

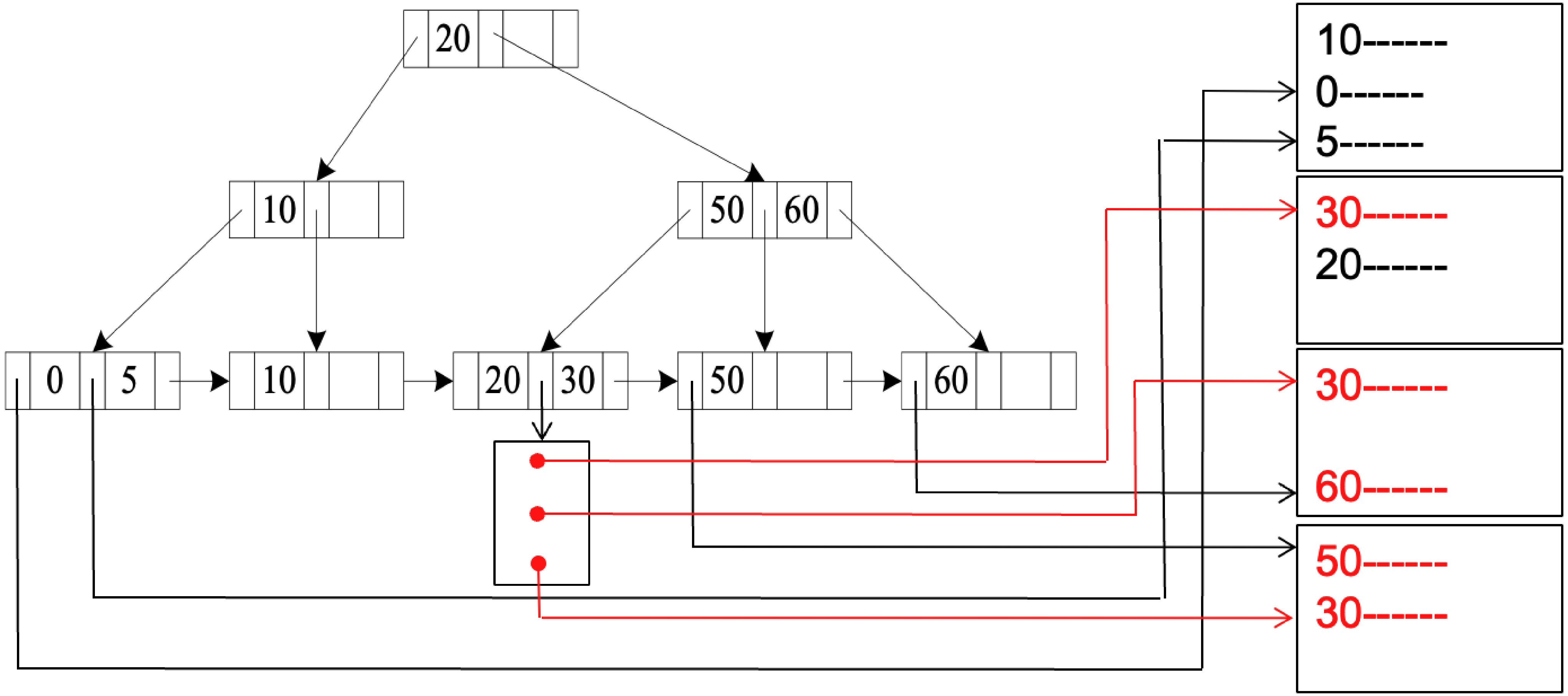
- **Clustered index, comparison** (i.e., Relation is sorted on *A*)
  - For $\sigma_{A \geq V}(r)$, use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$, just scan relation sequentially till first tuple > *v*; do not use index



Index Scan

- <u>**Non-clustered index, comparison**</u>
  - 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 per record
- Linear scan may be cheaper!



Complex Selections

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

- <u>Conjunctive selection using single-key index(es)</u>
  - Select a θ_i and algorithms mentioned above that results in the least cost for $\sigma_{\theta i}(r)$
  - Test other conditions on tuple after fetching it into memory buffer
- <u>Conjunctive selection using multi-key index</u>
  - Use appropriate composite (multiple-key) index if available
- <u>Conjunctive selection by intersection of identifiers</u>
  - Requires <u>indices</u> with record pointers/identifiers
    - Here, "indices" means the order number of records in the files, like array indices. They are not indexes
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - Then fetch records from file (and test remaining conditions)

Complex Selections

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$

Disjunctive selection by union of identifiers

(Similar to the third way on the previous page)

- Applicable if *all* conditions have available indexes
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers
    - Then fetch records from file
- Otherwise, just use linear scan
  - The disjunctive condition tested on each tuple during the scan

How `join` Works (with the help of indexes)

- Some widely-used `join` algorithms
  - Nested-loop join
  - Indexed nested-loop join
  - Merge join

Nested-loop Join

- To compute the *theta* join $r \bowtie_{\theta} s$

```
for each tuple tr in r do begin
   for each tuple ts in s do begin
     test pair (tr,ts) to see if they satisfy the join condition θ
     if they do, add tr • ts to the result.
   end
end
```

- r is called the **outer relation** and s the **inner relation** of the join
  - Think about the “outer loop” and the “inner loop” in programming

- Requires no indices and can be used with any kind of join condition
  - <u>Expensive</u> since it examines every pair of tuples in the two relations

Nested-loop Join via File Scan

- In the worst case, if the memory can only hold one block of each relation, the estimated cost is:
  - $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks
    - n_r - number of records in relation r
    - $b_{s,}$ b_r - number of blocks in relation s and r
    - $n_r * b_s + b_r$: for each record in r, need to read all blocks in s; and need to read all b_r blocks in r
    - $n_r + b_r$: for each record in r, need to do seek once for s; and need to seek r for b_r times
      - Assuming r and s are stored contiguously on disks
- If the smaller relation fits entirely in memory, use that as the inner relation
  - Reduces cost to $b_r + b_s$ block transfers and 1 + 1 seeks

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join ($r \bowtie_{r.A=s.B} s$) 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 to look up tuples in s that satisfy the join condition with tuple t_r
  - Essentially a selection operation given the values of the joining attribute in t_r

Indexed Nested-Loop Join

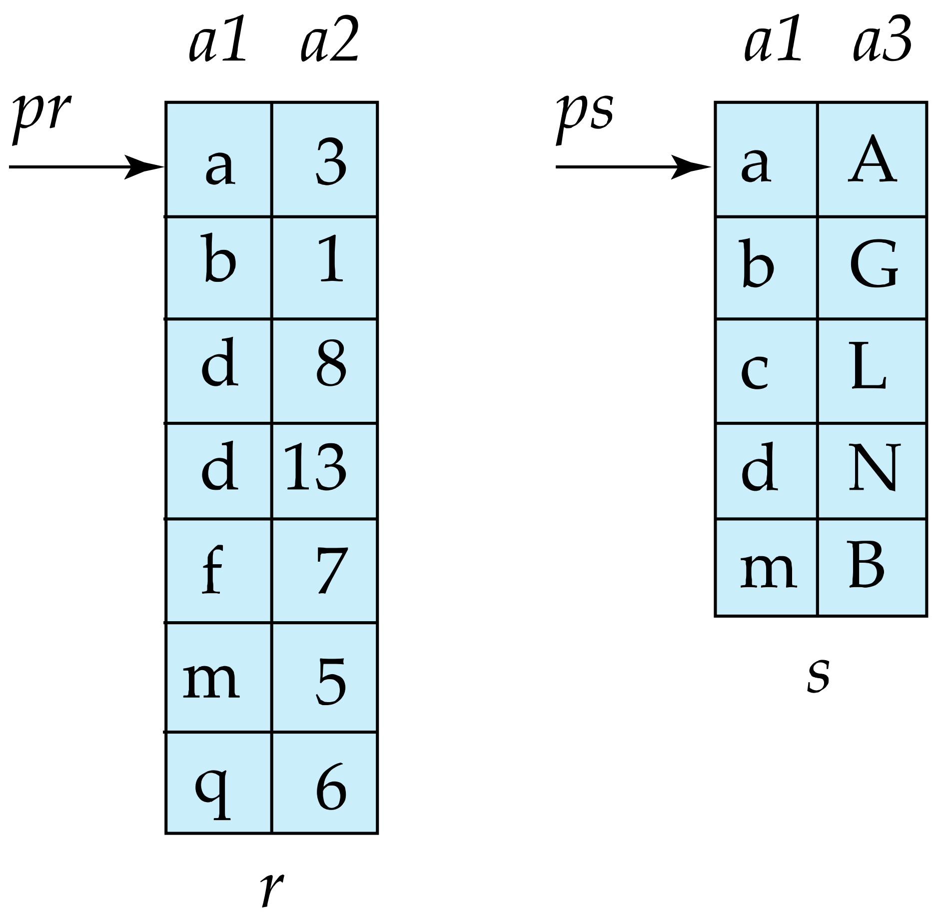
- Worst case: Buffer in memory has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s*

- 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 of *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition
  - We need b_r seeks as the disk head may have moved between each I/O

- If indices are available on join attributes of both *r* and *s*, use the relation with fewer tuples as the outer relation

Merge Join

- a.k.a., sort-merge join
  - Zipper-like joining
- Steps
  - Sort both relations on their join attribute (if not already sorted on the join attributes)
  - Merge the sorted relations to join them
    - if r.a1[pr] < s.a1[ps], pr++
    - elif r.a1[pr] > s.a1[ps], ps++
    - else, join and move pr and ps
- 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



Merge Join

- Can be used only for equi-joins ($r \bowtie_{r.A=s.B} s$) and natural joins

- Once the relations are in sorted order, each block needs to be read only once
  - Assuming all tuples for any given value of the join attributes fit in memory

- 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

b_b: memory buffer size, counted in number of blocks, for each relation

Hash Join

- Hash join
  - Build a set of buckets for a smaller table to speed up the data lookup

- Procedure:
  - 1. Create a hash table for the smaller table t1 in the memory
  - 2. Scan the larger table t2. For each record r,
    - 2.1 Compute the hash value of r.`join_attribute`
    - 2.2 Map to corresponding rows in t1 using the hash table

Principles of Database Systems (CS307)

Lecture 14 part 2: Query Optimization

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Query Optimization

- Purpose of query optimization
  - Select an effective way to retrieve the data based on queries while spending the least computational effort
    - However, it is only “spending less computational effort” in most scenarios, not least

Query Optimization

- Users don’t need to consider the best way of writing queries
  - We want DBMS to construct a query-evaluation plan that minimizes the cost of query evaluation
- Automated optimization can perform better (for most of the time)
  - Utilize the data dictionary
  - Real-time utilization based on physical storage changes
  - Optimizer can evaluate hundreds of execution plans in a very short time compared with human programmers
  - Human users do not need to learn advanced optimization techniques any more, which is conducted by optimizers instead

An Example in the Movie Dataset

- The same query can be represented in different plans
  - E.g., retrieve the titles of those movies from China

```sql
select m.title
from movies m, countries c
where m.country = c.country_code and c.country_name = 'China';
```

An Example in the Movie Dataset

- The corresponding relational algebra expressions:

(1) $\Pi_{title}\ (\sigma_{movies.country\ =\ countries.country\_code\ \land\ \text{countries.country}\ =\ \text{“}China\text{”}}(\text{movies} \times \text{countries}))$

(2) $\Pi_{title}\ (\sigma_{\text{countries.country}\ =\ \text{“}China\text{”}}\ (\text{movies} \bowtie_{movies.country\ =\ countries.country\_code}\ \text{countries}))$

(3) $\Pi_{title}\ (\text{movies} \bowtie_{movies.country\ =\ countries.country\_code} (\sigma_{\text{countries.country}\ =\ \text{“}China\text{”}}\ (\text{countries}))$

An Example in the Movie Dataset

- The corresponding relational algebra expressions:

(1) $\Pi_{title}\ (\sigma_{movies.country\ =\ countries.country\_code\ \wedge\ \text{countries.country}\ =\ \text{“China”}}(\text{movies} \times \text{countries}))$

(2) $\Pi_{title}\ (\sigma_{\text{countries.country}\ =\ \text{“China”}}\ (\text{movies} \bowtie_{movies.country\ =\ countries.country\_code}\ \text{countries}))$

(3) $\Pi_{title}\ (\text{movies} \bowtie_{movies.country\ =\ countries.country\_code}\ (\sigma_{\text{countries.country}\ =\ \text{“China”}}\ (\text{countries})))$

- In (1), a full Cartesian product will be computed, which costs huge time for matching all pairs and massive temporary storage space for the intermediate product table
- In (2), a smaller intermediate join table is to be cached, which saves some space
- In (3), the filter ($\sigma_{\text{c.country}\ =\ \text{“China”}}$) reduces the size of the right table in the join operation, which saves a lot of time for pair matching and caching intermediate join table

An Example in the Movie Dataset

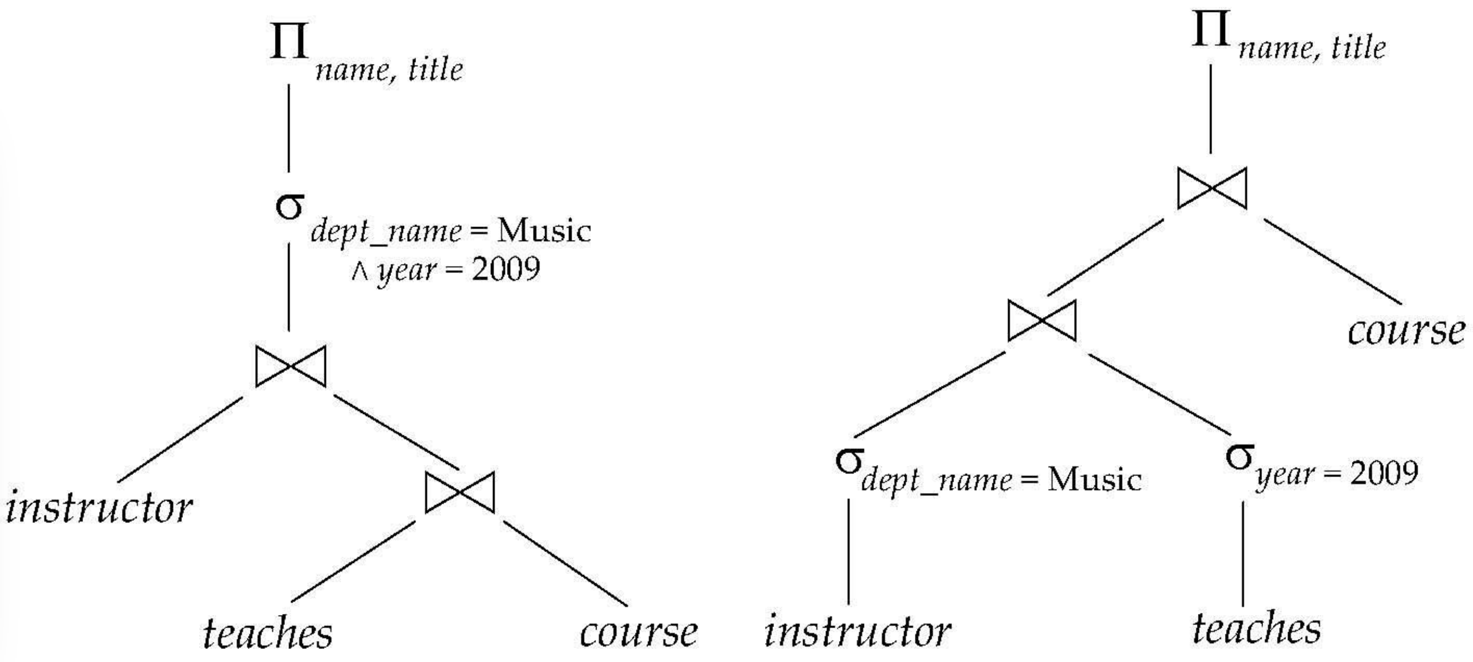
- In addition, the filter operation can be further accelerated once an index is built upon the *country* column

$$(3)\ \Pi_{title}\ (\text{movies} \bowtie_{movies.country\ =\ countries.country\_code} (\sigma_{\text{countries.country}\ =\ \text{“}China\text{”}} (\text{countries})))$$

Generating Equivalent Expressions

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

```
select name, title
from instructor
    natural join (teaches natural join course)
where dept_name = 'Music' and year = 2009;
```

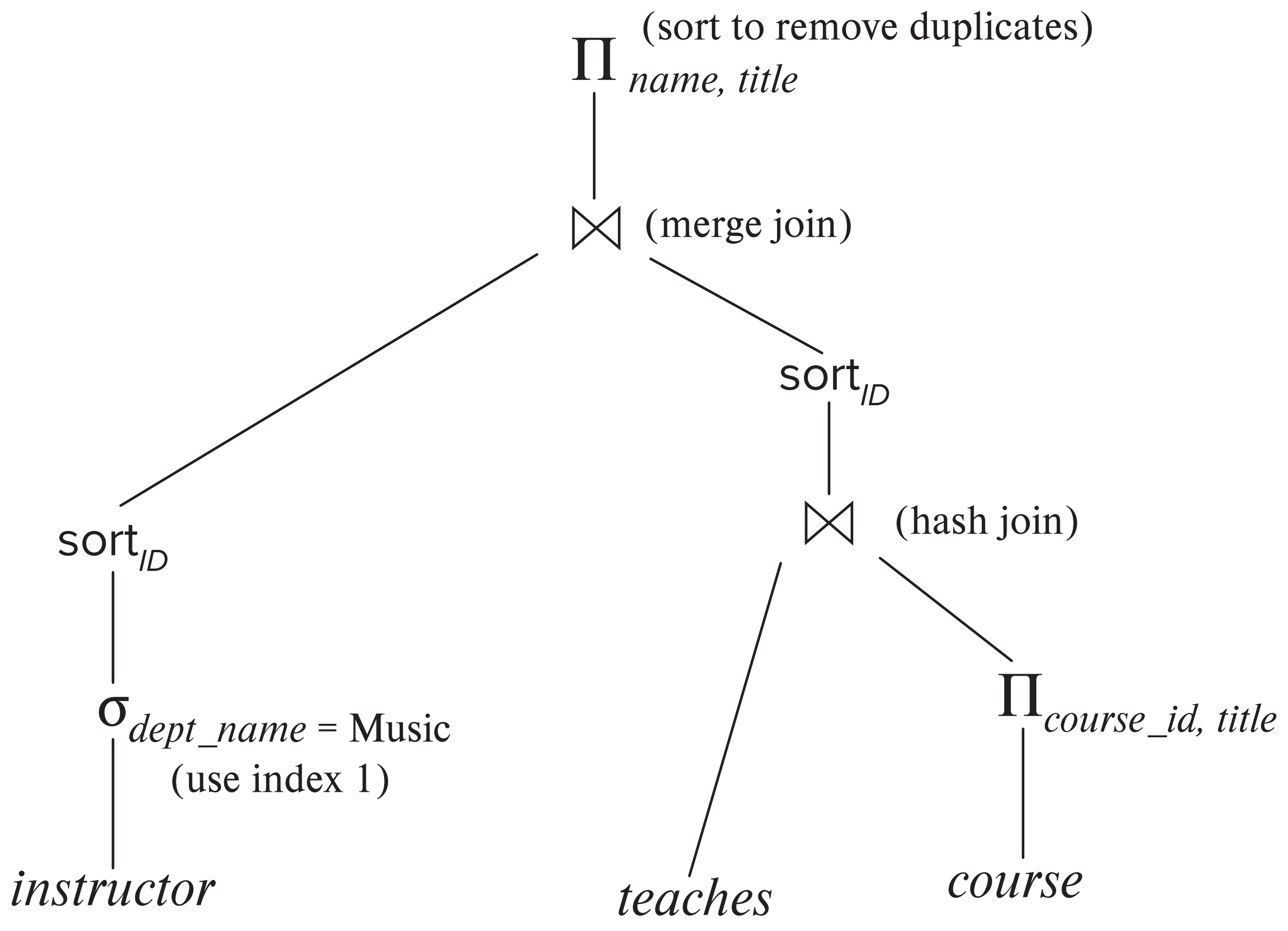


(a) Initial expression tree (b) Tree after multiple transformations

Generating Equivalent Expressions

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

```
select name, title
from instructor
    natural join (teaches natural join course)
where dept_name = 'Music' and year = 2009;
```



Transformation of Relational Expressions

- 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 don't care if they generate different results on databases that violate integrity constraints

- An equivalence rule says that expressions of two forms are equivalent
  - ... i.e., we can replace expression of the first form by second, or vice versa
  - The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions

Equivalence Rules

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

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

2. Selection operations are commutative

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

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

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$$

where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins

   a) $\sigma_\theta(E_1 \times E_2) \equiv E_1 \bowtie_\theta E_2$ (same as the definition of theta-join)

   b) $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Equivalence Rules

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

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

6. (a) Natural join operations are associative:

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

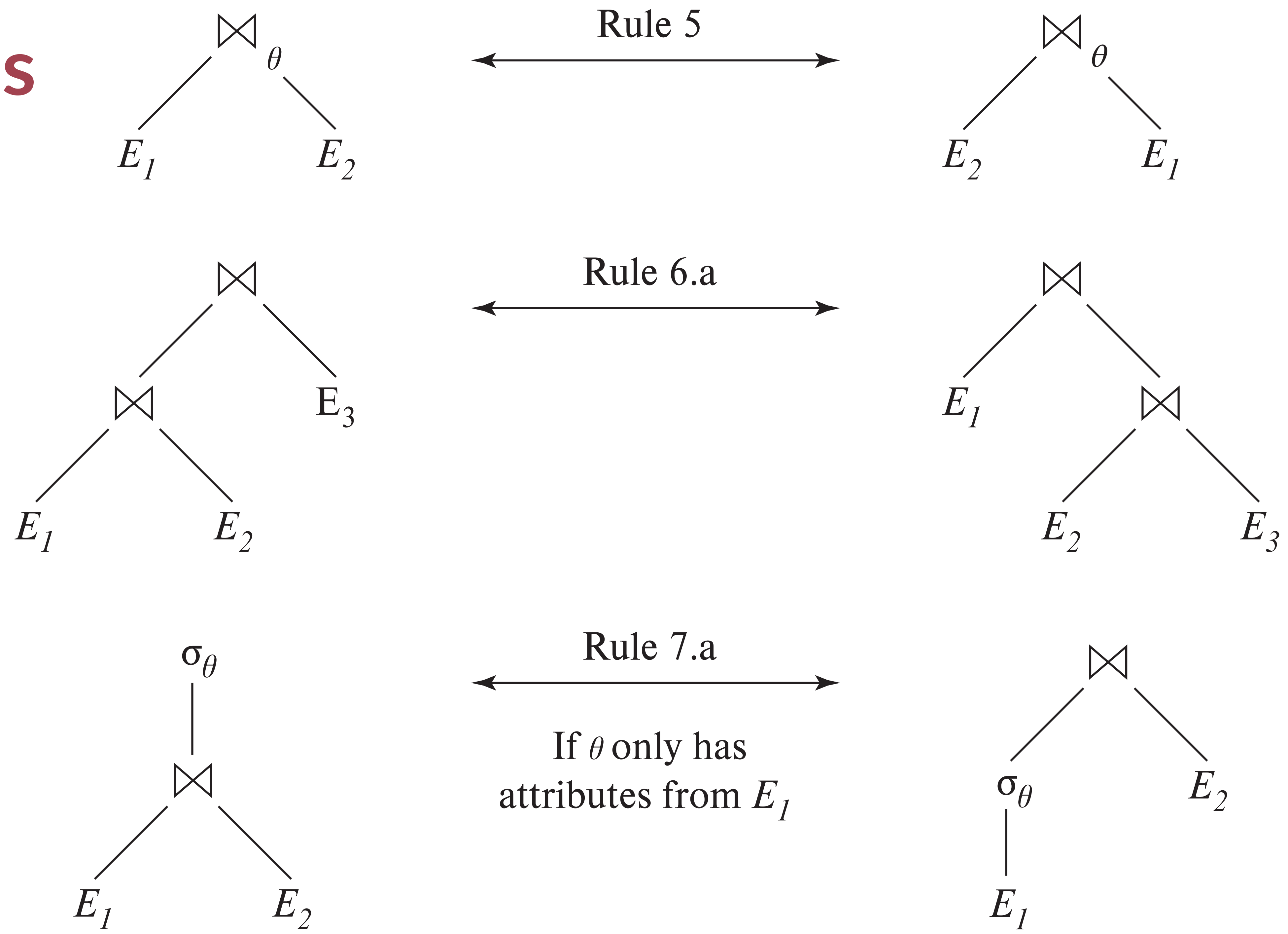
(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3

Equivalence Rules

- Representation of Rule 5, 6(a) and 6(b) with diagrams



Equivalence Rules

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:

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

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

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

Perform selection early

Equivalence Rules

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

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

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively,

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

(b) In general, 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, and,
- Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in L_2:

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

* Similar equivalences hold for left, right, and full outer join operations: ⟕, ⟖, and ⟗

Equivalence Rules

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$
$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

- However, set difference is not commutative

10. Set union and intersection are associative

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

Equivalence Rules

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

(a) $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$

(b) $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$

(c) $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$

(d) $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$

(e) $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

12. The projection operation distributes over union

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

Transformation Example: Pushing Selections

- Query: Find the names of all instructors in the Music department, along with the titles of the courses (in the Music department) that they teach

```
select name, title
from instructor natural join (teaches natural join course
where dept_name = 'Music';
```

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

- Transformation using rule 7(a):
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\ \text{'Music'}}\ (instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course)))$

Perform selection <u>as early as possible</u> reduces the size of the relation to be joined

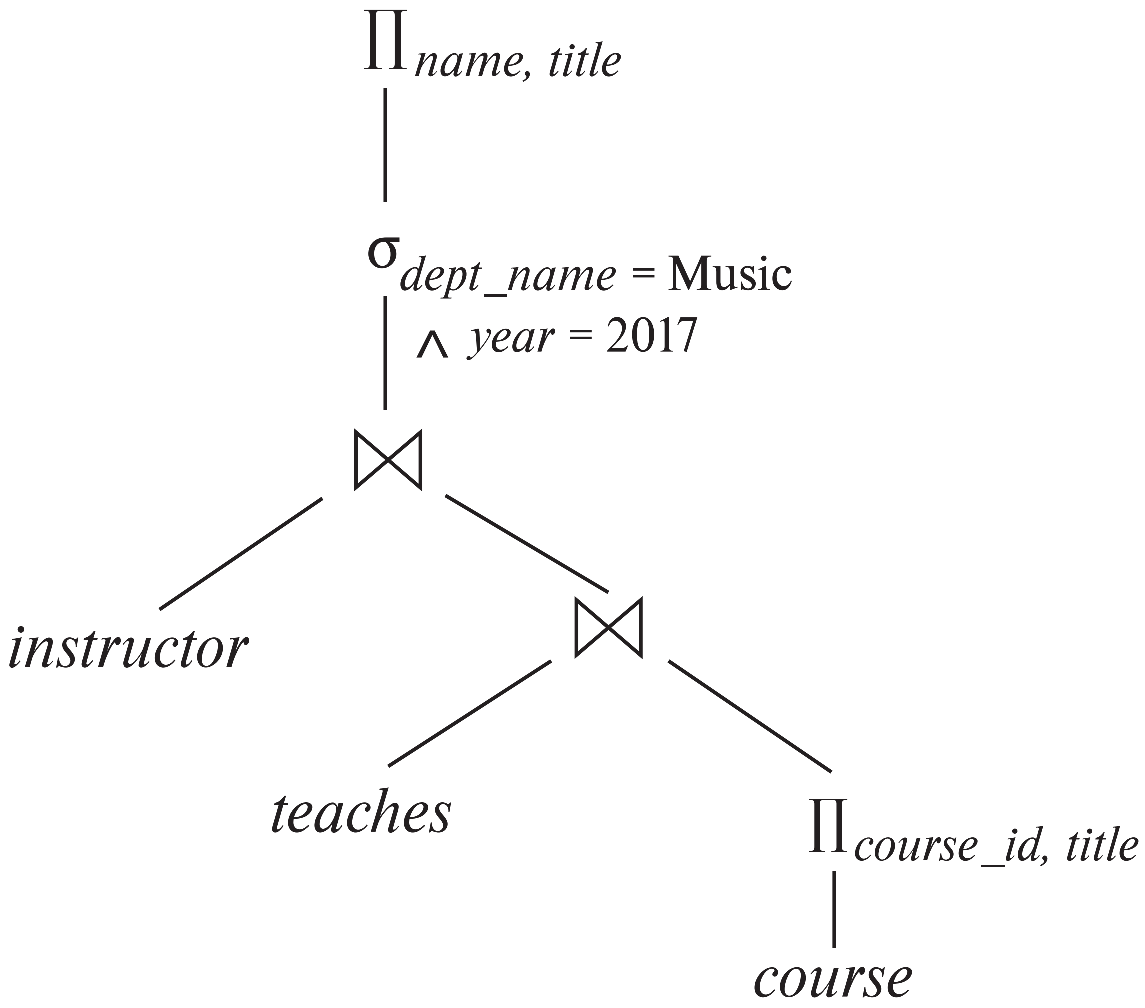
Transformation Example: Multiple Transformations

- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\ \text{"Music"} \wedge year\ =\ 2017}\ (instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course))))$

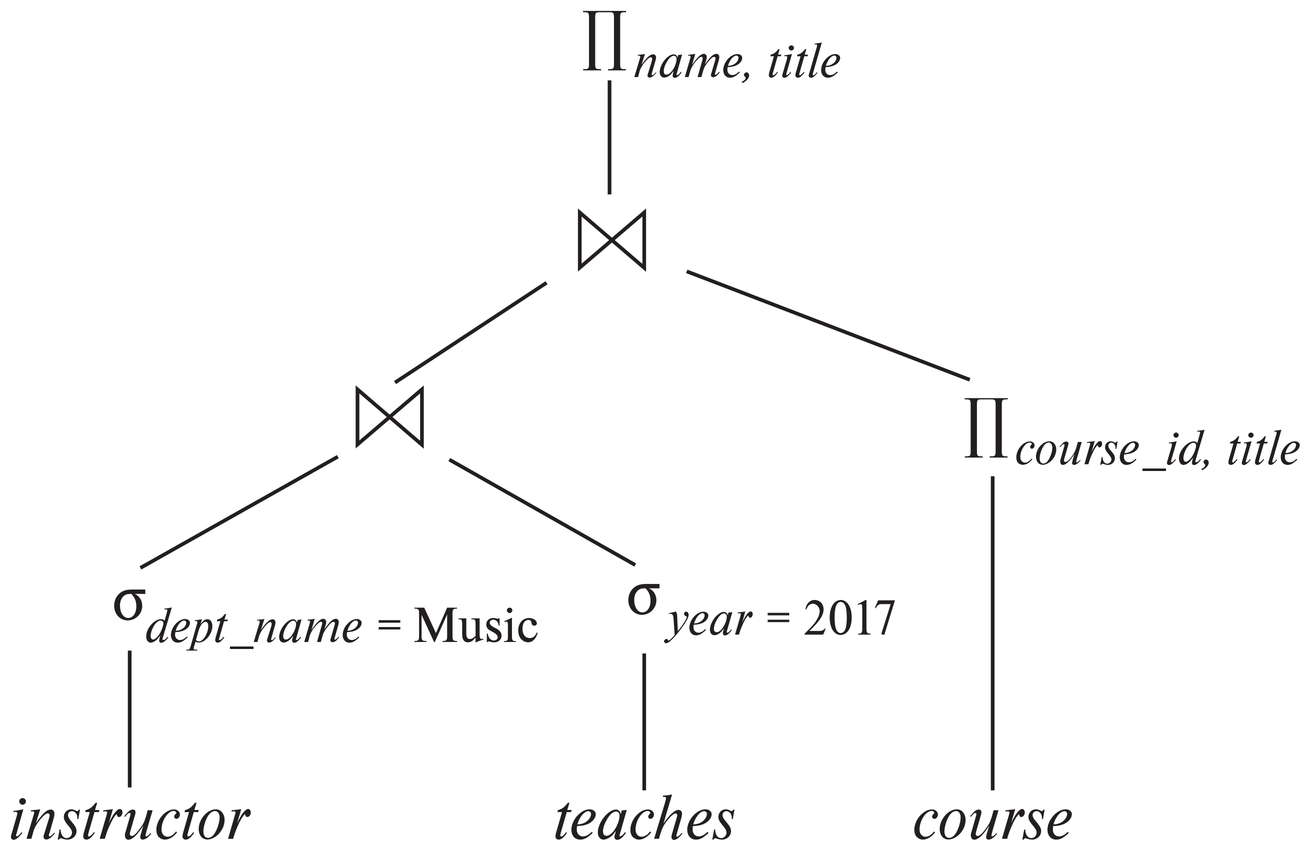
- Transformation using join associatively (Rule 6(a)):
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\ \text{"Music"} \wedge year\ =\ 2017}\ ((instructor \bowtie teaches) \bowtie\ \Pi_{course\_id,\ title}\ (course)))$

- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression:
  - $\sigma_{dept\_name\ =\ \text{"Music"}}\ (instructor) \bowtie \sigma_{\ year\ =\ 2017}\ (teaches)$

Transformation Example: Multiple Transformations



(a) Initial expression tree



(b) Tree after multiple transformations

* Transformation Example: Pushing Projections

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

- When we compute

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

  we obtain a relation whose schema is:

  (ID, name, dept_name, salary, course_id, sec_id, semester, year)

- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

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

Perform projections <u>as early as possible</u> reduces the size of the relation to be joined

Join Ordering Example

- 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) $\bowtie$

- 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

Cost Estimation

- 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

Cost Estimation

- Estimation of plan cost based on:
  - Statistical information about relations, such as:
    - 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

For more, please refer to Section 16.3 “Estimating Statistics of Expression Results” in the reference textbook